DSP System Toolbox™

Reference

MATLAB® & SIMULINK®

R2017a

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

DSP System Toolbox™ Reference

Revision History

April 2011	Online only	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)
October 2014	Online only	Revised for Version 8.7 (R2014b)
March 2015	Online only	Revised for Version 9.0 (R2015a)
September 2015	Online only	Revised for Version 9.1 (R2015b)
March 2016	Online only	Revised for Version 9.2 (R2016a)
September 2016	Online only	Revised for Version 9.3 (R2016b)
March 2017	Online only	Revised for Version 9.4 (R2017a)

Contents

1 Apps in DSP System Toolbox

2 Blocks — Alphabetical List

3 Analysis Methods for Filter System Objects

Analysis Methods for Filter System Objects 3-2

4 System Objects — Alphabetical List

5 Functions — Alphabetical List

6 Reference for the Properties of Filter Objects

Fixed-Point Filter Properties **6-2**
Overview of Fixed-Point Filters **6-2**
Fixed-Point Objects and Filters **6-2**
Summary — Fixed-Point Filter Properties **6-4**
Property Details for Fixed-Point Filters **6-17**

Adaptive Filter Properties **6-92**
Compatibility **6-92**
Property Summaries **6-92**
Property Details for Adaptive Filter Properties **6-96**
References for Adaptive Filters **6-103**

Multirate Filter Properties **6-104**
Compatibility **6-104**
Property Summaries **6-104**
Property Details for Multirate Filter Properties **6-108**
References for Multirate Filters **6-118**

Glossary

Apps in DSP System Toolbox

Filter Builder

Design filters starting with frequency and magnitude specifications (`filterBuilder`)

Description

The **Filter Builder** app enables you to choose a filter response and build a filter object based on the filter design specifications. The app designs the filter and exports the filter object to the base workspace. For more information, see the `filterBuilder` documentation.

Note: This app requires a screen resolution greater than 640 × 480.

Open the Filter Builder App

- MATLAB® Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.
- MATLAB command prompt: Enter `filterBuilder`.

Examples

- "Filter Builder Design Process"
- "Lowpass FIR Filter Design"

See Also

See Also

Apps

Filter Designer | Window Designer

Functions

`designfilt` | `filterBuilder` | `fvtool` | `sptool` | `wvtool`

Topics

“Filter Builder Design Process”
“Lowpass FIR Filter Design”

Introduced before R2006a

Logic Analyzer

Visualize, measure, and analyze transitions and states over time

Description

The **Logic Analyzer** is a tool for visualizing and inspecting signals in your Simulink® model. Using the **Logic Analyzer**, you can:

- Debug and analyze models
- Trace and correlate many signals simultaneously
- Detect and analyze timing violations
- Trace system execution

The Logic Analyzer does not support frame-based processing.

If you enable the Simulink **"Log Dataset data to file" (Simulink)** configuration parameter, to stream logged data to the Logic Analyzer set the `VisualizeLoggedSignalsWhenLoggingToFile` model parameter to `'on'`.

For keyboard shortcuts, click `more`.

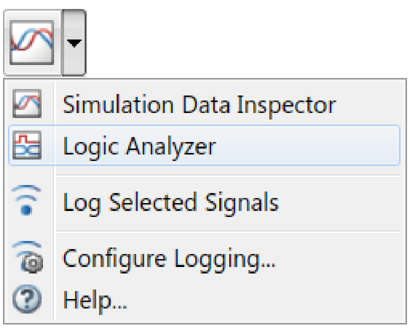
Keyboard Shortcuts

Actions	Description	Applicable When
Ctrl+X	Cut	Wave is selected
Ctrl+C	Copy	Wave is selected
Ctrl+V	Paste	Wave is selected
Delete	Delete	Wave is selected
Ctrl+-	Zoom out	Always
Shift+Ctrl+-	Zoom out around active cursor	Always
Ctrl++	Zoom in	Always
Shift+Ctrl++	Zoom out around active cursor	Always

Actions	Description	Applicable When
Shift+Ctrl+C	Move display to active cursor	When cursor is not in the display range
Space	Zoom out full	Always
Tab, **Right Arrow**	Next transition	Digital format wave is selected
Shift+Tab, **Left Arrow**	Previous transition	Digital format wave is selected
Ctrl+A	Select all waves	Always
Up Arrow	Select wave above selected	Wave is selected
Down Arrow	Select wave below selection	Wave is selected
Ctrl+Up Arrow	Move selected waves up	Wave is selected
Ctrl+Down Arrow	Move selected waves down	Wave is selected
Escape	Unselect all signals	Wave is selected
Page Up	Scroll up	Always
Page Down	Scroll down	Always

Open the Logic Analyzer App

- Simulink Toolbar: Click the Logic Analyzer button . If the button is not displayed, click the Simulation Data Inspector button arrow and select **Logic Analyzer** from the menu.



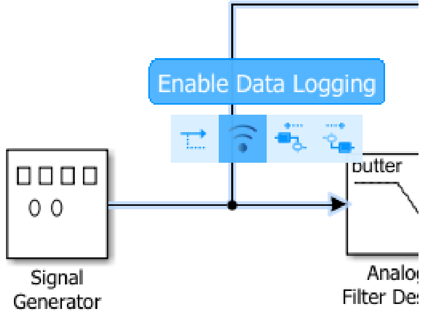
Your most recent choice for data visualization is saved across Simulink sessions.

Examples

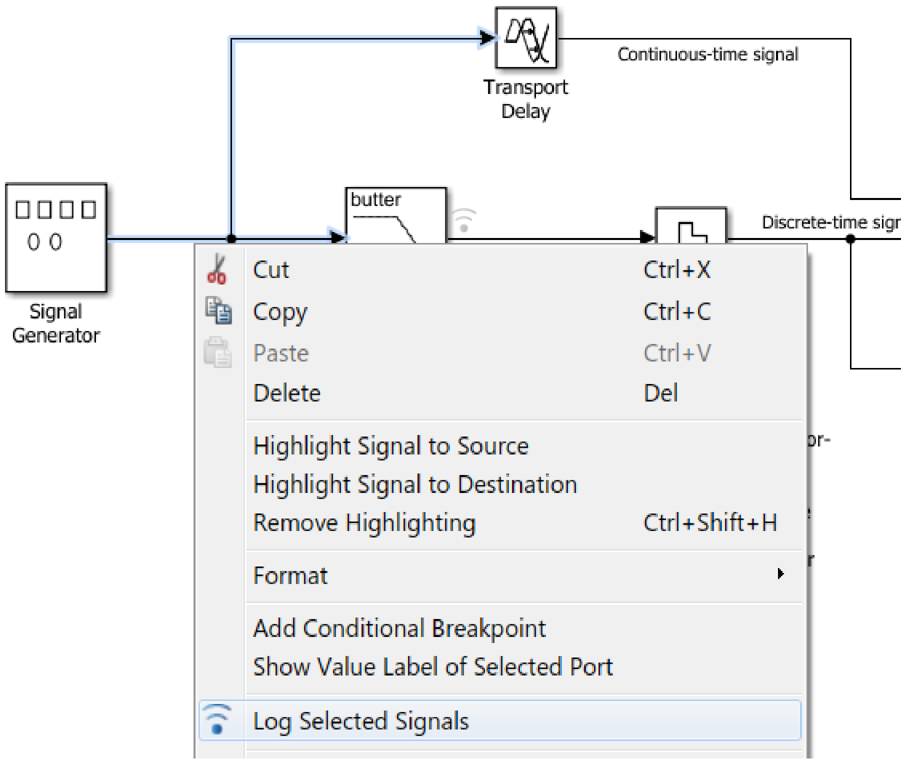
Select Signals to Analyze

The **Logic Analyzer** supports several methods for selecting data to visualize.

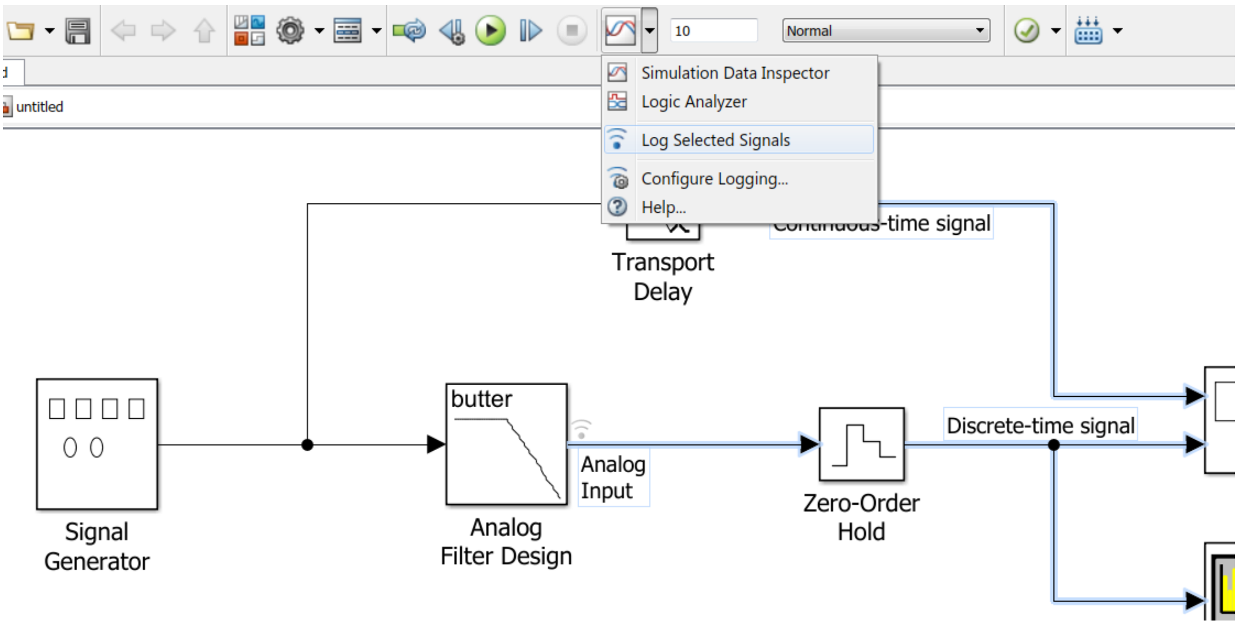
- Select a signal in your model. When you select a signal, an ellipsis appears above the signal line. Hover over the ellipsis to view options and then select the **Enable Data Logging** option.



- Right-click a signal in your model to open an options dialog box. Select the **Log Selected Signals** option.



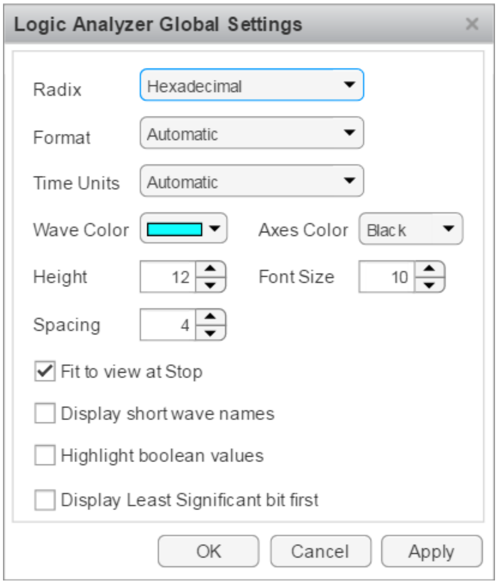
- Use any method to select multiple signal lines in your model, for example, use **Shift**+click to select multiple lines individually or **CTRL+A** to select all lines at once. Then click the Logic Analyzer button arrow and select **Log Selected Signals**.



When you open the **Logic Analyzer**, all signals marked for logging are listed. You can add and delete waves from your **Logic Analyzer** while it is open.

Modify Global Settings

Open the **Logic Analyzer** and select **Settings** from the toolstrip. A global settings dialog box opens. Any setting you change for an individual signal supersedes the global setting.



Set the display **Radix** of your signals as one of the following:

- `Hexadecimal` — Displays values as symbols from zero to nine and A to F
- `Octal` — Displays values as numbers from zero to seven
- `Binary` — Displays values as zeros and ones
- `Signed decimal` — Displays the signed, stored integer value
- `Unsigned decimal` — Displays the stored integer value

Set the display **Format** as one of the following:

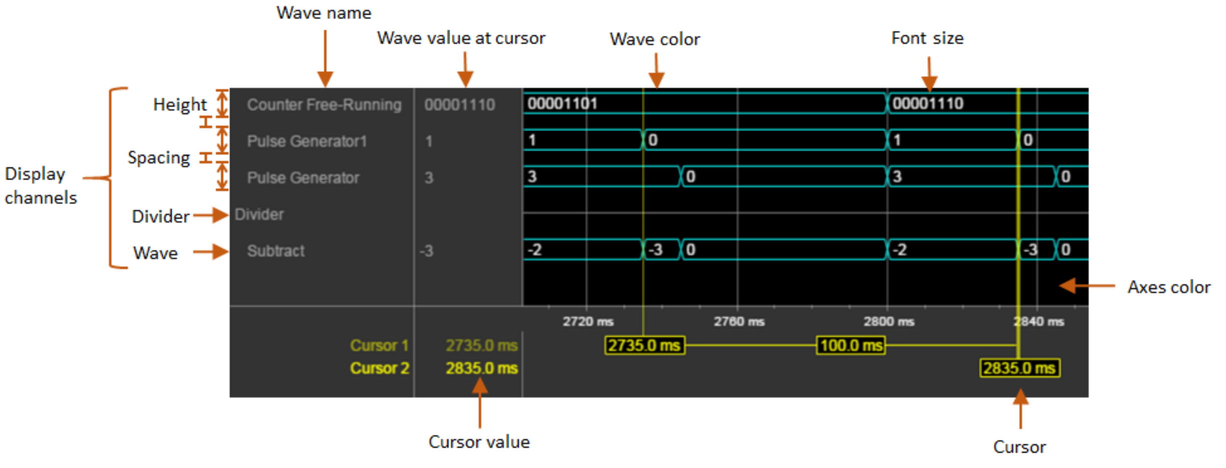
- `Automatic` — Displays floating point signals in `Analog` format and integer and fixed-point signals in `Digital` format. Boolean signals are displayed as zero or one.
- `Analog` — Displays values as an analog plot
- `Digital` — Displays values as digital transitions

Set the display **Time Units** to one of the following:

- `Automatic` — Uses a time scale appropriate to the time range shown in the current plot

- `seconds`
- `milliseconds`
- `microseconds`
- `nanoseconds`
- `picoseconds`
- `femtoseconds`

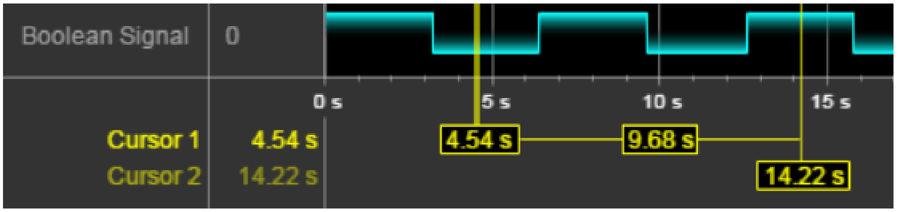
Inspect the graphic for an explanation of the global settings: `Wave Color`, `Axes Color`, `Height`, `Font Size`, and `Spacing`. `Font Size` applies only to the text within the axes.



Select **Fit to view at Stop** to plot data over the entire time range in one screen when you stop the simulation.

Select **Display short wave names** to use the short name of waves to display. Short wave names do not include path information.

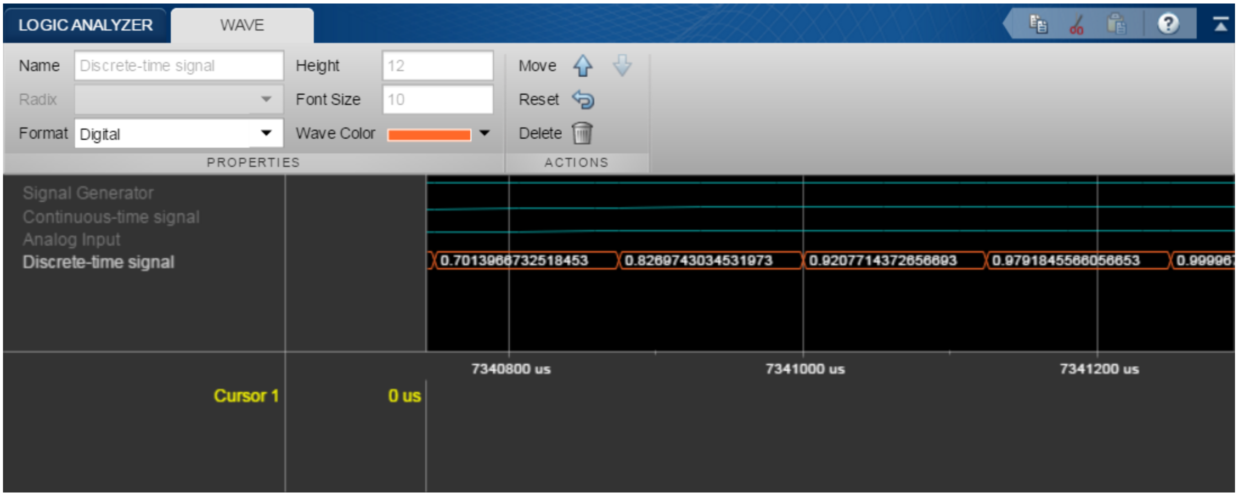
Select **Highlight boolean values** to add highlighting to Boolean signals. If the signal value is `true`, the highlight fades out below. If the signal value is `false`, the signal fades out above. This property enables you to visually deduce the value of the signal.



Select **Display Least Significant bit first** to reverse the display order of bits for bit-expanded fixed-point and integer signals.

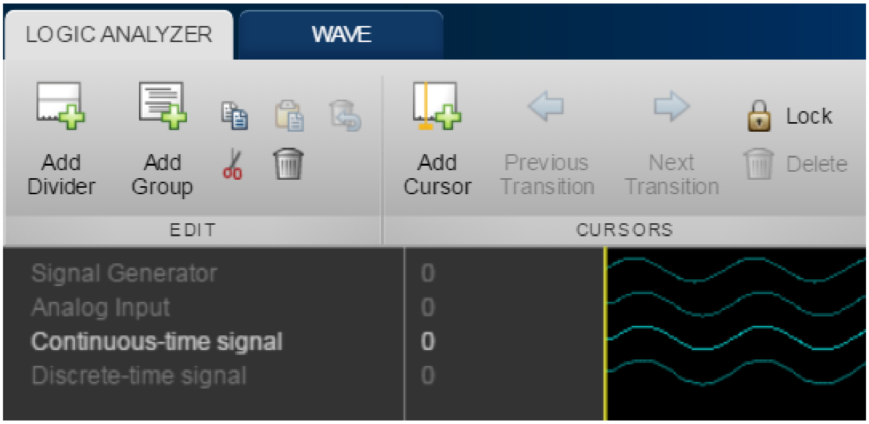
Modify Individual Wave Settings

Open the **Logic Analyzer** and select a wave by double-clicking the wave name. Then from the **Wave** tab, set parameters specific to the individual wave you selected. Any settings made on individual signals supersedes the global setting. To return individual wave parameters to the global settings, click **Reset**.



Delete and Restore Waves

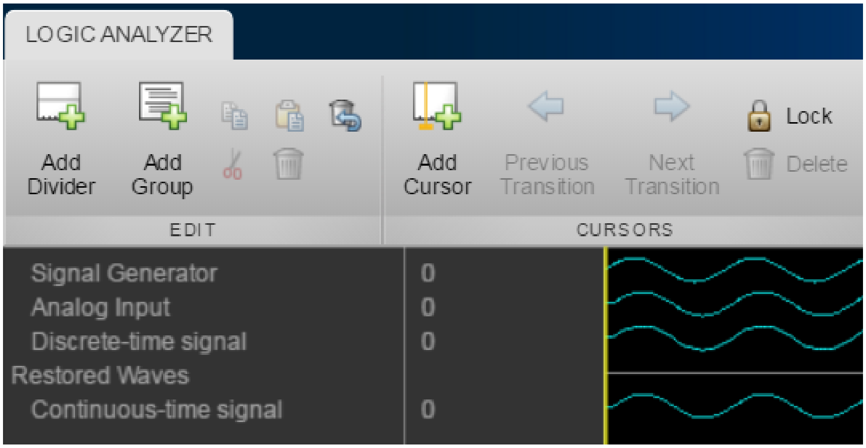
1 Open the **Logic Analyzer** and select a wave by clicking the wave name.



2 From the **Logic Analyzer** toolstrip, click . The wave is removed from the **Logic Analyzer**.

3 To restore the wave, from the **Logic Analyzer** toolstrip, click .

A divider named **Restored Waves** is added to the bottom of your channels, with all deleted waves placed below it.

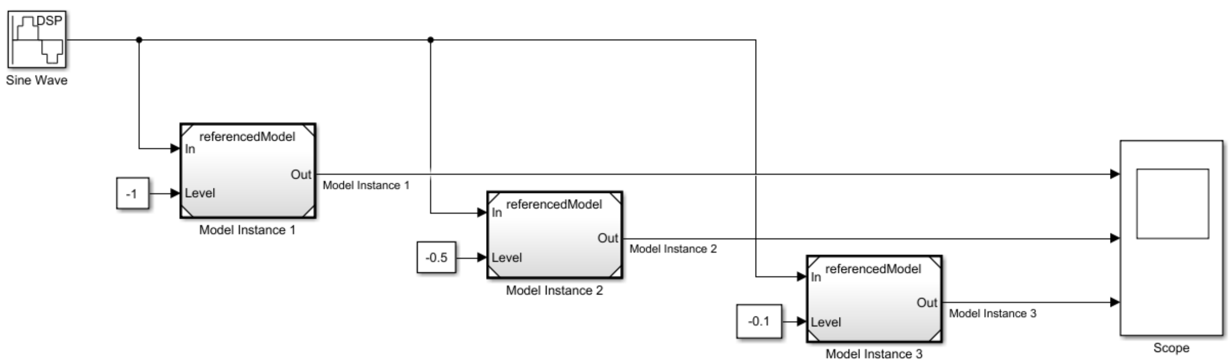


Choose Visible Instance of Multi-Instance Model Block

The **Logic Analyzer** can stream only a single instance of a multi-instance Model block. This example shows how to select an instance of a multi-instance Model block for streaming on the **Logic Analyzer**.

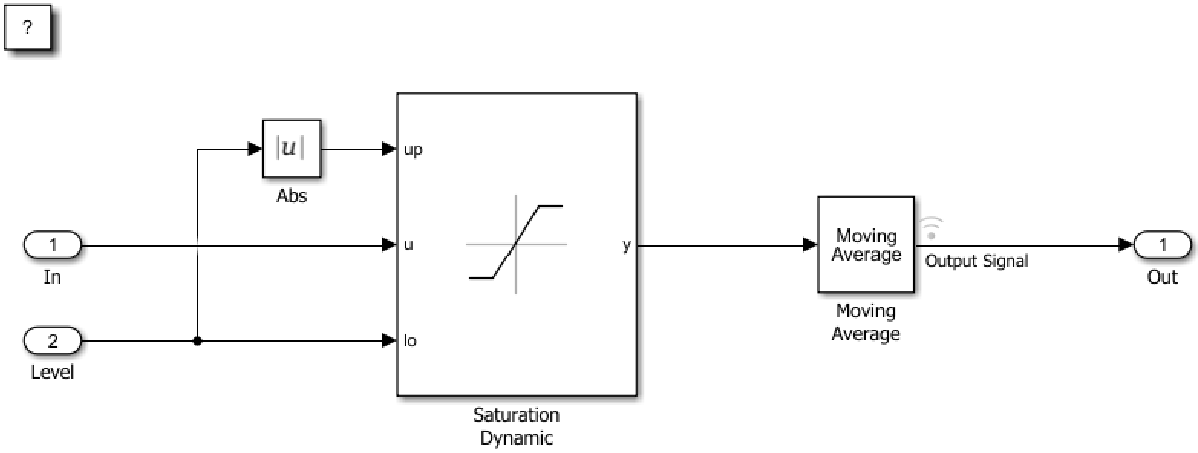
1 Open the multipleModelInstances model:

```
open_system([matlabroot,'\examples\dsp\multipleModelInstances.slx'])
```

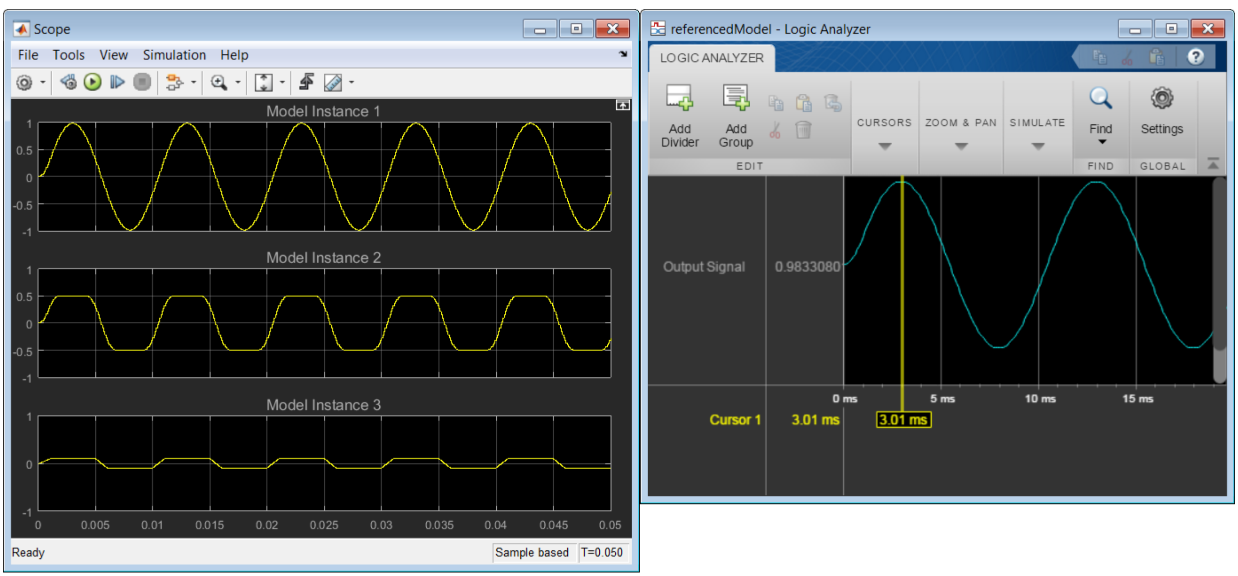


The model contains three instances of the `referencedModel` model.

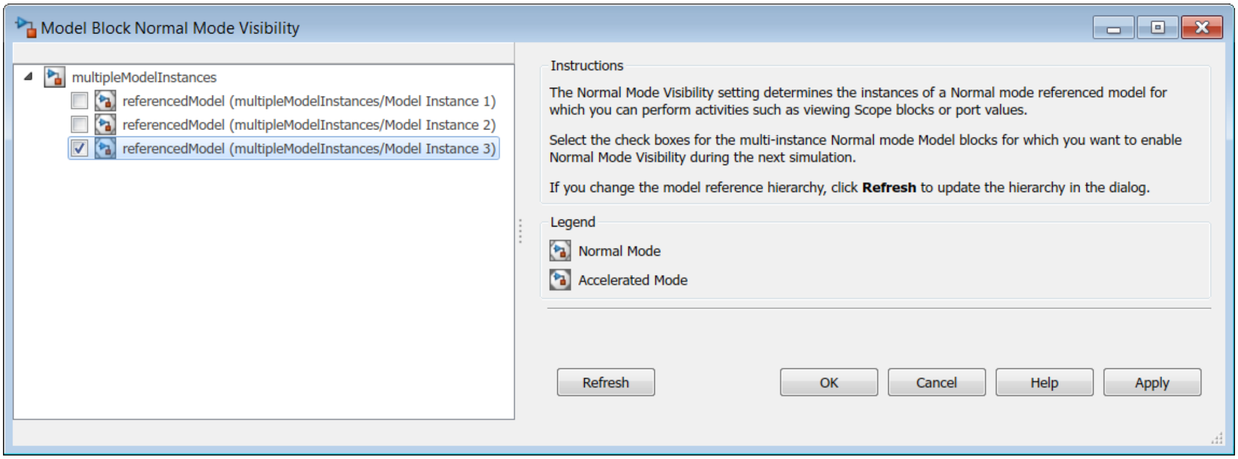
2 Double-click any of the Model blocks to open the model referenced by all three Model blocks.



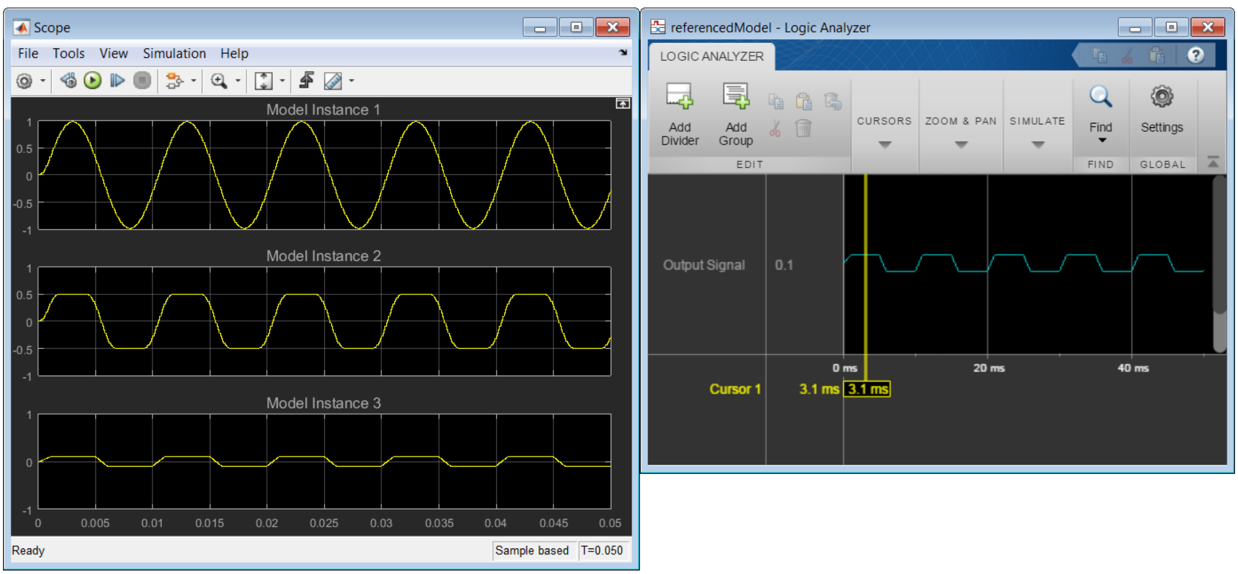
3 Open the **Logic Analyzer** in the referenced model.
4 From the `multipleModelInstances` model, run the model. The **Logic Analyzer** displays a single instance of a multi-instant Model block.



5 To switch between instances, from the Simulink Editor menu, select **Diagram** > **Subsystem & Model Reference** > **Model Block Normal Mode Visibility**. Select **Model Instance 3** and then click **OK**.

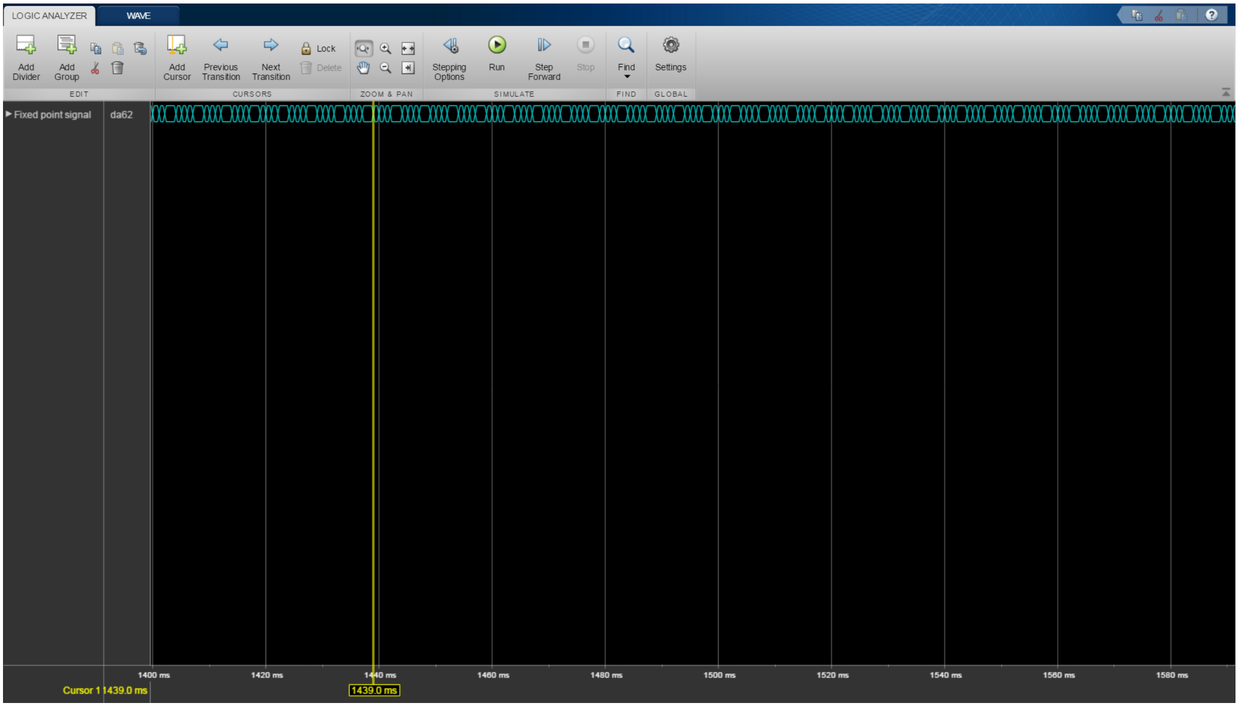


6 Run the `multipleModelInstances` model again. The **Logic Analyzer** displays `Model Instance 3` data.



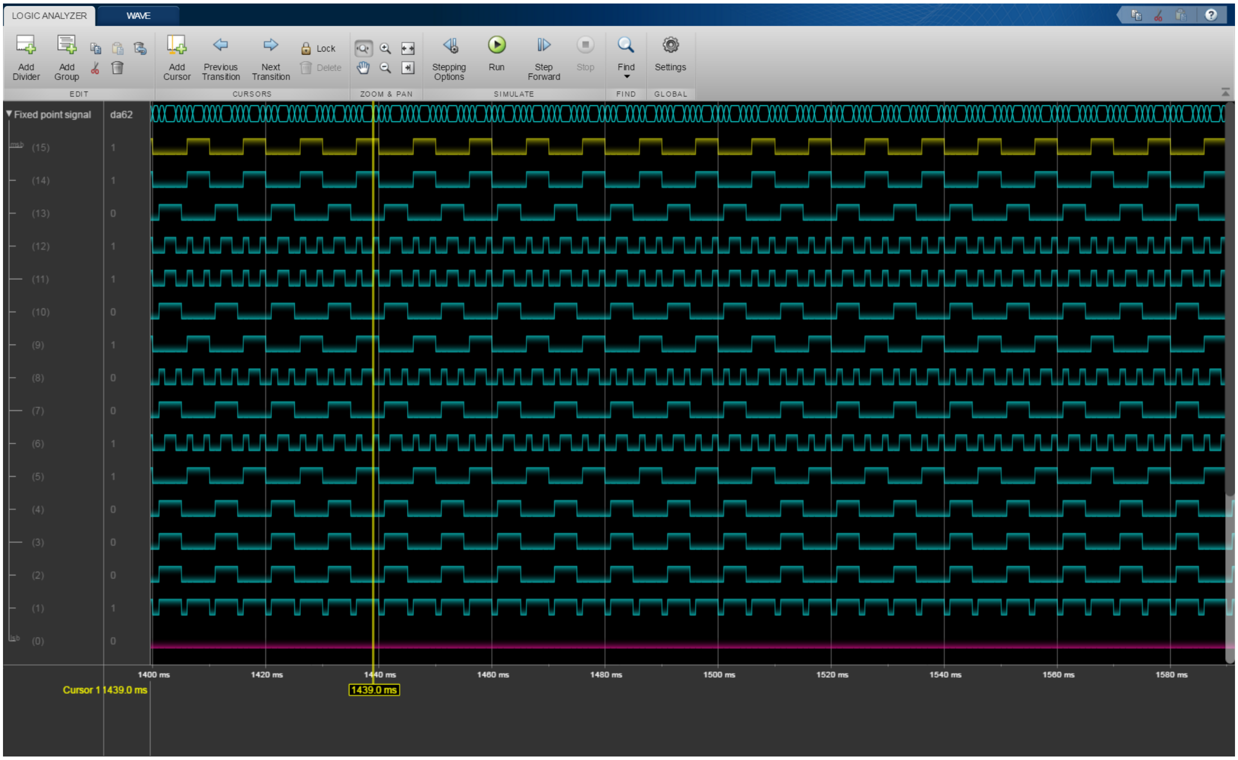
View Bit-Expanded Wave and Reverse Display Order of Bits

The **Logic Analyzer** enables you to bit-expand fixed-point and integer waves.

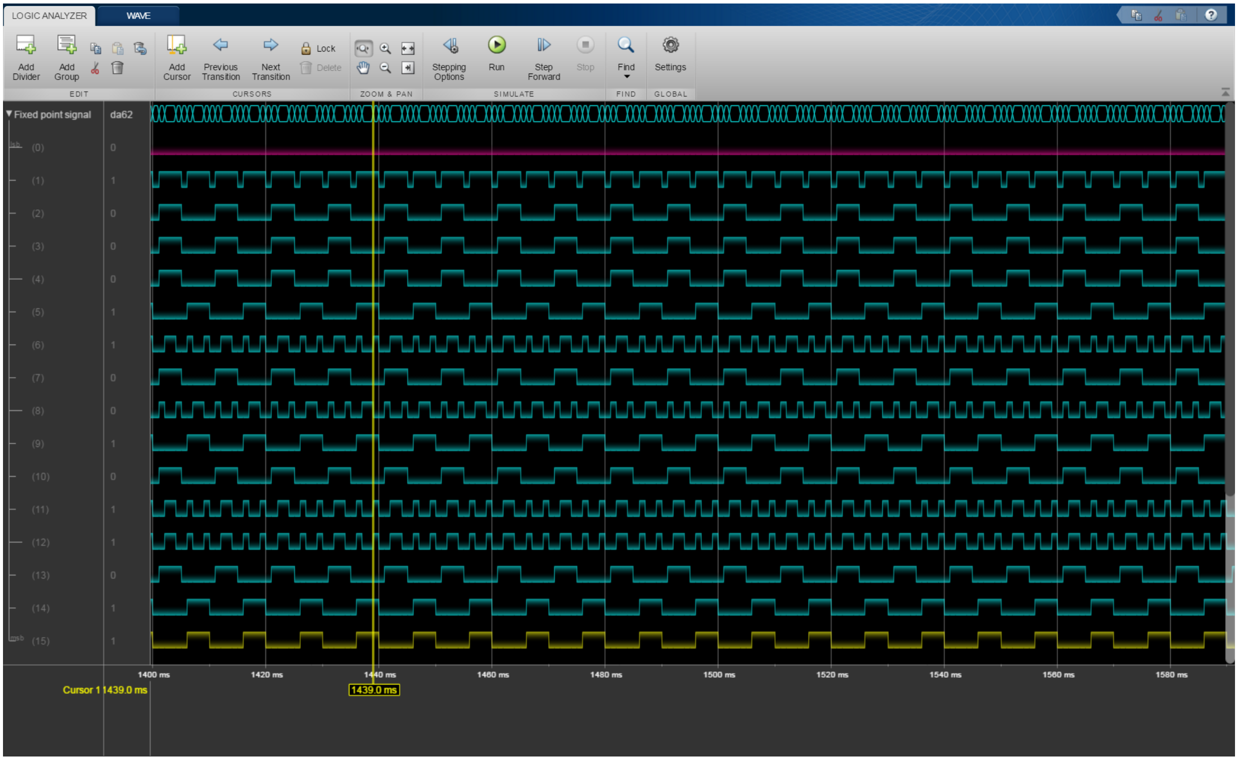


1. In the **Logic Analyzer**, click the arrow next to a fixed-point or integer wave to view the bits.

   The least significant bit and the most significant bit are marked with **lsb** and **msb** next to the wave names.



2 Click Settings, and then select **Display Least Significant bit first** to reverse the order of the displayed bits.



- “Inspect and Measure Transitions Using the Logic Analyzer”

Limitations

- You can use the **Logic Analyzer** in models running the following supported simulation modes.

Mode	Support	Notes and Limitations
Normal	Yes	
Accelerator	Yes	
Rapid Accelerator	Yes	Data is not available in the **Logic Analyzer** during simulation.

Mode	Support	Notes and Limitations
		If you simulate a model with the simulation mode set to rapid accelerator, then the following signals cannot be streamed into the **Logic Analyzer**: • Multi-instance model reference signals • Nonvirtual bus signals
Processor-in-the-loop (PIL)	No	
Software-in-the-loop (SIL)	No	
External	No	

For more information about these modes, see "How Acceleration Modes Work" (Simulink).

- The zoom and pan features of the **Logic Analyzer** are disabled while your simulation runs.
- You cannot use the **Logic Analyzer** to visualize signals in Model blocks with **Simulation mode** set to `Accelerator`.
- Signals marked for streaming for the **Logic Analyzer** must have fewer than 8000 samples per simulation step.
- Signals marked for streaming using `Simulink.sdi.markSignalForStreaming` or Dashboard Scope do not appear on the **Logic Analyzer**.

See Also

See Also

System Objects
dsp.LogicAnalyzer

Topics
"Inspect and Measure Transitions Using the Logic Analyzer"

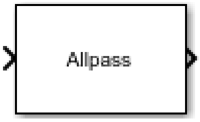
Introduced in R2016b

Blocks — Alphabetical List

Allpass Filter

Single-section or multiple-section allpass filter

Library: DSP System Toolbox / Filtering / Filter Implementations



Description

The Allpass Filter block filters each channel of the input signal independently using a single-section or multiple-section (cascaded) allpass filter. You can implement the allpass filter using a minimum multiplier, wave digital filter, or lattice structure.

In minimum multiplier form, the block uses the minimum number of required multipliers, n, with $2n$ delay units and $2n$ adders. In wave digital filter form, the block uses only n multipliers and n delay units, at the expense of $3n$ adders. The lattice structure uses 2n multipliers, n delay units, and $2n$ adders. For more details on these structures, see “Algorithms” on page 2-8.

Ports

Input

`Port_1` — Input data

column vector | row vector | matrix

Input data that is passed into the allpass filter. The block accepts real-valued or complex-valued multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$ and $n \geq 1$. The block also accepts variable-size inputs. That is, you can change the size of each input channel during simulation. However, the number of channels cannot change.

Data Types: `single` | `double`
Complex Number Support: Yes

Output

`Port_1` — Output of the allpass filter
column vector | row vector | matrix

The size of the filtered output matches the size of the input.

Data Types: `single` | `double`
Complex Number Support: Yes

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

`Internal allpass structure` — Filter structure
`Minimum multiplier` (default) | `Wave Digital Filter` | `Lattice`

- `Minimum multiplier` — This structure uses the minimum number of required multipliers, n, with $2n$ delay units and $2n$ adders. The coefficients to this structure are specified through the **Allpass polynomial coefficients** parameter.
- `Wave Digital Filter` — The structure uses n multipliers and n delay units, at the expense of $3n$ adders. The coefficients to this structure are specified through the **Wave Digital Filter allpass coefficients** parameter.
- `Lattice` — The structure uses $2n$ multipliers, n delay units, and $2n$ adders. The coefficients to this structure are specified through the **Lattice allpass coefficients** parameter.

For more details on these structures, see "Algorithms" on page 2-8.

`Allpass polynomial coefficients` — Coefficients in minimum multiplier form
`[-2^(-1/2), 1/2]` (default) | N-by-1 matrix | N-by-2 matrix

Specify the real allpass polynomial filter coefficients in minimum multiplier form as an N-by-1 matrix or an N-by-2 matrix.

- N-by-1 matrix — The block realizes N first-order allpass sections.
- N-by-2 matrix — The block realizes N second-order allpass sections.

The default value, `[ -2^(-1/2), 1/2 ]`, defines a stable second-order allpass filter with poles and zeros at $\pm\pi/3$ in the z-plane.

Tunable: Yes

Dependencies

To enable this parameter, set **Internal allpass structure** to `Minimum multiplier`.

`Wave Digital Filter allpass coefficients` — Coefficients in wave digital filter form

`[1/2, -2^(1/2)/3]` (default) | *N*-by-1 matrix | *N*-by-2 matrix

Specify the real allpass filter coefficients in wave digital filter form. The coefficients can be *N*-by-1 matrix for *N* first-order allpass sections and *N*-by-2 matrix for *N* second-order allpass sections. The default value, `[1/2, -2^(1/2)/3]`, is a transformed version of the default value of allpass polynomial coefficients. This value is computed using `allpass2wdf(Allpass polynomial coefficients)`. These coefficients define the same stable second-order allpass filter as when the allpass structure is set to `Minimum multiplier`.

Tunable: Yes

Dependencies

To enable this parameter, set **Internal allpass structure** to `Wave Digital Filter`.

`Indicate if last section is first order` — Is the last section first order

off (default) | on

- on — When you set select this check box, the last section is considered first order. Also, the second element of the last row of the *N*-by-2 matrix is ignored.
- off — When you do not select this check box, the last section is considered second-order.

Dependencies

To enable this parameter, set **Internal allpass structure** to `Minimum multiplier` or `Wave Digital Filter`.

`Lattice allpass coefficients` — **Coefficients in lattice form**
`[-2^(1/2)/3, 1/2]` (default) | *N*-by-1 column vector | 1-by-*N* row vector

Specify the real or complex allpass coefficients as lattice reflection coefficients. The default value, `[-2^(1/2)/3, 1/2]`, is a transformed and transposed version of the default value of the allpass polynomial coefficients. This value is computed using `transpose(tf2latc(1, [1 A]))`, where *A* is the value specified in **Allpass polynomial coefficients**.

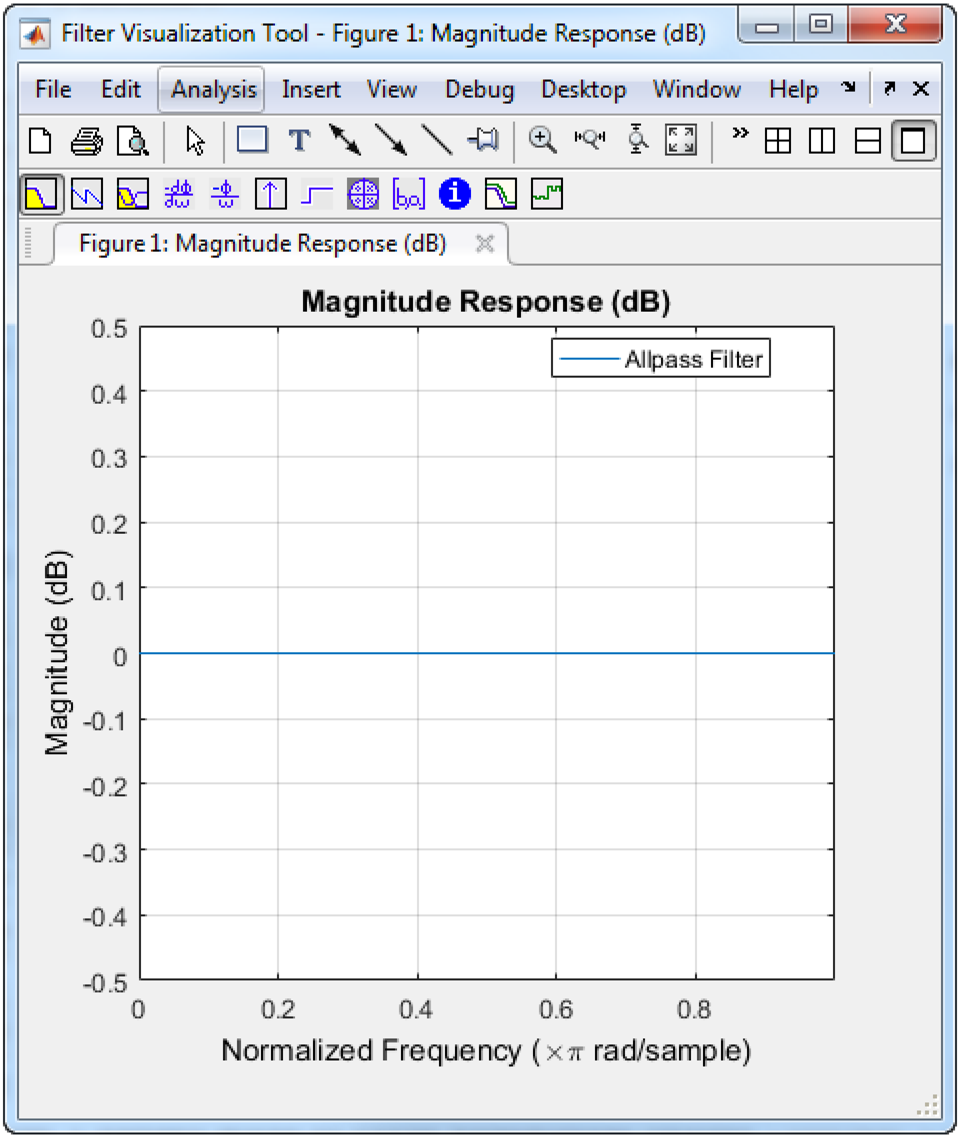
Tunable: Yes

Dependencies

To enable this parameter, set **Internal allpass structure** to `Lattice`.

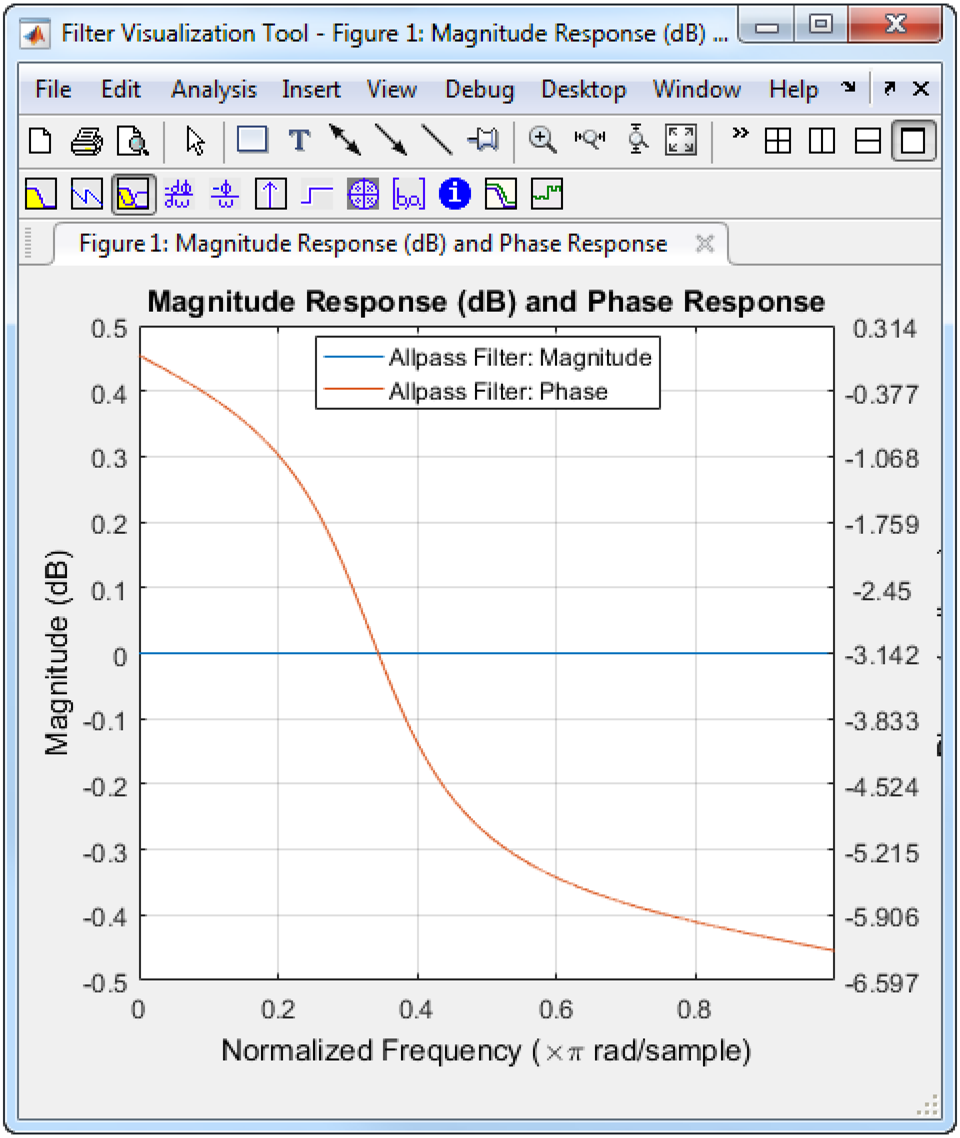
`View Filter Response` — **Visualize the filter response**
button

Opens the Filter Visualization Tool, `fvtool`, and displays the magnitude response of the allpass filter. The response is based on the parameters. Changes made to these parameters update `fvtool`.



To update the magnitude response while fvtool is running, modify the block parameters and click **Apply**.

To view the magnitude response and phase response simultaneously, click the 'Magnitude and Phase responses' button on the toolbar.



In this example, the magnitude response is flat and the phase response varies with frequencies. This varying phase response has applications in phase equalization, notch filtering, and multirate filtering. You can realize a lowpass filter using a parallel combination of two allpass filters that have 180 degrees of phase shift with respect to each other.

`Simulate using` — Type of simulation to run

`Code generation` (default) | `Interpreted execution`

- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Algorithms

The transfer function of an allpass filter is given by

$$H(z) = \frac{c(n) + c(n-1)z^{-1} + \ldots + z^{-n}}{1 + c(1)z^{-1} + \ldots + c(n)z^{-n}}.$$

c is allpass polynomial coefficients vector. The order, n, of the transfer function is the length of vector c.

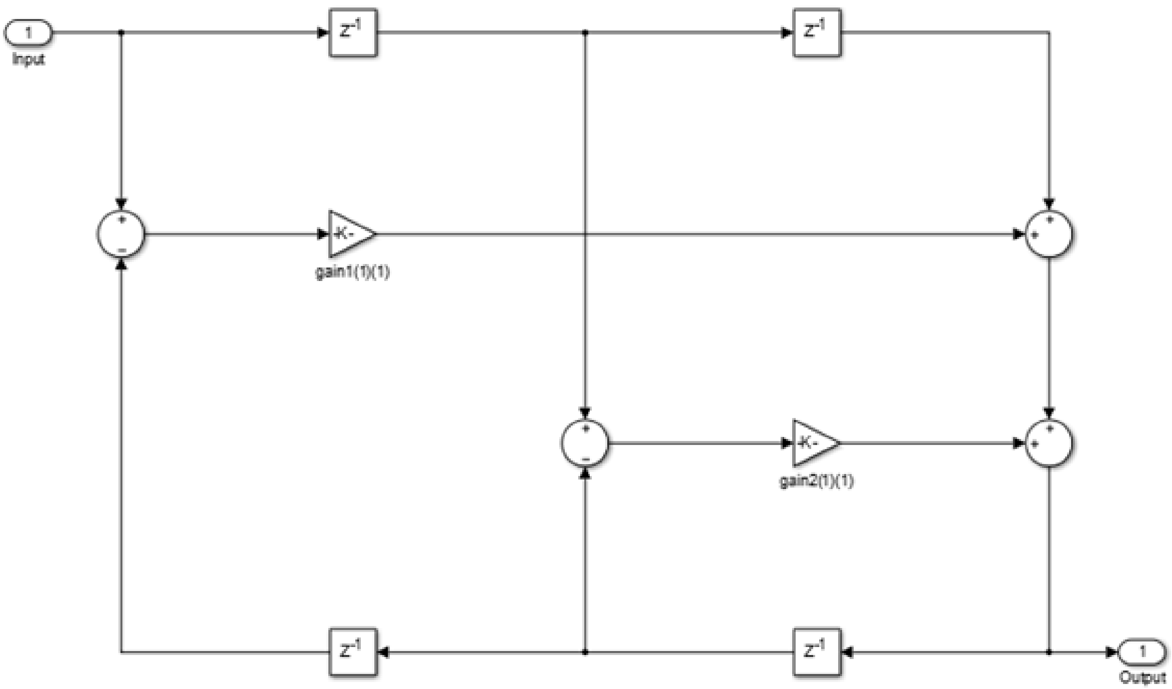
In the minimum multiplier form and wave digital form, the allpass filter is implemented as a cascade of either second-order (biquad) sections or first-order sections. When the coefficients are specified as an N-by-2 matrix, each row of the matrix specifies the coefficients of a second-order filter. The last element of the last row can be ignored based on the trailing first-order setting. When the coefficients are specified as an N-by-1 matrix, each element in the matrix specifies the coefficient of a first-order filter. The cascade of all the filter sections forms the allpass filter.

In the lattice form, the coefficients are specified as a vector.

These structures are computationally more economical and structurally more stable compared to the generic IIR filters, such as df1, df1t, df2, df2t. For all structures, the allpass filter can be a single-section or a multiple-section (cascaded) filter. The different sections can have different orders, but they are all implemented according to the same structure.

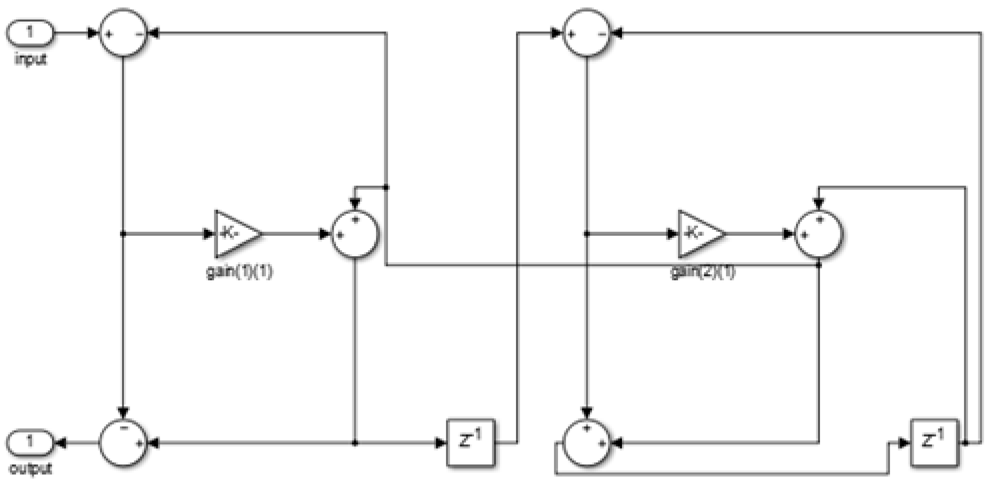
Minimum Multiplier

This structure realizes the allpass filter with the minimum number of required multipliers, equal to the order `n`. It also uses `2n` delay units and `2n` adders. The multipliers uses the specified coefficients, which are equal to the polynomial vector *c* in the allpass transfer function. In this second-order section of the minimum multiplier structure, the coefficients vector, *c*, is equal to [`0.1 -0.7`].



Wave Digital Filter

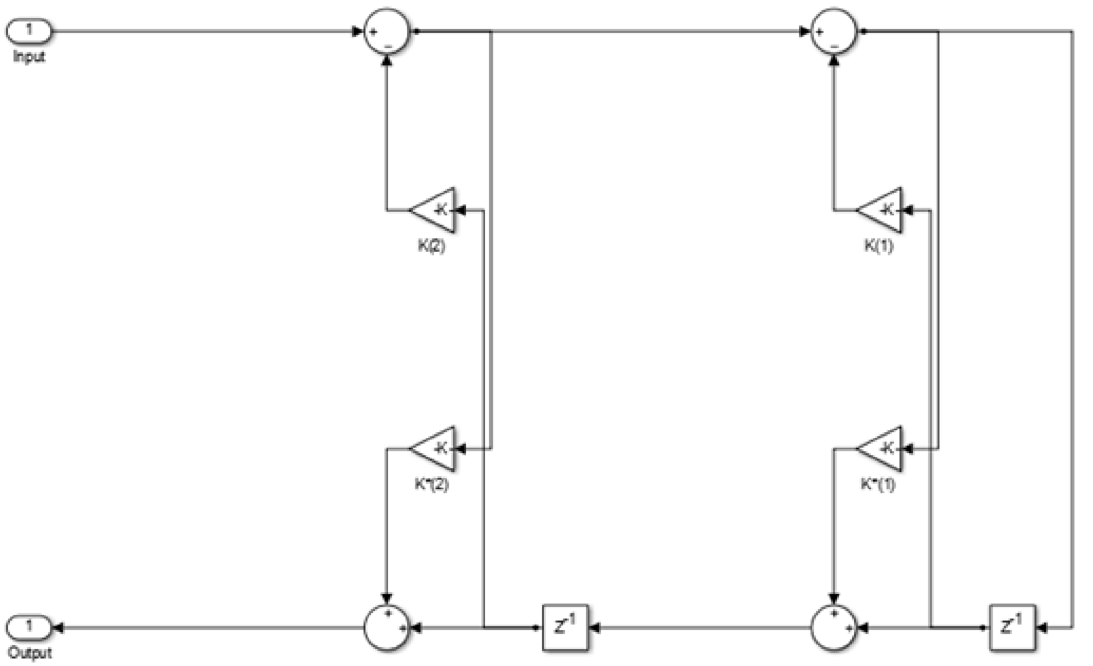
This structure uses n multipliers, but only n delay units, at the expense of requiring 3n adders. To use this structure, specify the coefficients in wave digital filter (WDF) form. Obtain the WDF equivalent of the conventional allpass coefficients using `allpass2wdf(allpass_coefficients)`. To convert WDF coefficients into the equivalent allpass polynomial form, use `wdf2allpass(WDF coefficients)`. In this second-order section of the WDF structure, the coefficients vector *w* is equal to `allpass2wdf([0.1 -0.7])`.



Lattice

This lattice structure uses 2n multipliers, n delay units, and 2n adders. To use this structure, specify the coefficients as a vector.

You can obtain the lattice equivalent of the conventional allpass coefficients using `transpose(tf2latc(1, [1 allpass_coefficients]))`. In the following second-order section of the lattice structure, the coefficients vector is computed using `transpose(tf2latc(1, [1 0.1 -0.7]))`. Use these coefficients for a filter that is functionally equivalent to the minimum multiplier structure with coefficients [0.1 -0.7].



References

[1] Regalia, Philip A., Mitra Sanjit K., and Vaidyanathan, P. P. (1988) "The Digital All-Pass Filter: A Versatile Signal Processing Building Block." *Proceedings of the IEEE*. Vol. 76, No. 1, 1988, pp. 19–37.

[2] Lutovac, M., D. Tosic, and B. Evans. *Filter Design for Signal Processing Using MATLAB and Mathematica.* Upper Saddle River, NJ: Prentice Hall, 2001.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

See Also

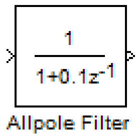
Blocks
`Biquad Filter` | `IIR Halfband Decimator` | `IIR Halfband Interpolator`

System Objects
dsp.AllpassFilter | dsp.BiquadFilter | dsp.CoupledAllpassFilter | dsp.IIRFilter | dsp.IIRHalfbandDecimator | dsp.IIRHalfbandInterpolator

Introduced in R2016b

Allpole Filter

Model allpole filters



Library

Filtering / Filter Implementations

dsparch4

Description

The Allpole Filter block independently filters each channel of the input signal with the specified allpole filter. The block can implement static filters with fixed coefficients, as well as time-varying filters with coefficients that change over time. You can tune the coefficients of a static filter during simulation.

This block filters each channel of the input signal independently over time. The **Input processing** parameter allows you to specify whether the block treats each element of the input as an independent channel (sample-based processing), or each column of the input as an independent channel (frame-based processing).

This block supports the Simulink state logging feature. See “States” (Simulink) in the *Simulink User's Guide* for more information.

Filter Structure Support

You can change the filter structure implemented with the Allpole Filter block by selecting one of the following from the **Filter structure** parameter:

- `Direct form`
- `Direct form transposed`
- `Lattice AR`

Specifying Initial States

The Allpole Filter block initializes the internal filter states to zero by default, which has the same effect as assuming that past inputs and outputs are zero. You can optionally use the **Initial states** parameter to specify nonzero initial conditions for the filter delays.

To determine the number of initial states you must specify and how to specify them, see the table on valid initial states. The **Initial states** parameter can take one of the forms described in the next table.

Valid Initial States

Initial Condition	Description
Scalar	The block initializes all delay elements in the filter to the scalar value.
Vector or matrix (for applying different delay elements to each channel)	Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel: • The vector length equals the product of the number of input channels and the number of delay elements in the filter, `#_of_filter_coeffs-1` (or `#_of_reflection_coeffs` for `Lattice AR`). • The matrix must have the same number of rows as the number of delay elements in the filter, `#_of_filter_coeffs-1` (`#_of_reflection_coeffs` for `Lattice AR`), and must have one column for each channel of the input signal.

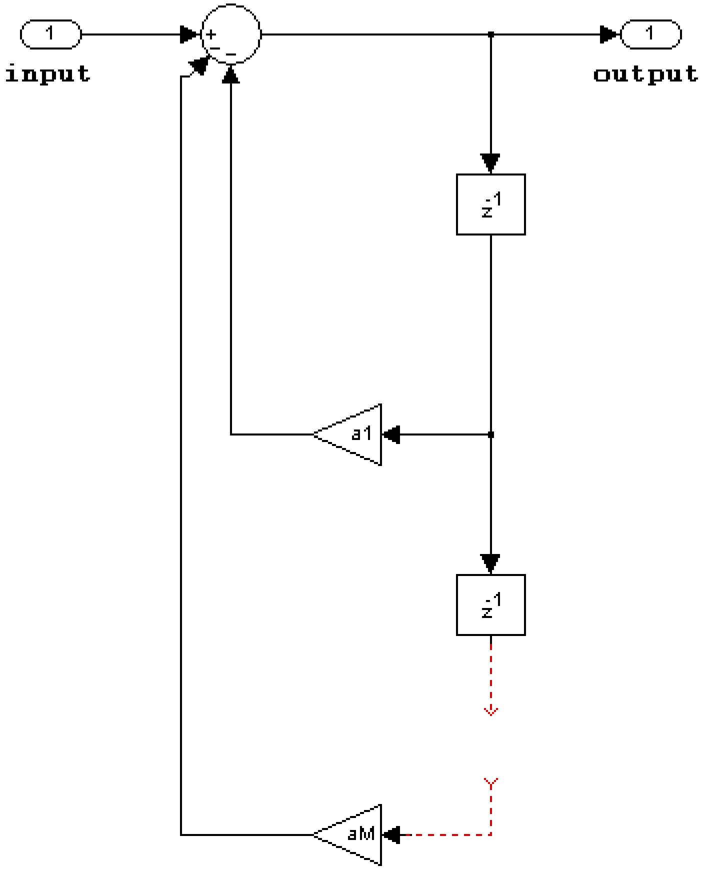
Data Type Support

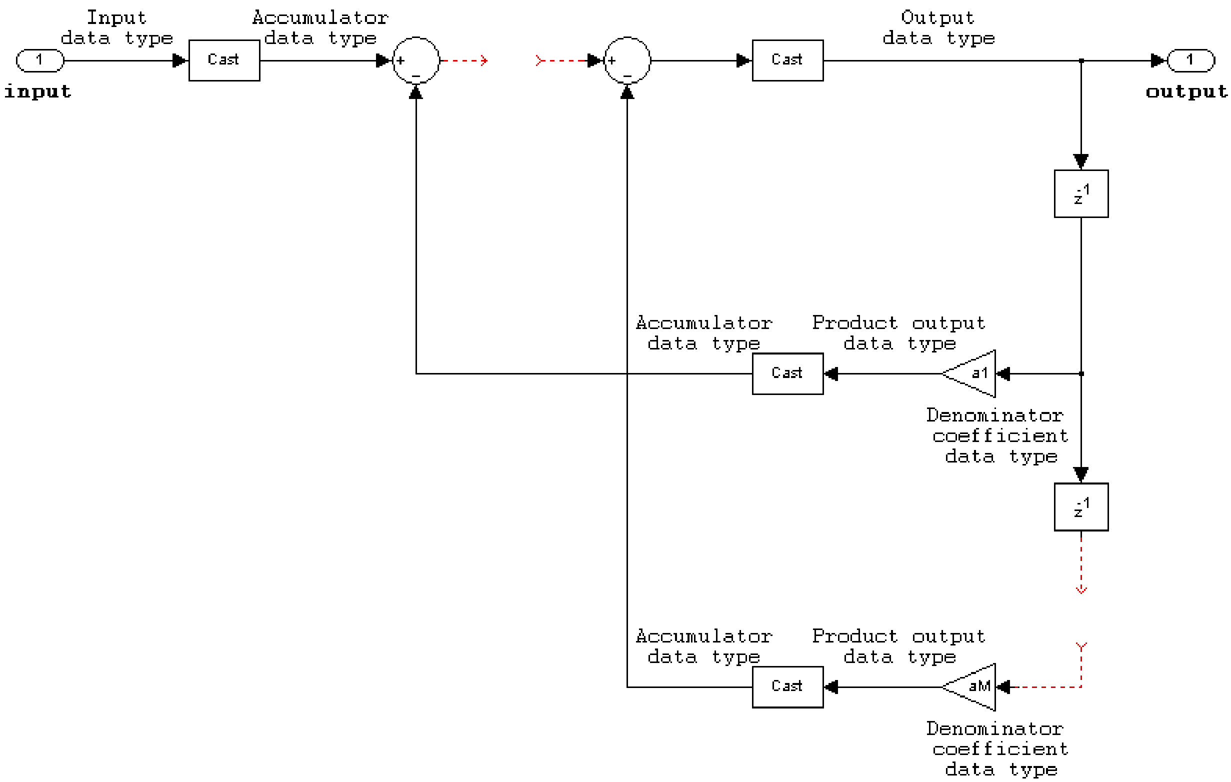
The Allpole Filter block accepts and outputs real and complex signals of any numeric data type supported by Simulink. The block supports the same types for the coefficients.

The following diagrams show the filter structure and the data types used within the Allpole Filter block for fixed-point signals.

Direct Form

You cannot specify the state data type on the block mask for this structure because the output states have the same data types as the output.

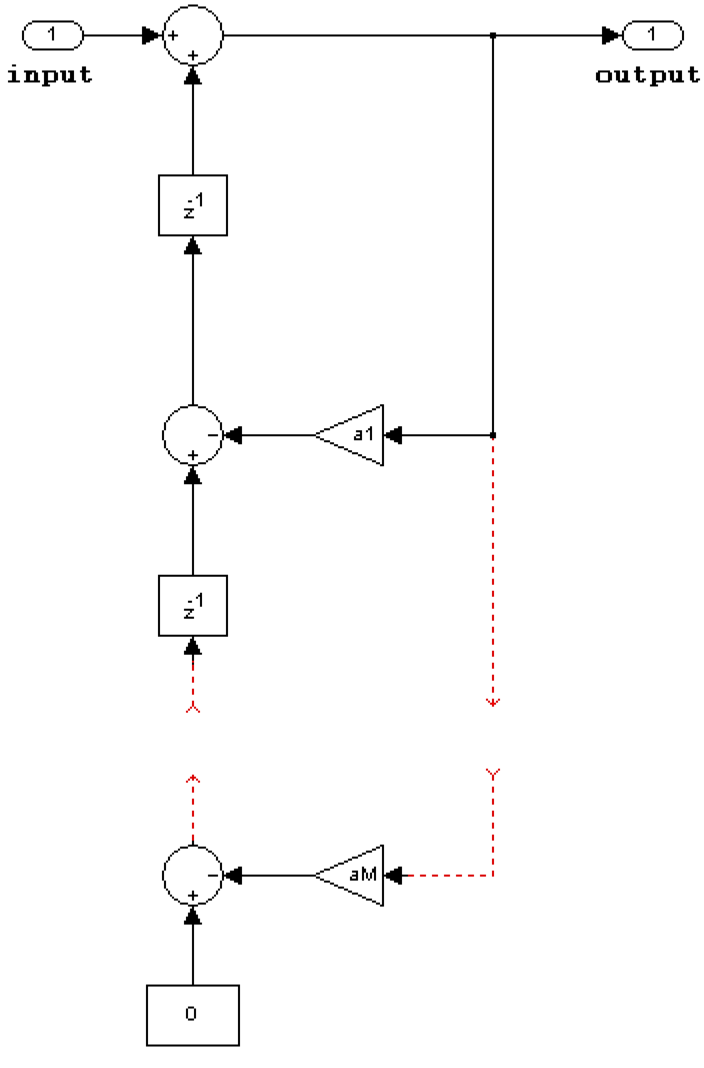

1
input
1
output
z
-1
a1
z
-1
aM

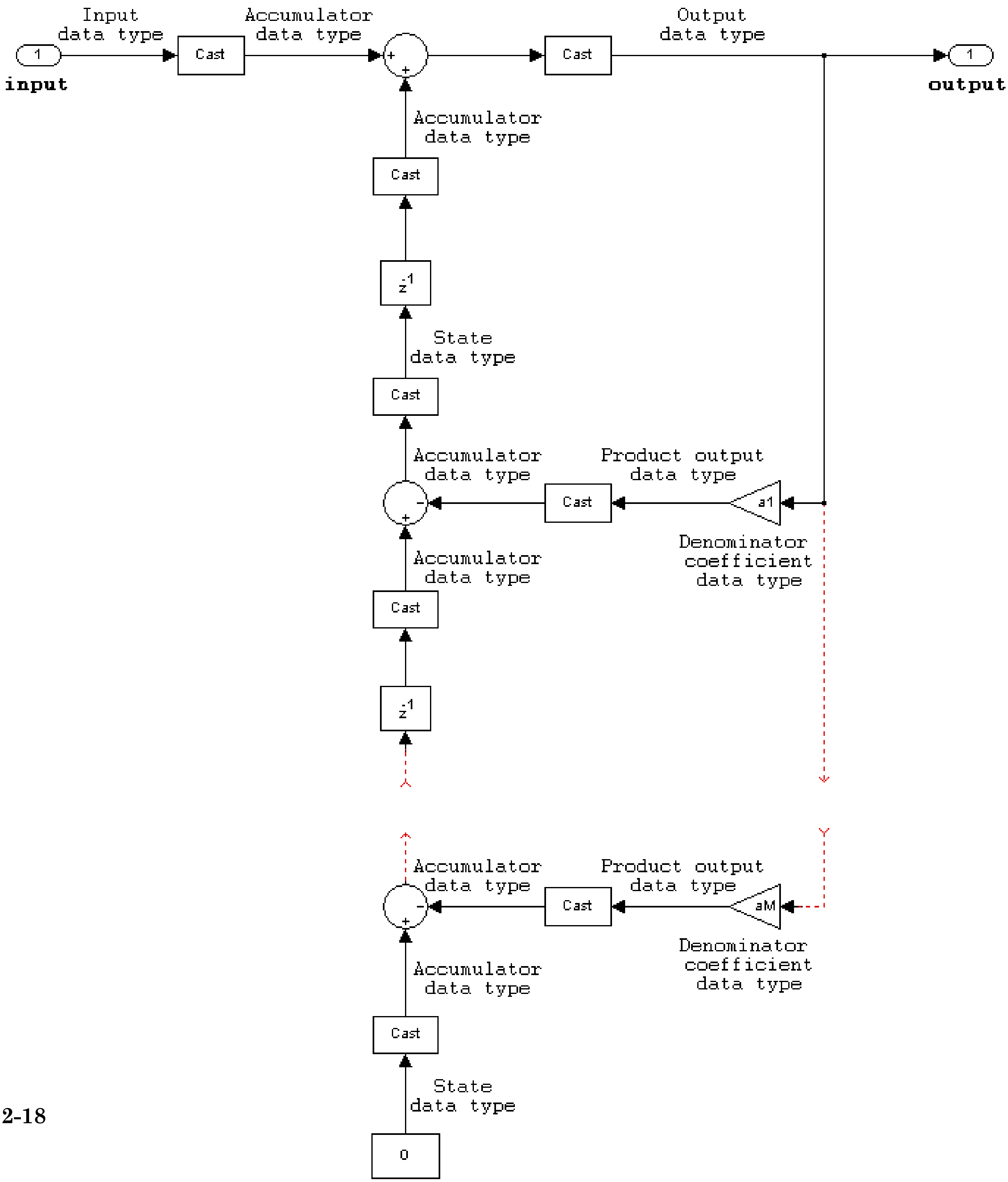


Input
data type
Accumulator
data type
Output
data type
1
input
Cast
Cast
1
output
z^-1
Accumulator
data type
Product output
data type
Cast
a1
Denominator
coefficient
data type
z^-1
Accumulator
data type
Product output
data type
Cast
aM
Denominator
coefficient
data type

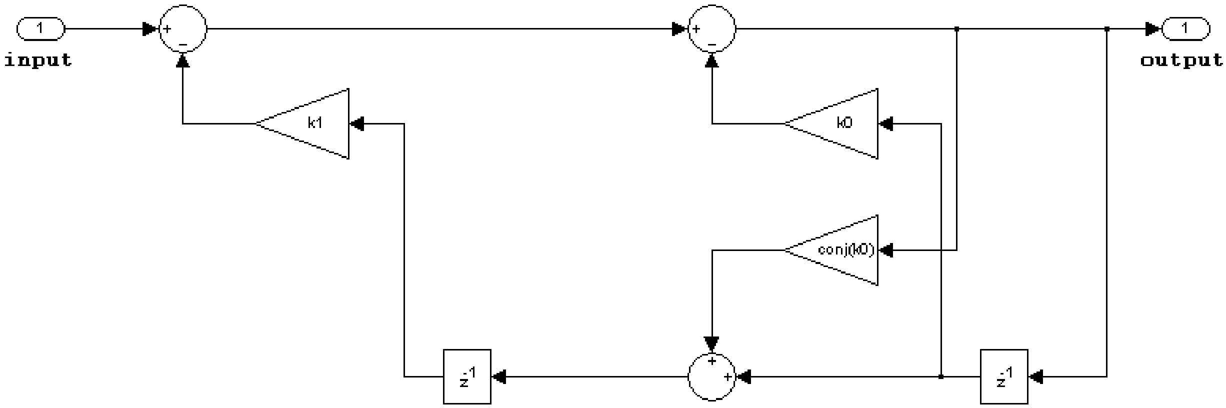
Direct Form Transposed

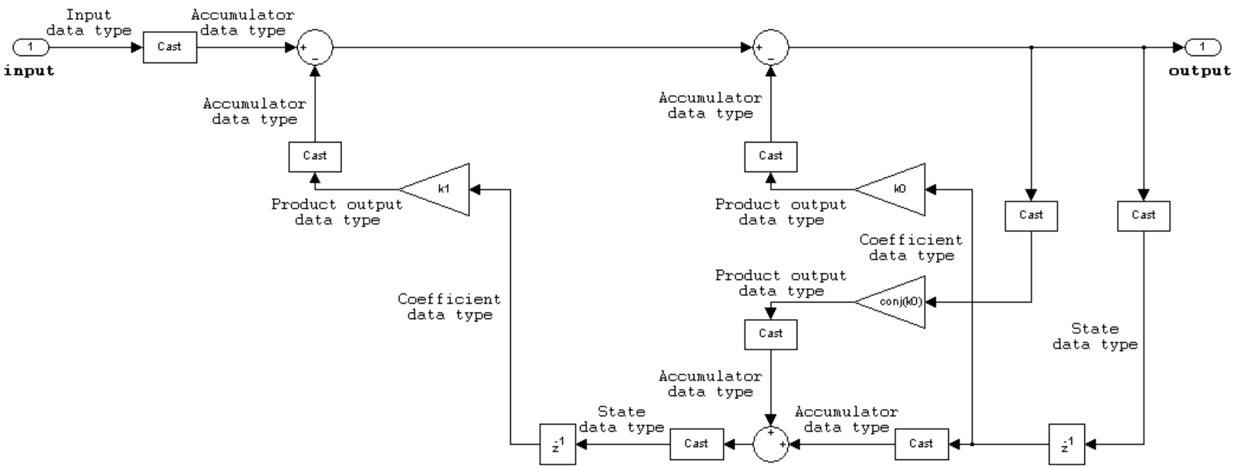
States are complex when either the inputs or the coefficients are complex.




Input
data type
Accumulator
data type
Output
data type
1
input
Cast
Cast
1
output
Accumulator
data type
Cast
z-1
State
data type
Cast
Accumulator
data type
Product output
data type
Cast
a1
Denominator
coefficient
data type
Accumulator
data type
Cast
z-1
Accumulator
data type
Product output
data type
Cast
aM
Denominator
coefficient
data type
Accumulator
data type
Cast
State
data type
0

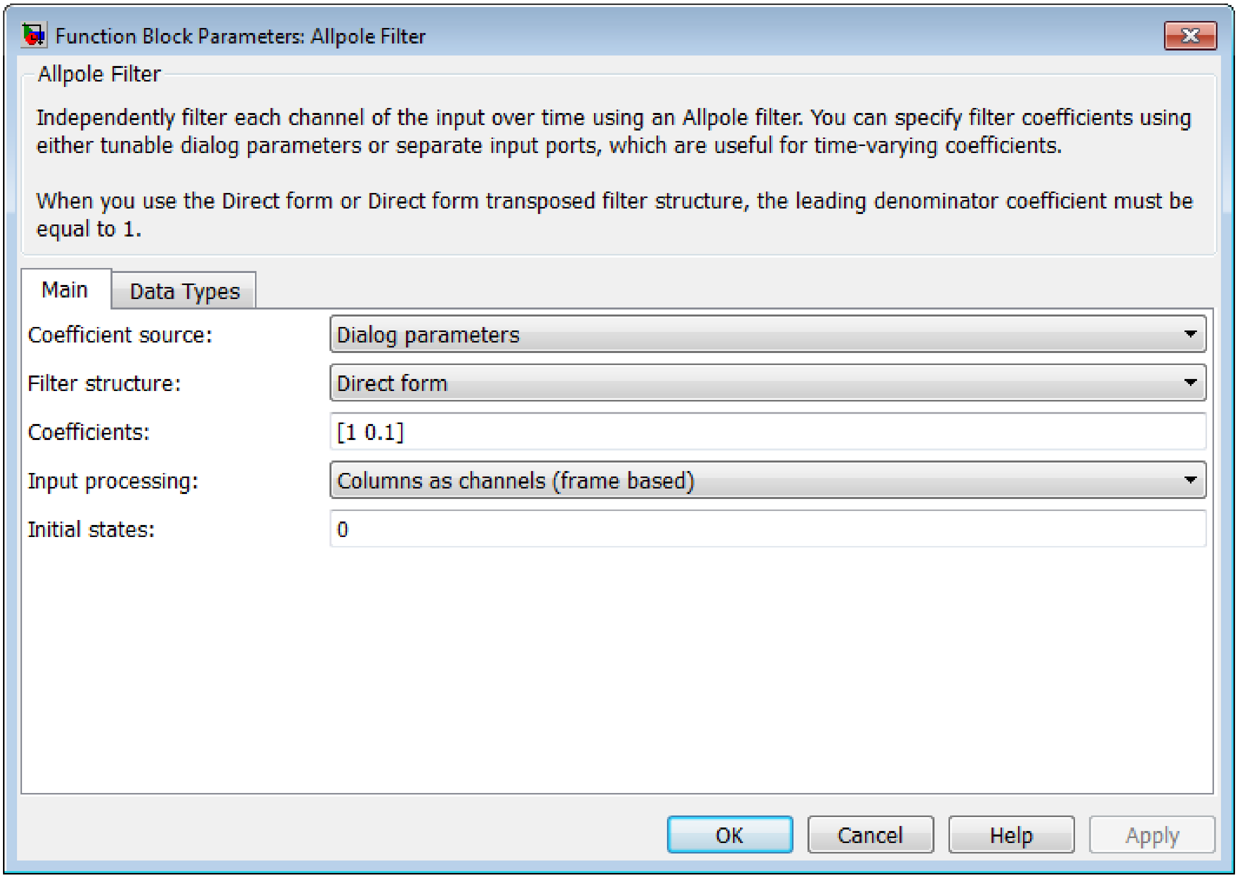
Lattice AR





Dialog Box

The **Main** pane of the Allpole Filter block dialog box appears as follows.



Coefficient source

Select whether you want to specify the filter coefficients on the block mask or through an input port.

Filter structure

Select the filter structure you want the block to implement. You can select `Direct form`, `Direct form transposed`, or `Lattice AR`.

Coefficients

Specify the row vector of coefficients of the filter's transfer function.

This parameter is visible only when you set the **Coefficient source** to `Dialog parameters`.

Input processing

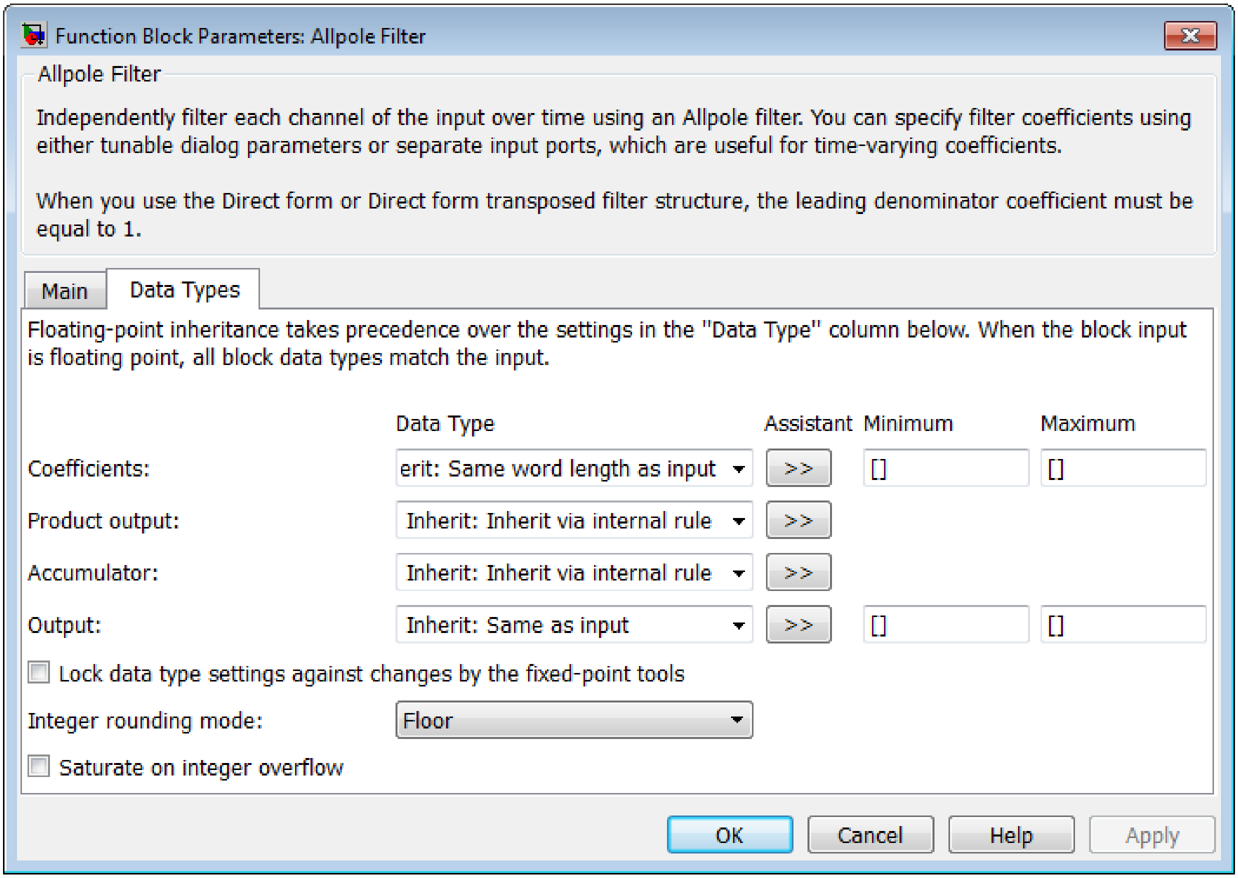
Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- `Elements as channels (sample based)` — Treat each element of the input as an independent channel (sample-based processing).
- `Columns as channels (frame based)` — Treat each column of the input as an independent channel (frame-based processing).

Initial states

Specify the initial conditions of the filter states. To learn how to specify initial states, see "Specifying Initial States" on page 2-14.

The **Data Types** pane of the Allpole Filter block dialog box appears as follows.



Coefficients

Specify the coefficient data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Coefficient** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) for more information.

Coefficients minimum

Specify the minimum value that a filter coefficient should have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters” (Simulink))
- Automatic scaling of fixed-point data types

Coefficients maximum

Specify the maximum value that a filter coefficient should have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters” (Simulink))
- Automatic scaling of fixed-point data types

Product output

Specify the product output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Product output** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) for more information.

Accumulator

Specify the accumulator data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- A built-in data type, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) for more information.

State

Specify the state data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

This parameter is only visible when the selected filter structure is `Lattice MA`.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **State** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) for more information.

Output

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- A built-in data type, for example, `int8`

- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See “Control Signal Data Types” (Simulink) in the Simulink User's Guide (Simulink) for more information.

Output minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Integer rounding mode

Specify the rounding mode for fixed-point operations.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow and you want explicit saturation protection in the generated code.	Overflows saturate to either the minimum or maximum value that the data type can represent.	An overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Check for Signal Range Errors" (Simulink).	Overflows wrap to the appropriate value that is representable by the data type.	The number 130 does not fit in a signed 8-bit integer and wraps to -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Signed fixed point
- 8-, 16-, and 32-bit signed integers

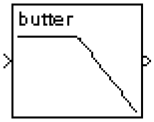
See Also

`Discrete FIR Filter`	DSP System Toolbox
`Filter Realization Wizard`	DSP System Toolbox
`filterDesigner`	DSP System Toolbox
`fvtool`	Signal Processing Toolbox

Introduced in R2011b

Analog Filter Design

Design and implement analog filters



Library

Filtering / Filter Implementations

`dsparch4`

Description

The Analog Filter Design block designs and implements a Butterworth, Chebyshev type I, Chebyshev type II, or elliptic filter in a highpass, lowpass, bandpass, or bandstop configuration.

The input must be a sample-based, continuous-time, real-valued, scalar signal.

The design and band configuration of the filter are selected from the **Design method** and **Filter type** pop-up menus in the dialog box. For each combination of design method and band configuration, an appropriate set of secondary parameters is displayed.

Filter Design	Description
`Butterworth`	The magnitude response of a Butterworth filter is maximally flat in the passband and monotonic overall.
`Chebyshev type I`	The magnitude response of a Chebyshev type I filter is equiripple in the passband and monotonic in the stopband.
`Chebyshev type II`	The magnitude response of a Chebyshev type II filter is monotonic in the passband and equiripple in the stopband.
`Elliptic`	The magnitude response of an elliptic filter is equiripple in both the passband and the stopband.

The following table lists the available parameters for each design/band combination. For lowpass and highpass band configurations, these parameters include the passband edge frequency Ω_p, the stopband edge frequency Ω_s, the passband ripple R_p, and the stopband attenuation R_s. For bandpass and bandstop configurations, the parameters include the lower and upper passband edge frequencies, Ω_{p1} and Ω_{p2}, the lower and upper stopband edge frequencies, Ω_{s1} and Ω_{s2}, the passband ripple R_p, and the stopband attenuation R_s. Frequency values are in rad/s, and ripple and attenuation values are in dB.

	Lowpass	Highpass	Bandpass	Bandstop
`Butterworth`	Order, Ω_p	Order, Ω_p	Order, Ω_{p1}, Ω_{p2}	Order, Ω_{p1}, Ω_{p2}
`Chebyshev Type I`	Order, Ω_p, R_p	Order, Ω_p, R_p	Order, Ω_{p1}, Ω_{p2}, $\mathrm{R_p}$	Order, Ω_{p1}, Ω_{p2}, R_p
`Chebyshev Type II`	Order, Ω_s, R_s	Order, Ω_s, R_s	Order, Ω_{s1}, Ω_{s2}, R_s	Order, Ω_{s1}, Ω_{s2}, R_s
`Elliptic`	Order, Ω_p, R_p, R_s	Order, Ω_p, R_p, R_s	Order, Ω_{p1}, Ω_{p2}, R_p, R_s	Order, Ω_{p1}, Ω_{p2}, R_p, R_s

The Analog Filter Design block uses a state-space filter representation, and applies the filter using the State-Space block in the Simulink Continuous library. All of the design methods use Signal Processing Toolbox™ functions to design the filter:

- The Butterworth design uses the toolbox function `butter`.
- The Chebyshev type I design uses the toolbox function `cheby1`.
- The Chebyshev type II design uses the toolbox function `cheby2`.
- The elliptic design uses the toolbox function `ellip`.

The Analog Filter Design block is built on the filter design capabilities of Signal Processing Toolbox software.

Note: The Analog Filter Design block does not work with the Simulink discrete solver, which is enabled when the **Solver** list is set to `Discrete (no continuous states)` in the **Solver** pane of the **Model Configuration Parameters** dialog box. Select one of the continuous solvers (such as `ode4`) instead.

Parameters

Design method

The filter design method: `Butterworth`, `Chebyshev type I`, `Chebyshev type II`, or `Elliptic`. Tunable (Simulink).

Filter type

The type of filter to design: `Lowpass`, `Highpass`, `Bandpass`, or `Bandstop`. Tunable (Simulink).

Filter order

The order of the filter, for lowpass and highpass configurations. For bandpass and bandstop configurations, the order of the final filter is *twice* this value.

Passband edge frequency

The passband edge frequency, in rad/s, for the highpass and lowpass configurations of the Butterworth, Chebyshev type I, and elliptic designs. Tunable (Simulink).

Lower passband edge frequency

The lower passband frequency, in rad/s, for the bandpass and bandstop configurations of the Butterworth, Chebyshev type I, and elliptic designs. Tunable (Simulink).

Upper passband edge frequency

The upper passband frequency, in rad/s, for the bandpass and bandstop configurations of the Butterworth, Chebyshev type I, or elliptic designs. Tunable (Simulink).

Stopband edge frequency

The stopband edge frequency, in rad/s, for the highpass and lowpass band configurations of the Chebyshev type II design. Tunable (Simulink).

Lower stopband edge frequency

The lower stopband edge frequency, in rad/s, for the bandpass and bandstop configurations of the Chebyshev type II design. Tunable (Simulink).

Upper stopband edge frequency

The upper stopband edge frequency, in rad/s, for the bandpass and bandstop filter configurations of the Chebyshev type II design. Tunable (Simulink).

Passband ripple in dB

The passband ripple, in dB, for the Chebyshev Type I and elliptic designs. Tunable (Simulink).

Stopband attenuation in dB

The stopband attenuation, in dB, for the Chebyshev Type II and elliptic designs. Tunable (Simulink).

References

Antoniou, A. *Digital Filters: Analysis, Design, and Applications*. 2nd ed. New York, NY: McGraw-Hill, 1993.

Supported Data Types

- Double-precision floating point

See Also

`Digital Filter Design`	DSP System Toolbox
`butter`	Signal Processing Toolbox
`cheby1`	Signal Processing Toolbox
`cheby2`	Signal Processing Toolbox
`ellip`	Signal Processing Toolbox

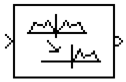
See the following sections for related information:

- "Filter Design"
- "Filter Analysis"

Introduced before R2006a

Analytic Signal

Compute analytic signals of discrete-time inputs



Library

Transforms

`dspxfrm3`

Description

The Analytic Signal block computes the complex analytic signal corresponding to each channel of the real M-by-N input, u

$$y = u + jH\{u\}$$

where $j = \sqrt{-1}$ and $H\{\}$ denotes the Hilbert transform. The real part of the output in each channel is a replica of the real input in that channel; the imaginary part is the Hilbert transform of the input. In the frequency domain, the Fourier transform of the analytic signal doubles the positive frequency content of the original signal while zeroing-out negative frequencies and retaining the DC component.

The block computes the Hilbert transform using an equiripple FIR filter with the order specified by the **Filter order** parameter, n. The linear phase filter is designed using the Remez exchange algorithm, and imposes a delay of $n/2$ on the input samples.

The output has the same dimensions as the input.

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block performs frame-based processing. In this mode, the block treats an

M-by-N matrix input as N independent channels containing M sequential time samples. The block computes the analytic signal for each channel over time.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block performs sample-based processing. In this mode, the block treats an M-by-N matrix input as $M*N$ independent channels and computes the analytic signal for each channel (matrix element) over time.

Parameters

Filter order

The length of the FIR filter used to compute the Hilbert transform.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

Introduced before R2006a

Arbitrary Response Filter

Design arbitrary response filter

Library

Filtering / Filter Designs

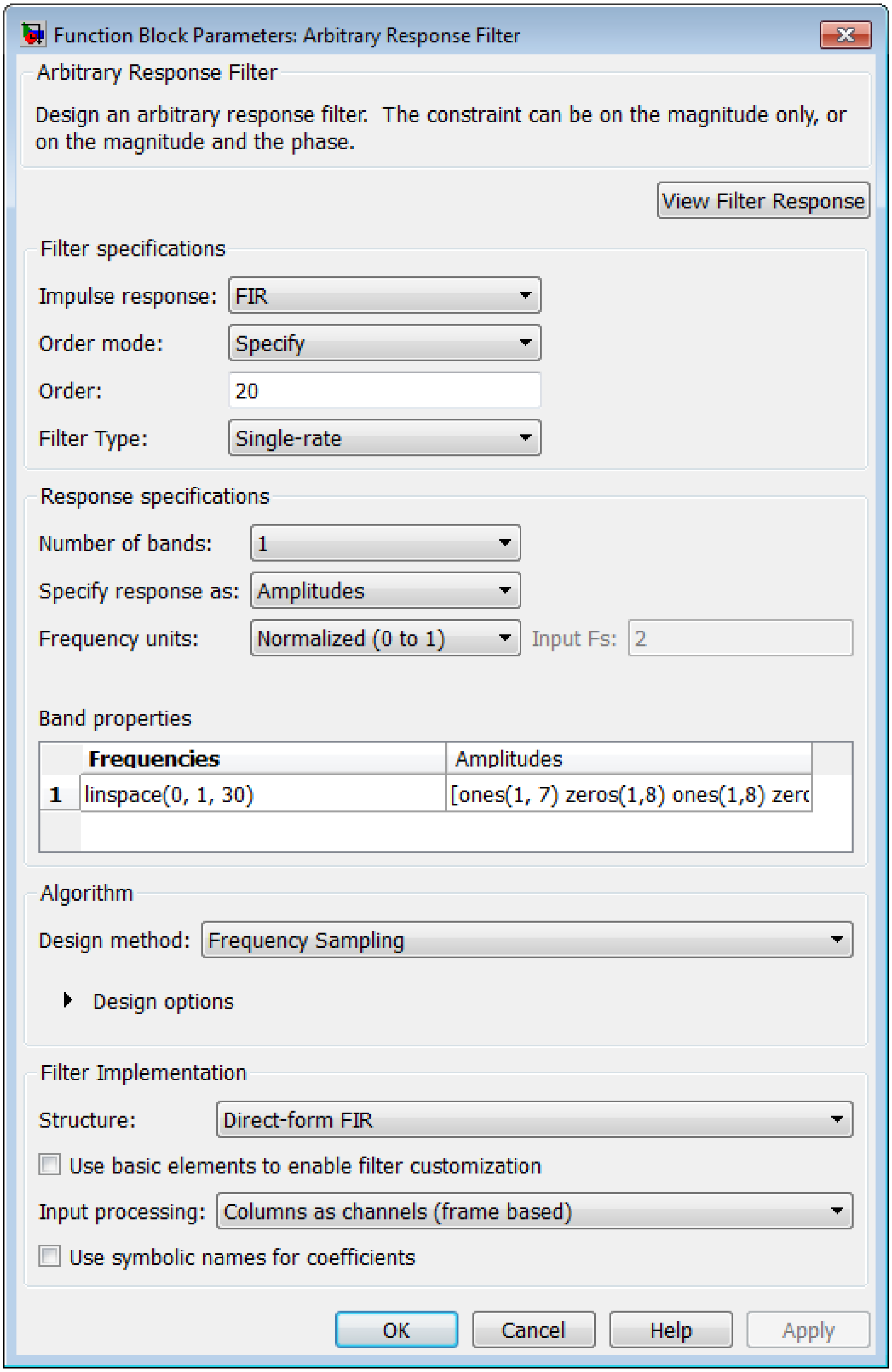
`dspfdesign`

Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See "Arbitrary Response Filter Design — Main Pane" on page 5-709 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox™ Filter Designs library.


Function Block Parameters: Arbitrary Response Filter
Arbitrary Response Filter
Design an arbitrary response filter. The constraint can be on the magnitude only, or on the magnitude and the phase.
View Filter Response
Filter specifications
Impulse response: FIR
Order mode: Specify
Order: 20
Filter Type: Single-rate
Response specifications
Number of bands: 1
Specify response as: Amplitudes
Frequency units: Normalized (0 to 1)
Input Fs: 2
Band properties
Frequencies
Amplitudes
1
linspace(0, 1, 30)
[ones(1, 7) zeros(1,8) ones(1,8) zerc
Algorithm
Design method: Frequency Sampling
Design options
Filter Implementation
Structure: Direct-form FIR
Use basic elements to enable filter customization
Input processing: Columns as channels (frame based)
Use symbolic names for coefficients
OK
Cancel
Help
Apply

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop down list, where `FIR` is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Order mode

Select `Minimum` or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option so you can enter the filter order. When you set the **Impulse response** to `IIR`, you can specify different numerator and denominator orders. To specify a different denominator order, you must select the **Denominator order** check box.

Order

Enter the order for FIR filter, or the order of the numerator for the IIR filter.

Denominator order

Select the check box and enter the denominator order. This option is enabled only if `IIR` is selected for **Impulse response**.

Filter type

This option is available for FIR filters only. Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Response Specification

Number of Bands

Select the number of bands in the filter. Multiband design is available for both FIR and IIR filters.

Specify response as

Specify the response as `Amplitudes`, `Magnitudes and phases`, `Frequency response`, or `Group delay`. `Group delay` is only available for IIR designs.

Frequency units

Specify frequency units as either `Normalized`, which means normalized by the input sampling frequency, or select from `Hz`, `kHz`, `MHz`, or `GHz`.

Input Fs

Enter the input sampling frequency in the units specified in the **Frequency units** drop-down list. When you select the frequency units, this option is available.

Band Properties

These properties are modified automatically depending on the response chosen in the **Specify response as** drop-down list. Two or three columns are presented for input. The first column is always Frequencies. The other columns are Amplitudes, Magnitudes, Phases, or Frequency Response. Enter the corresponding vectors of values for each column.

- **Frequencies** and **Amplitudes** — These columns are presented for input if the response chosen in the **Specify response as** drop-down list is `Amplitudes`.

- **Frequencies**, **Magnitudes**, and **Phases** — These columns are presented for input if the response chosen in the **Specify response as** drop-down list is `Magnitudes and phases`.
- **Frequencies** and **Frequency response** — These columns are presented for input if the response chosen in the **Specify response as** drop-down list is `Frequency response`.

Algorithm

Design Method

Select the design method for the filter. Different methods are enabled depending on the defining parameters entered in the previous sections.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications.

- **Window** — Replace the square brackets with the name of a `window` function or function handle. For example, `hamming` or `@hamming`. If the window function takes parameters other than the length, use a cell array. For example, `{'kaiser',3.5}` or `{@chebwin,60}`.
- **Density factor** — Valid when the **Design method** is `Equiripple`. Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

  Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade-off between the accurate approximation to the ideal filter and the time to design the filter.
- **Phase constraint** — Valid when the **Design method** is `Equiripple`, you have the DSP System Toolbox installed, and **Specify response as** is set to `Amplitudes`. Choose one of `Linear`, `Minimum`, or `Maximum`.
- **Weights** — Valid when the **Design method** is `Equiripple`. Uses the weights in **Weights** to weight the error for a single-band design. If you have multiple

frequency bands, the **Weights** design option changes to **B1 Weights**, **B2 Weights** to designate the separate bands.

Filter Implementation

Structure

Select the structure for the filter, available for the corresponding design method.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

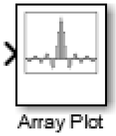
Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2009b

Array Plot

Display vectors or arrays



Library

Sinks

dspsnks4

Description

The Array Plot block plots vectors or arrays of data. It is a vector plot where data is uniformly spaced along the *x*-axis. You can specify the spacing to use with the `Sample increment` property.

The Array Plot block input signals must have the following characteristics:

- Discrete sample time
- Real- or complex values
- Floating- or fixed-point data type
- Fixed dimensions
- 2-D and non-scalar

Parameters

To change the signal display settings, select **View** > **Configuration Properties** to bring up the Configuration Properties: Array Plot dialog box. Then, modify the values for the **Sample increment**, **X-offset**, **X-axis sclae** and **Maximize axes** parameters on the **Main** tab. You can modify other display parameters on the **Display** tab.

Axes Scaling Properties: Array Plot Dialog Box

The Axes Scaling Properties dialog box allows you to set the following axes properties:

Properties

Axes Scaling

Specify when the Array Plot automatically scales the axes. You can select one of the following options:

- `Manual` — When you select this option, the Array Plot does not automatically scale the axes. You can manually scale the axes in any of the following ways:
  - Select **Tools** > **Axes Scaling Properties**.
  - Press one of the **Scale Axis Limits** toolbar buttons.
  - When the Array Plot figure is the active window, press **Ctrl-A**.
- `Auto` — When you select this option, the Array Plot scales the axes as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** check box.
- `After N Updates` — Selecting this option causes the Array Plot to scale the axes after a specified number of updates. This option is useful and more efficient when your display starts with one axis scale, but quickly reaches a different steady state axis scale. Selecting this option shows the **Number of updates** edit box.

By default, this property is set to `Auto`. This property is Tunable (Simulink).

Do not allow Y-axis limits to shrink

When you select this property, the *y*-axis is allowed only to grow during axes scaling operations. If you clear this check box, the *y*-axis or color limits may shrink during axes scaling operations.

This property appears only when you select `Auto` for the **Axis scaling** property. When you set the **Axes scaling** property to `Manual` or `After N Updates`, the *y*-axis or color limits are allowed to shrink. Tunable (Simulink).

Number of Updates

Specify as a positive integer the number of updates after which to scale the axes. This property appears only when you select `After N Updates` for the **Axes scaling** property. Tunable (Simulink).

Scale axes limits at stop

Select this check box to scale the axes when the simulation stops. The *y*-axis is always scaled. The *x*-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

Y-axis: Data Range

Set the percentage of the *y*-axis that the Array Plot uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the Scope scales the *y*-axis limits such that your data uses the entire *y*-axis range. If you then set this property to `30`, the scope increases the *y*-axis range such that your data uses only 30% of the *y*-axis range. Tunable (Simulink).

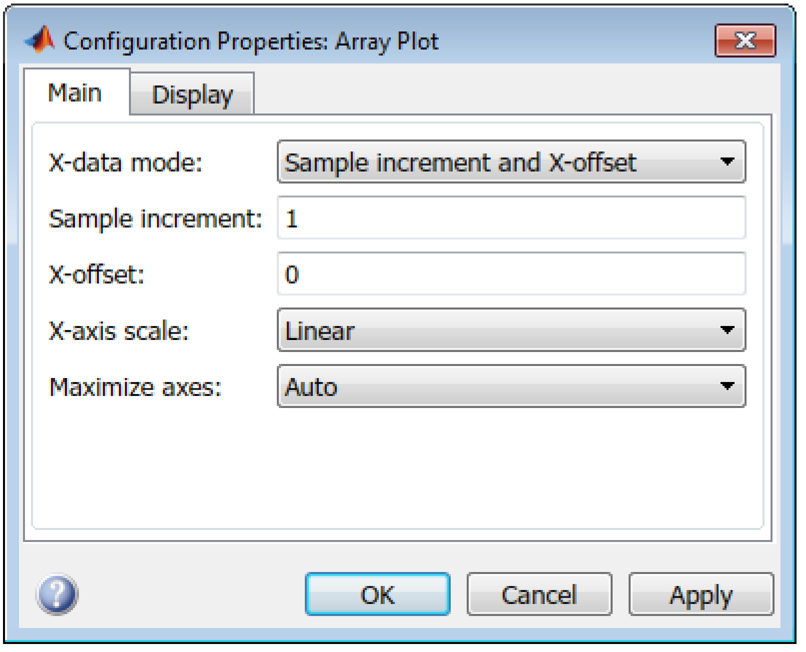
Y-axis: Align

Specify where the scope aligns your data along the *y*-axis when it scales the axes. You can select `Top`, `Center`, or `Bottom`. Tunable (Simulink).

Configuration Properties: Array Plot

The Configuration Properties dialog box controls various properties about the Time Scope displays. From the Time Scope menu, select **View > Configuration Properties** to open this dialog box. Alternatively, in the Array Plot toolbar, click the Configuration Properties  button.

Main Pane



X-data mode

Select the type of spacing to use between *x*-data values. `Sample increment and X-offset` uses the **Sample increment** and **X-offset** values. To specify your own custom spacing, select `'Custom'` and specify the data values in **CustomXData**.

Sample increment

Specify the spacing between samples along the *x*-axis as a finite numeric scalar. This property is displayed only if the **XDataMode** property is `Sample increment and X-offset`. The input signal is only *y*-axis data. *x*-axis data is set automatically based on both the **Sample increment** and **X-offset** values. For example, when **X-offset** is 0 and **Sample increment** is 1, the x-data for the input signal is set to 0, 1, 2, 3, 4, etc. If you set **Sample increment** to 0.25, the x-axis data becomes 0, 0.25, 0.5, 0.75, 1, etc.

X-offset

Specify the offset to apply to the *x*-axis, as a numeric scalar. This property is displayed only if the **XDataMode** property is `Custom`, This property is Tunable (Simulink).

Specify the offset to apply to the x-axis. This property is a numeric scalar. This property is Tunable (Simulink).

Custom X-data

Specify the x-data values as a vector of length equal to the frame length of the inputs. This property is displayed only if the **XDataMode** property is `'Custom'`. If you use the default (empty vector) value, the x-data is uniformly spaced and set to (0:L-1), where L is the frame length. An example of custom logarithmic x-data scaling is `[0:log10(44100/2):1024]`.

X-axis scale

Select `Linear` or `Log` as the x-axis scale. If **X-offset** is a negative value, you cannot set **X-axis cale** to `Log`. Tunable (Simulink)

Maximize axes

Specify whether to display the scope in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- `Auto` — In this mode, the axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- `On` — In this mode, the axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- `Off` — In this mode, none of the axes appear maximized.

This property is Tunable (Simulink).

Display Pane

Title

Specify the display title as a character vector. By default, the Array Plot has no title. Enter `%<SignalLabel>` to use the signal labels in the Simulink Model as the axes titles. Tunable (Simulink)

Show legend

Select this check box to show the legend in the display. The legend displays a name for each input signal. When the legend appears, you can place it anywhere inside of the scope window.

The legend lets you modify what signals are shown. To show only one signal, click the signal name. To toggle a signal on/off, right-click the signal name.

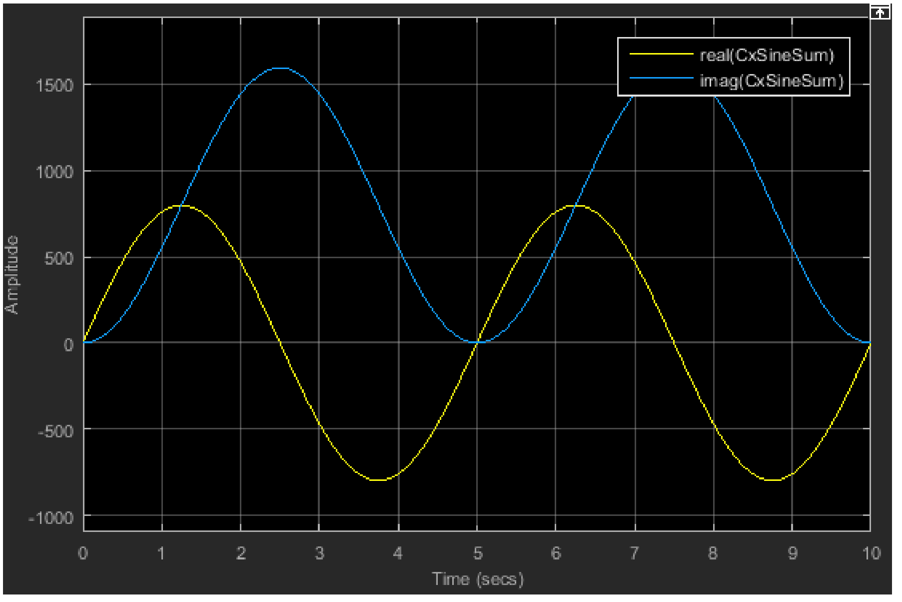
You can edit the name of any signal in the legend by double-clicking the current name, and entering a new name. By default, the scope sets each signal to its signal name or the name of the block from which it comes. Tunable (Simulink)

Show grid

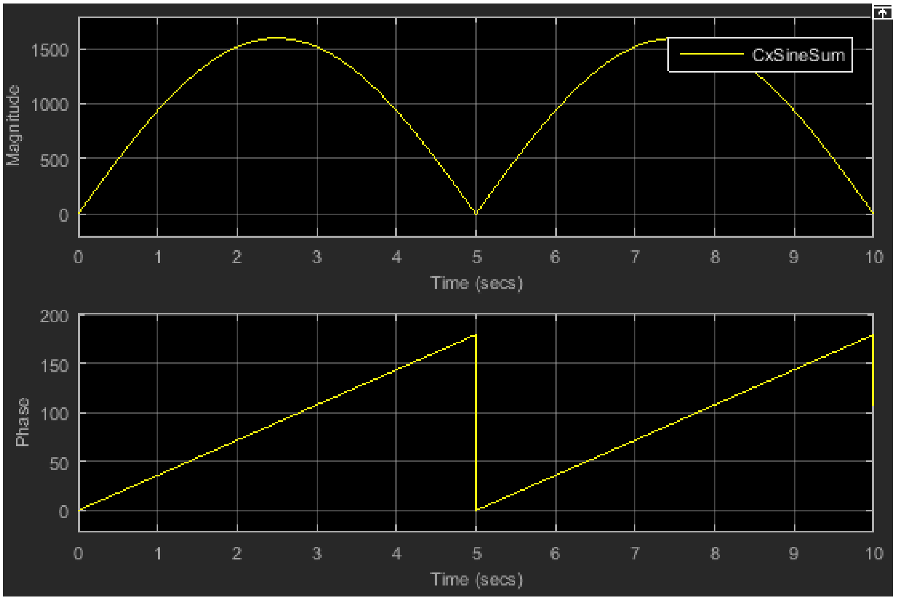
Select this check box to display a grid in the Array Plot. Tunable (Simulink)

Plot signals as magnitude and phase

When you select this check box, the scope splits the display into a magnitude plot and a phase plot. By default, this check box is cleared. If the input signal has complex values, the scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes, as shown in the following figure.



Selecting this check box and clicking the **Apply** or **OK** button changes the display. The magnitude of the input signal appears on the top axes and its phase, in degrees, appears on the bottom axes. See the following figure.



This feature is useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box returns the absolute value of the signal for the magnitude. The phase is 0 degrees for nonnegative input and 180 degrees for negative input. Tunable (Simulink)

X-label

Specify the text for the scope to display below the *x*-axis. This property is Tunable (Simulink).

Y-label

Specify the text for the scope to display to the left of the *y*-axis. Tunable (Simulink)

This property is not shown when you select the **Plot signals as magnitude and phase** check box. In that case, the *y*-axis label always appears as `Magnitude` on the top axis and `Phase` on the bottom axis.

Y-axis (Minimum)

Specify the minimum value of the *y*-axis. Tunable (Simulink)

When you select the **Plot signals as magnitude and phase** check box, the value of this property always applies to the magnitude plot on the top axis. The phase plot on the bottom axis is always limited to a minimum value of -180 degrees.

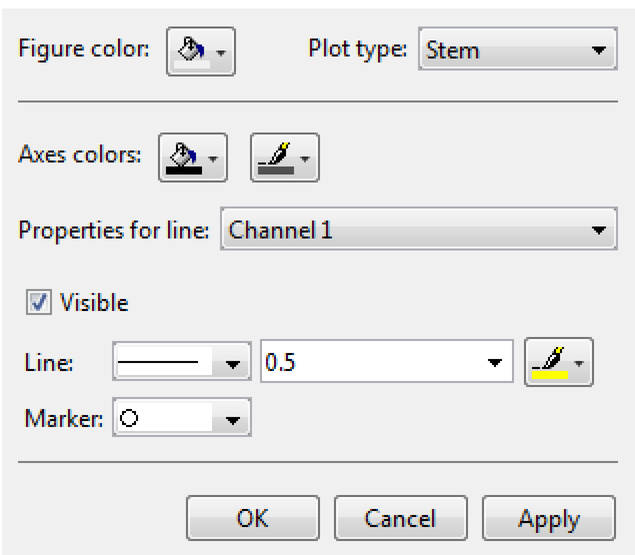
Y-axis (Maximum)

Specify the minimum value of the *y*-axis. Tunable (Simulink)

When you select the **Plot signals as magnitude and phase** check box, the value of this property always applies to the magnitude plot on the top axis. The phase plot on the bottom axis is always limited to a maximum value of 180 degrees.

Style: Array Plot

Select **View** > **Style** or the Style button () in the dropdown below the Configuration Properties button to open the Style dialog box. In this dialog box, you can change the figure colors, background axes colors, foreground axes colors, and properties of lines in a display.



Properties

The **Style** dialog box allows you to modify the following properties of the array plot:

Figure color

Specify the color that you want to apply to the background of the scope figure. By default, the figure color is black.

Plot type

Specify the type of plot to use for all the input signals displayed in the scope window.

- When you set this property to `'Stem'` , the scope displays the input signal as circles with vertical lines extending down to the *x*-axis at each of the sampled values . This approach is similar to the functionality of the MATLAB `stem` function.
- When you set this property to `'Line'` , the scope displays the input signal as lines connecting each of the sampled values. This approach is similar to the functionality of the MATLAB `line` or `plot` function.
- When you set this property to `'Stairs'`, the scope displays the input signal as a *stairstep* graph. A stairstep graph is made up of only horizontal lines and vertical lines. Each horizontal line represents the signal value for a discrete sample period and is connected to two vertical lines. Each vertical line represents a change in values occurring at a sample. This approach is equivalent to the MATLAB `stairs` function. Stairstep graphs are useful for drawing time history graphs of digitally sampled data.

This property is Tunable (Simulink).

Axes colors

Specify the colors that you want to apply to the background and line for the array plot.

Properties for line

Select the signal for which you want to modify the visibility, line properties, and marker properties.

Visible

Specify whether the selected signal is visible. If you clear this check box, the line disappears.

Line

Specify the line style, line width, and line color for the selected signal.

Marker

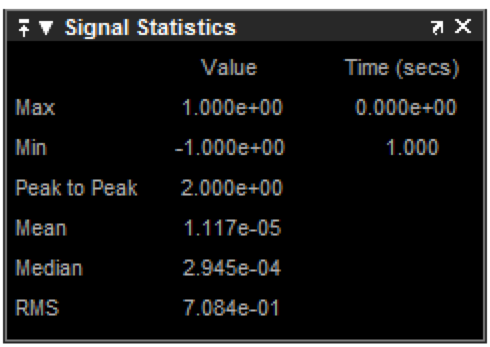
Specify marks for the selected signal to show at data points. This property is similar to the `Marker` property for plot objects. Choose any of the marker symbols from the dropdown list.

Measurements Panels

The Measurements panels are the panels that appear to the right side of the Time Scope GUI. Select the desired panels using **Tools** > **Measurements** in the scope toolbar or click the icons in the measurements dropdown.

Signal Statistics Panel

The **Signal Statistics** panel displays the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal. It also shows the *x*-axis indices at which the maximum and minimum values occur. In the Array Plot menu, select **Tools** > **Measurements** > **Signal Statistics**. Alternatively, in the scope toolbar, click the Signal Statistics  button.



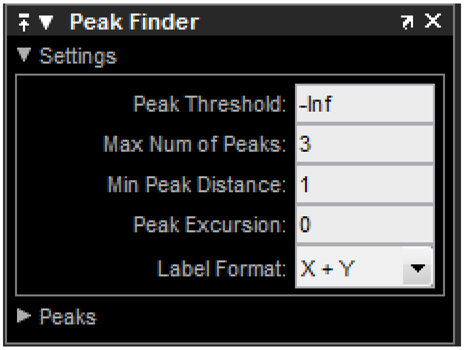
The statistics shown are:

- **Max** — The maximum or largest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `max` function reference.
- **Min** — The minimum or smallest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `min` function reference.
- **Peak to Peak** — The difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `peak2peak` function reference.
- **Mean** —The average or mean of all the values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `mean` function reference.
- **Median** — The median value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `median` function reference.
- **RMS** — Shows the difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `rms` function reference.

When you use the zoom options in the Array Plot, the Signal Statistics measurements automatically adjust to the time range shown in the display. For example, you can zoom in on one pulse to make the **Signal Statistics** panel display information about only that particular pulse.

Peak Finder Panel

The **Peak Finder** panel displays the maxima, showing the *x*-axis values at which they occur. Peaks are defined as a local maximum where lower values are present on both sides of a peak. Endpoints are not considered to be peaks. This panel allows you to modify the settings for peak threshold, maximum number of peaks, and peak excursion. In the Array Plot menu, select **Tools** > **Measurements** > **Peak Finder**. Alternatively, in the toolbar, select the Peak Finder  button.



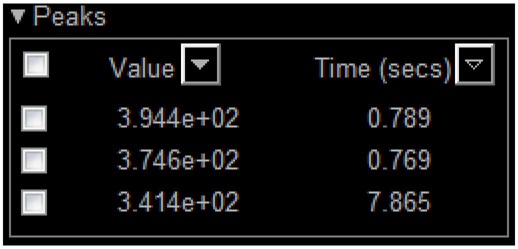
Settings

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input signal. For more information on the algorithms this pane uses, see the Signal Processing Toolbox `findpeaks` function reference.

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the `findpeaks` function `MINPEAKHEIGHT` parameter.
- **Max Num of Peaks** — The maximum number of peaks to show. The value you enter must be a scalar integer from 1 through 99. This setting is equivalent to the `findpeaks` function `NPEAKS` parameter.
- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the `findpeaks` function `MINPEAKDISTANCE` parameter.
- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples The peak excursion setting is equivalent to the `findpeaks` function`THRESHOLD` parameter.
- **Label Format** — The coordinates to display next to the calculated peak values on the plot.
  - `X+Y` — Display both *x*-axis and *y*-axis values
  - `X` — Display only *x*-axis values
  - `Y` — Display only *y*-axis values

Peaks

The **Peaks** pane displays all of the largest calculated peak values. It also shows the coordinates at which the peaks occur, using the parameters you define in the **Settings** pane. The value you specify in **Max Num of Peaks** determines the number of peaks shown in the list.



The numerical values displayed in the **Value** column are equivalent to the `findpeaks` function `pks` output argument. The numerical values displayed in the second column are similar to the `findpeaks` function `locs` output argument.

By default, the **Peaks** pane displays the largest calculated peak values in decreasing order of peak height. Use the sort descending ( ) or ascending ( ) button to rearrange the order in which the peak values display.

Use the check boxes to control which peak values are displayed. By default, all check boxes are cleared and the **Peak Finder** panel hides all the peak values.

Cursor Measurements Panel

The **Cursor Measurements** panel displays screen cursors. The panel provides two types of cursors for measuring signals. Waveform cursors are vertical cursors that track along the signal. Screen cursors are both horizontal and vertical cursors that you can place anywhere in the display.

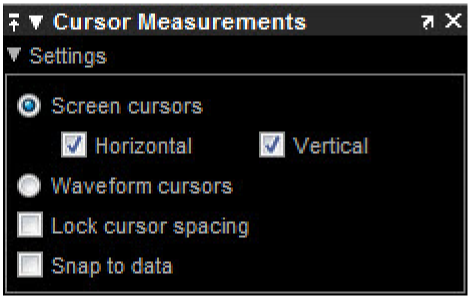
Note: If a data point in your signal has more than one value, the cursor measurement at that point is undefined and no cursor value is displayed.

In the Scope menu, select **Tools** > **Measurements** > **Cursor Measurements**.

Alternatively, in the Scope toolbar, click the Cursor Measurements button.

Settings

The **Settings** pane enables you to modify the type of screen cursors used for calculating measurements. When more than one signal is displayed, you can assign cursors to each trace individually.

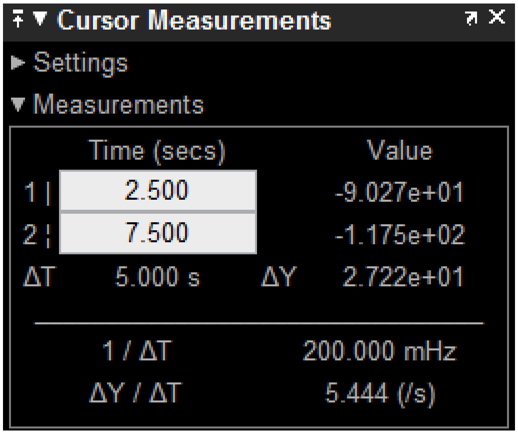


- **Screen Cursors** — Shows screen cursors
- **Horizontal** — Shows horizontal screen cursors
- **Vertical** — Shows vertical screen cursors
- **Waveform Cursors** — Shows cursors that attach to the input signals
- **Lock Cursor Spacing** — Locks the frequency difference between the two cursors
- **Snap to Data** — Positions the cursors on signal data points

You can use the mouse or the left and right arrow keys to move vertical or waveform cursors and the up and down arrow keys for horizontal cursors.

Measurements

The **Measurements** pane shows the time and value measurements.



- **1 |**— Shows or enables you to modify the time or value at cursor number one, or both.
- **2 :**— Shows or enables you to modify the time or value at cursor number two, or both.
- **Δt**— Shows the absolute value of the difference in the times between cursor number one and cursor number two.
- **ΔV**— Shows the absolute value of the difference in signal amplitudes between cursor number one and cursor number two.
- **1/Δt**— Shows the rate, the reciprocal of the absolute value of the difference in the times between cursor number one and cursor number two.
- **ΔV/Δt**— Shows the scope, the ratio of the absolute value of the difference in signal amplitudes between cursors to the absolute value of the difference in the times between cursors.

See Also

See Also

dsp.ArrayPlot | `Matrix Viewer` | `Spectrum Analyzer` | `Time Scope`

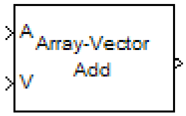
Topics

"Streaming Power Spectrum Estimation Using Welch's Method"

Introduced in R2015b

Array-Vector Add

Add vector to array along specified dimension



Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

Description

The Array-Vector Add block adds the values in the specified dimension of the N-dimensional input array *A* to the values in the input vector *V*.

The length of the input *V* must be the same as the length of the specified dimension of *A*. The Array-Vector Add block adds each element of *V* to the corresponding element along that dimension of *A*.

Consider a 3-dimensional *M*-by-*N*-by-*P* input array *A(i,j,k)* and a *N*-by-1 input vector *V*. When the **Add along dimension** parameter is set to 2, the output of the block *Y(i,j,k)* is

$$Y(i,j,k) = A(i,j,k) + V(j)$$

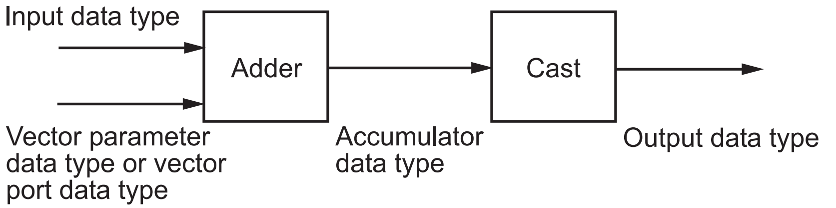
where

$$1 \le i \le M$$
$$1 \le j \le N$$
$$1 \le k \le P$$

The output of the Array-Vector Add block is the same size as the input array, *A*. This block accepts real and complex floating-point and fixed-point inputs.

Fixed-Point Data Types

The following diagram shows the data types used within the Array-Vector Add block for fixed-point signals.

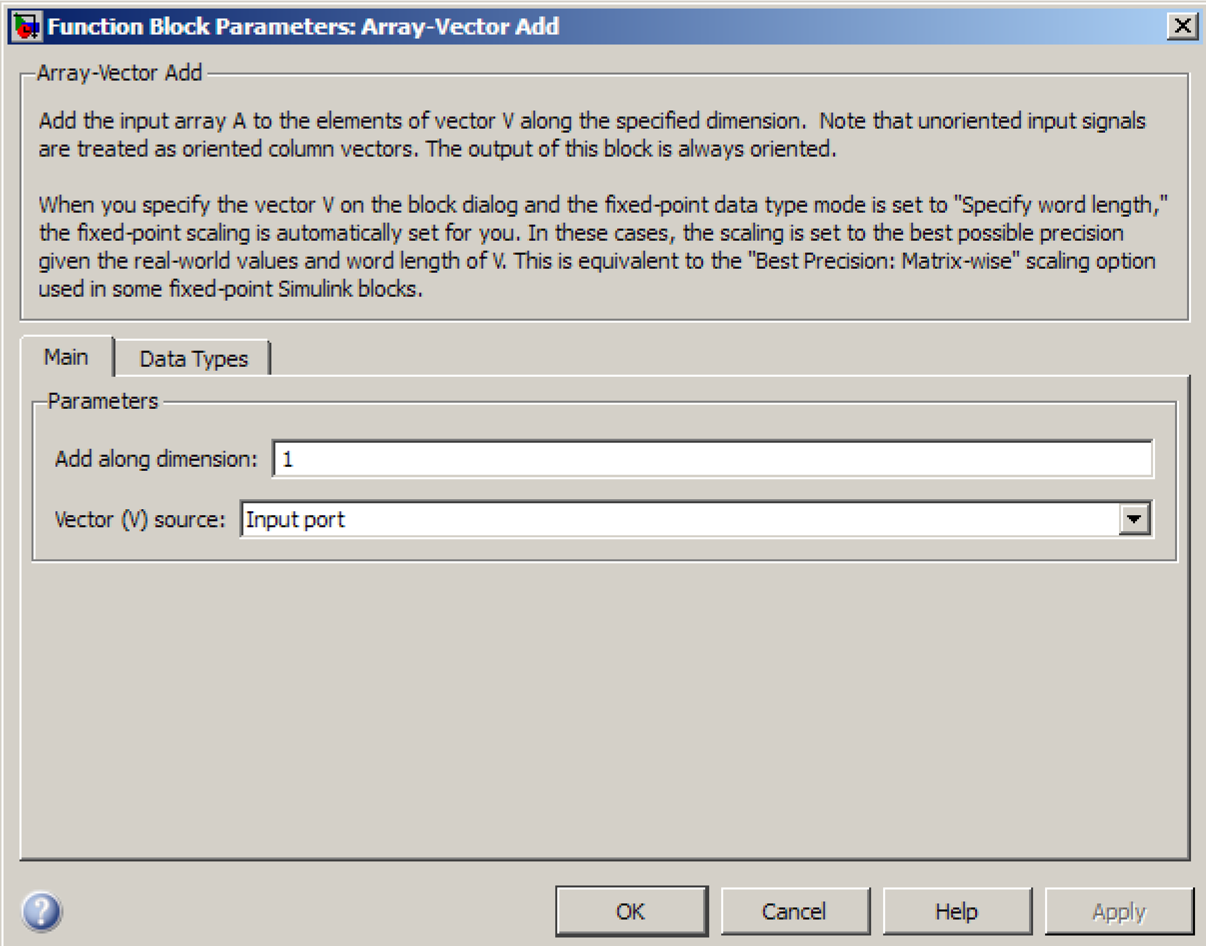


When you specify the vector *V* on the **Main** pane of the block mask, you must specify the data type and scaling properties of its elements in the **Vector (V)** parameter on the **Data Types** tab. When the vector comes in through the block port, its elements inherit their data type and scaling from the driving block.

You can set the vector, accumulator, and output data types in the block dialog as discussed below.

Dialog Box

The **Main** pane of the Array-Vector Add block dialog appears as follows.



Add along dimension

Specify the dimension along which to add the input array A to the elements of vector V.

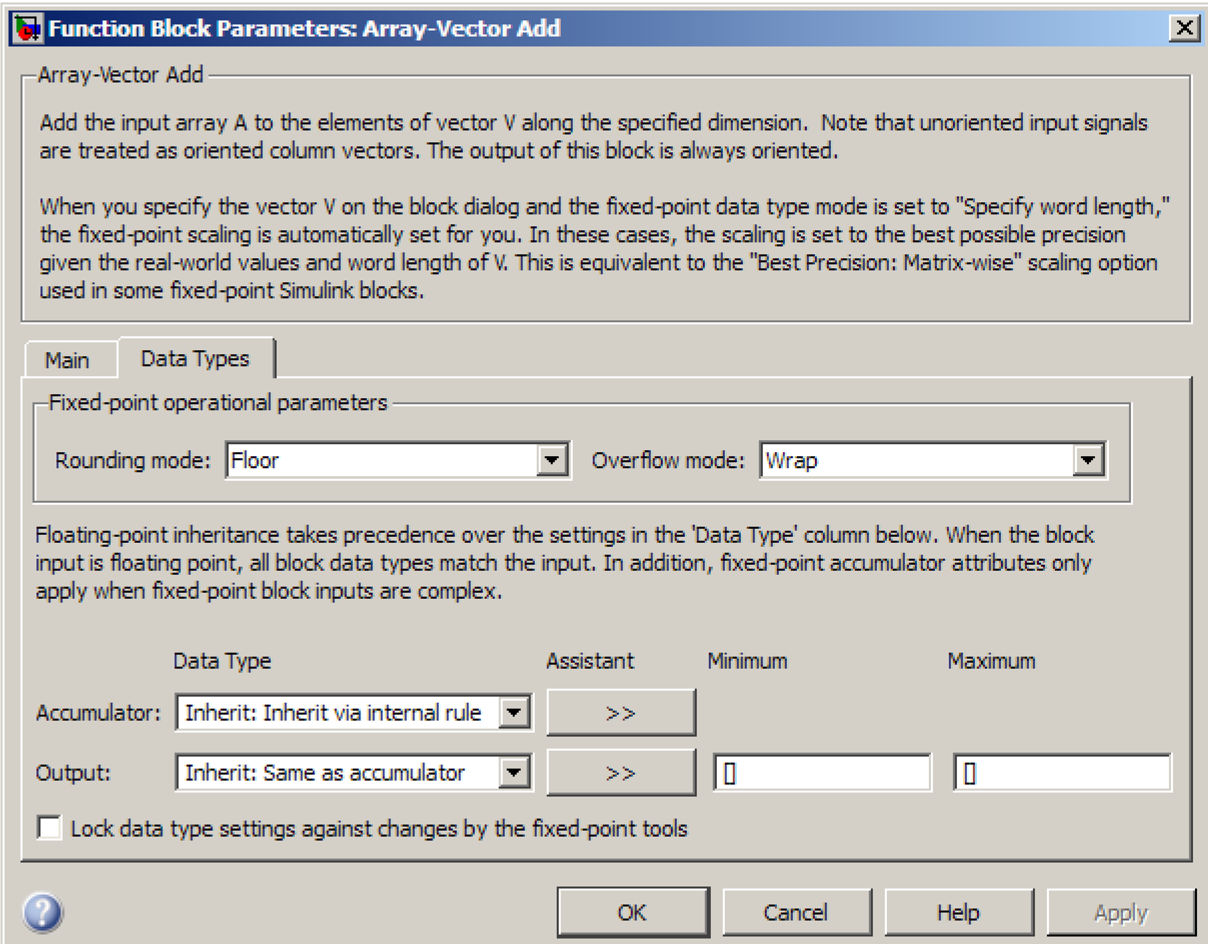
Vector (V) source

Specify the source of the vector, V. The vector can come from the `Input port` or from a `Dialog parameter`.

Vector (V)

Specify the vector, *V*. This parameter is visible only when you select `Dialog parameter` for the **Vector (V) source** parameter.

The **Data Types** pane of the Array-Vector Add block dialog appears as follows.



Note: Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations. If Accumulator Data Type is Inherit: Inherit via internal rule and Output Data Type is Inherit: Same as accumulator, the value of Rounding mode does not affect the numerical results.

Note: The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when both of the following conditions exist:

- **Accumulator data type** is `Inherit: Inherit via internal rule`
- **Output data type** is `Inherit: Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.

Overflow mode

Select the overflow mode for fixed-point operations.

Vector (V)

Use this parameter to specify the word and fraction lengths for the elements of the vector, *V*. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Note The **Vector (V)** parameter on the **Data Types** pane is only visible when you select `Dialog parameter` for the **Vector (V) source** parameter on the **Main** pane of the block mask. When the vector comes in through the block's input port, the data type and scaling of its elements are inherited from the driving block.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-58 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-58 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
V	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

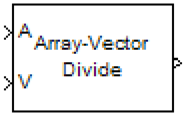
See Also

`Array-Vector Divide`	DSP System Toolbox
`Array-Vector Multiply`	DSP System Toolbox
`Array-Vector Subtract`	DSP System Toolbox

Introduced in R2007b

Array-Vector Divide

Divide array by vector along specified dimension



Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

Description

The Array-Vector Divide block divides the values in the specified dimension of the N-dimensional input array *A* by the values in the input vector *V*.

The length of the input *V* must be the same as the length of the specified dimension of *A*. The Array-Vector Divide block divides each element of *V* by the corresponding element along that dimension of *A*.

Consider a 3-dimensional *M*-by-*N*-by-*P* input array *A(i,j,k)* and a *N*-by-1 input vector *V*. When the **Divide along dimension** parameter is set to 2, the output of the block *Y(i,j,k)* is

$$Y(i,j,k) = \frac{A(i,j,k)}{V(j)}$$

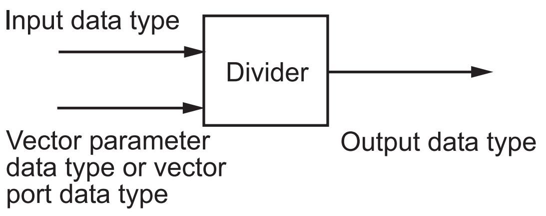
where

$$1 \le i \le M$$
$$1 \le j \le N$$
$$1 \le k \le P$$

The output of the Array-Vector Divide block is the same size as the input array, *A*. This block accepts real and complex floating-point and fixed-point input arrays, and real floating-point and fixed-point input vectors.

Fixed-Point Data Types

The following diagram shows the data types used within the Array-Vector Divide block for fixed-point signals.

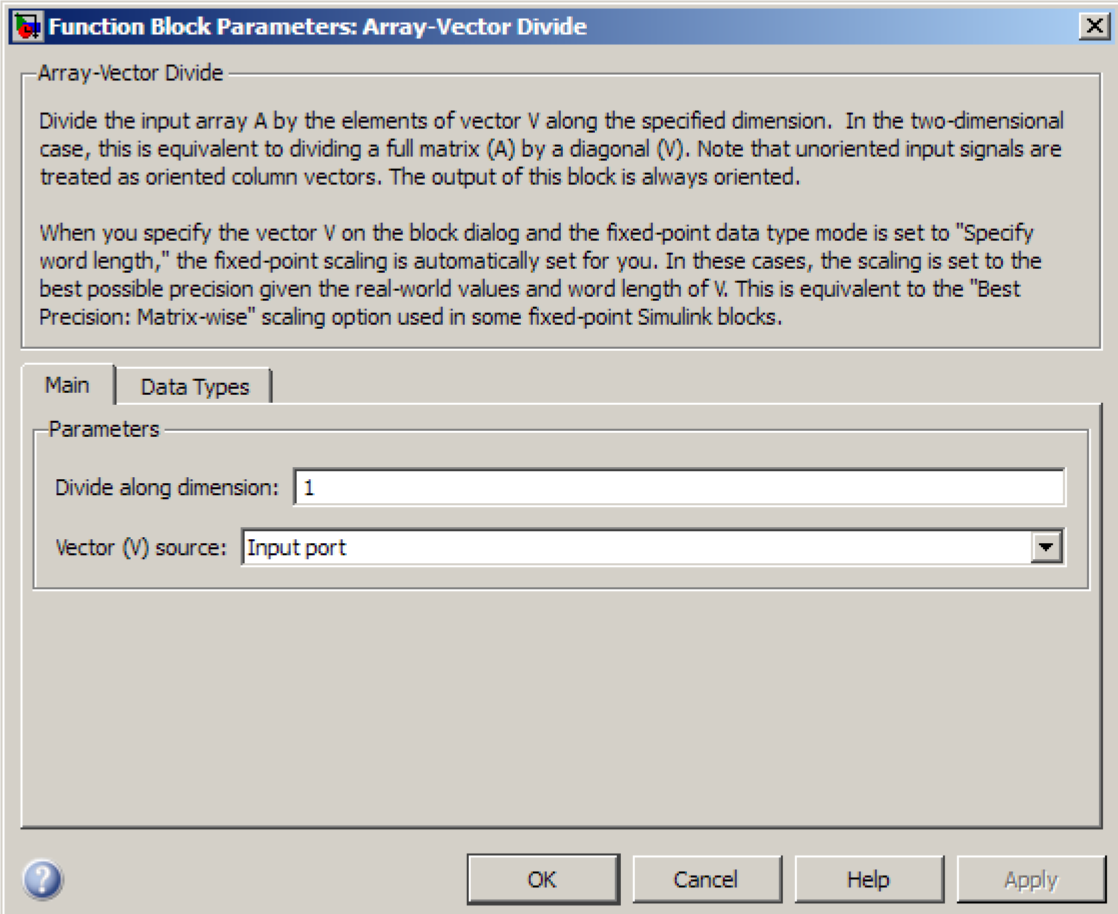


When you specify the vector *V* on the **Main** pane of the block mask, you must specify the data type and scaling properties of its elements in the **Vector (V)** parameter on the **Data Types** tab. When the vector comes in through the block port, its elements inherit their data type and scaling from the driving block.

You can set the vector and output data types in the block dialog as discussed below.

Dialog Box

The **Main** pane of the Array-Vector Divide block dialog appears as follows.



Divide along dimension

Specify the dimension along which to divide the input array *A* by the elements of vector *V*.

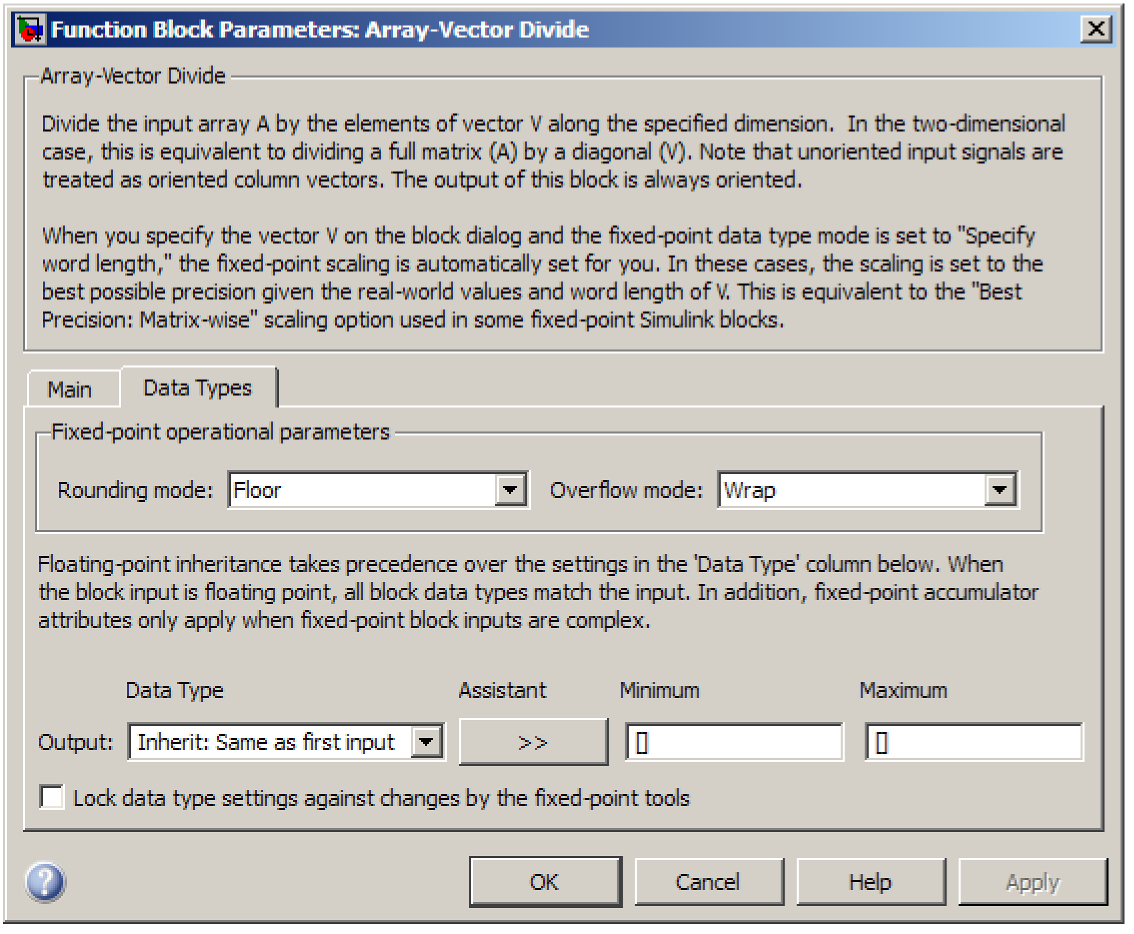
Vector (V) source

Specify the source of the vector, *V*. The vector can come from the `Input port` or from a `Dialog parameter`.

Vector (V)

Specify the vector, *V*. This parameter is visible only when you select `Dialog parameter` for the **Vector (V) source** parameter.

The **Data Types** pane of the Array-Vector Divide block dialog appears as follows.



Note: Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Vector (V)

Use this parameter to specify the word and fraction lengths for the elements of the vector, *V*. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Note The **Vector (V)** parameter on the **Data Types** pane is only visible when you select `Dialog parameter` for the **Vector (V) source** parameter on the **Main** pane of the block mask. When the vector comes in through the block's input port, the data type and scaling of its elements are inherited from the driving block.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-65 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as first input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
V	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

See Also

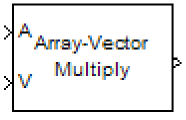
`Array-Vector Add`	DSP System Toolbox

`Array-Vector Multiply`	DSP System Toolbox
`Array-Vector Subtract`	DSP System Toolbox

Introduced in R2007b

Array-Vector Multiply

Multiply array by vector along specified dimension



Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

Description

The Array-Vector Multiply block multiplies the values in the specified dimension of the N-dimensional input array *A* by the values in the input vector *V*.

The length of the input *V* must be the same as the length of the specified dimension of *A*. The Array-Vector Multiply block multiplies each element of *V* by the corresponding element along that dimension of *A*.

Consider a 3-dimensional *M*-by-*N*-by-*P* input array *A(i,j,k)* and a *N*-by-1 input vector *V*. When the **Multiply along dimension** parameter is set to 2, the output of the block *Y(i,j,k)* is

$$Y(i,j,k) = A(i,j,k) * V(j)$$

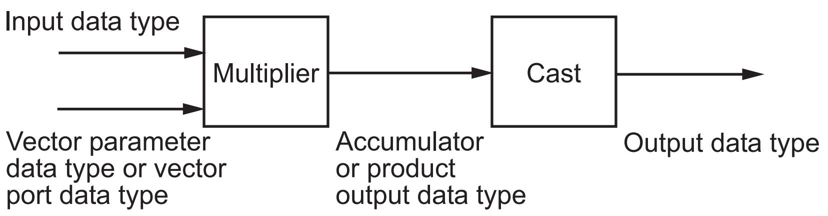
where

$$1 \le i \le M$$
$$1 \le j \le N$$
$$1 \le k \le P$$

The output of the Array-Vector Multiply block is the same size as the input array, *A*. This block accepts real and complex floating-point and fixed-point inputs.

Fixed-Point Data Types

The following diagram shows the data types used within the Array-Vector Multiply block for fixed-point signals.



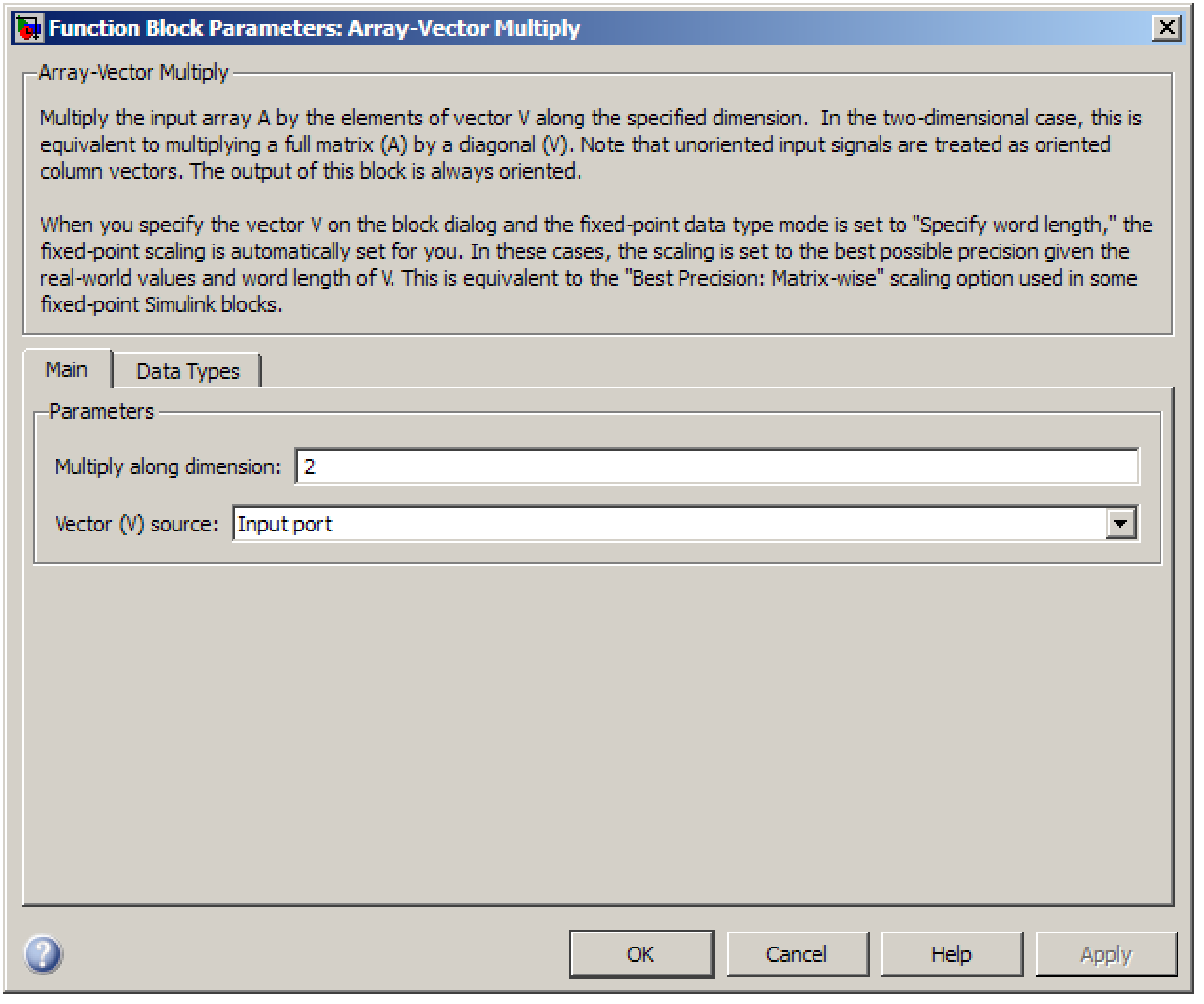
When you specify the vector *V* on the **Main** pane of the block mask, you must specify the data type and scaling properties of its elements in the **Vector (V)** parameter on the **Data Types** tab. When the vector comes in through the block port, its elements inherit their data type and scaling from the driving block.

The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

You can set the vector, accumulator, product output, and output data types in the block dialog as discussed below.

Dialog Box

The **Main** pane of the Array-Vector Multiply block dialog appears as follows.



Multiply along dimension

Specify the dimension along which to multiply the input array *A* by the elements of vector *V*.

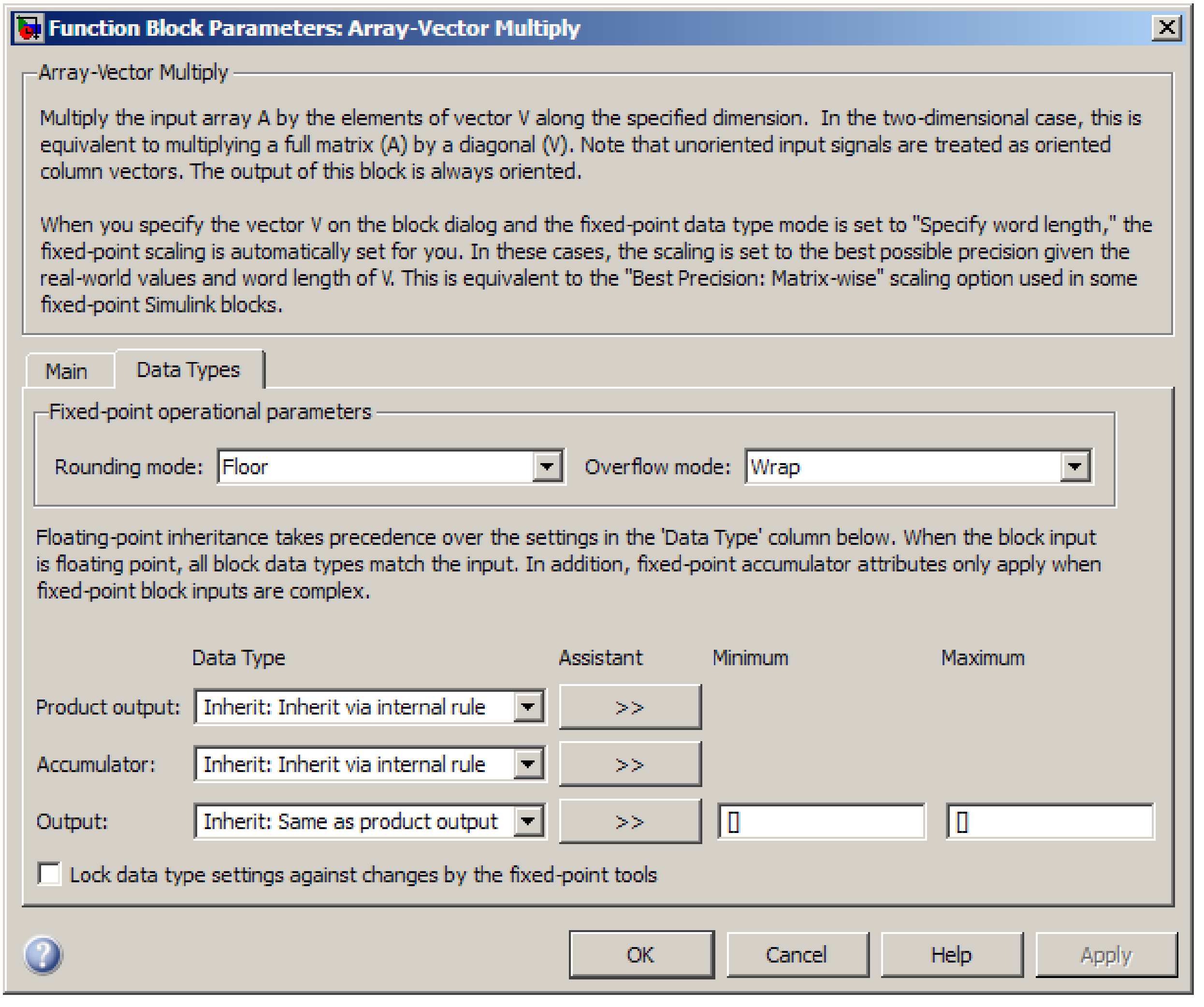
Vector (V) source

Specify the source of the vector, *V*. The vector can come from the `Input port` or from a `Dialog parameter`.

Vector (V)

Specify the vector, *V*. This parameter is visible only when you select `Dialog parameter` for the **Vector (V) source** parameter.

The **Data Types** pane of the Array-Vector Multiply block dialog appears as follows.



Note: Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Vector (V)

Use this parameter to specify the word and fraction lengths for the elements of the vector, *V*. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Note The **Vector (V)** parameter on the **Data Types** pane is only visible when you select `Dialog parameter` for the **Vector (V) source** parameter on the **Main** pane of the block mask. When the vector comes in through the block's input port, the data type and scaling of its elements are inherited from the driving block.

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-72 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-72 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-72 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
V	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

See Also

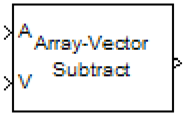
`Array-Vector Add`	DSP System Toolbox

`Array-Vector Divide`	DSP System Toolbox
`Array-Vector Subtract`	DSP System Toolbox

Introduced in R2007b

Array-Vector Subtract

Subtract vector from array along specified dimension



Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

Description

The Array-Vector Subtract block subtracts the values in the input vector V from the values in the specified dimension of the N-dimensional input array A.

The length of the input V must be the same as the length of the specified dimension of A. The Array-Vector Subtract block subtracts each element of V from the corresponding element along that dimension of A.

Consider a 3-dimensional M-by-N-by-P input array $A(i,j,k)$ and a N-by-1 input vector V. When the **Subtract along dimension** parameter is set to `2`, the output of the block $Y(i,j,k)$ is

$$Y(i,j,k) = A(i,j,k) - V(j)$$

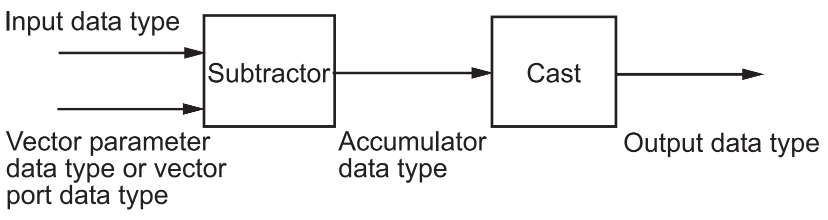
where

$$1 \le i \le M$$
$$1 \le j \le N$$
$$1 \le k \le P$$

The output of the Array-Vector Subtract block is the same size as the input array, *A*. This block accepts real and complex floating-point and fixed-point inputs.

Fixed-Point Data Types

The following diagram shows the data types used within the Array-Vector Subtract block for fixed-point signals.



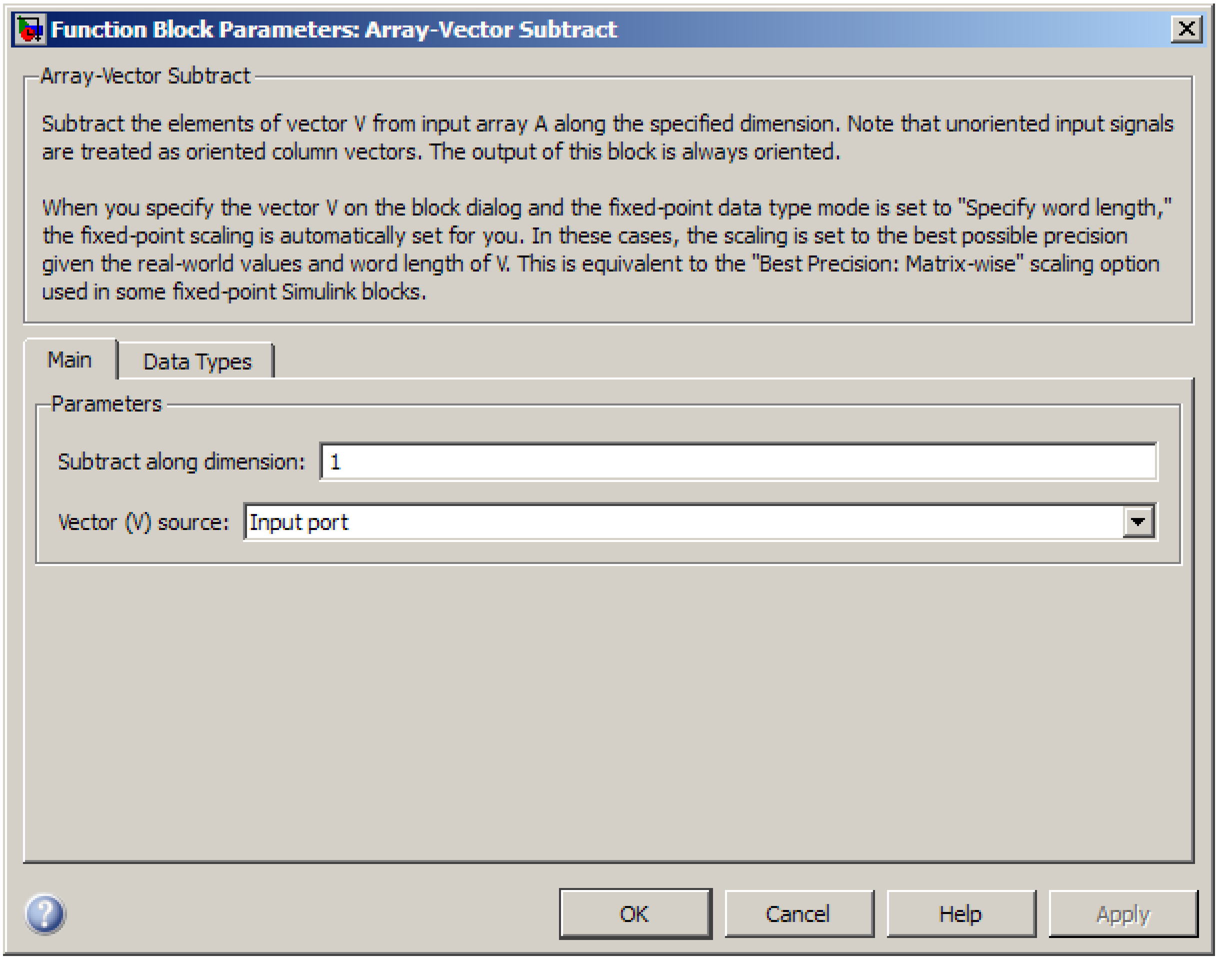
When you specify the vector *V* on the **Main** pane of the block mask, you must specify the data type and scaling properties of its elements in the **Vector (V)** parameter on the **Data Types** tab. When the vector comes in through the block port, its elements inherit their data type and scaling from the driving block.

The output of the subtractor is in the accumulator data type.

You can set the vector, accumulator, and output data types in the block dialog as discussed below.

Dialog Box

The **Main** pane of the Array-Vector Subtract block dialog appears as follows.



Subtract along dimension

Specify the dimension along which to subtract the elements of vector *V* from the input array *A*.

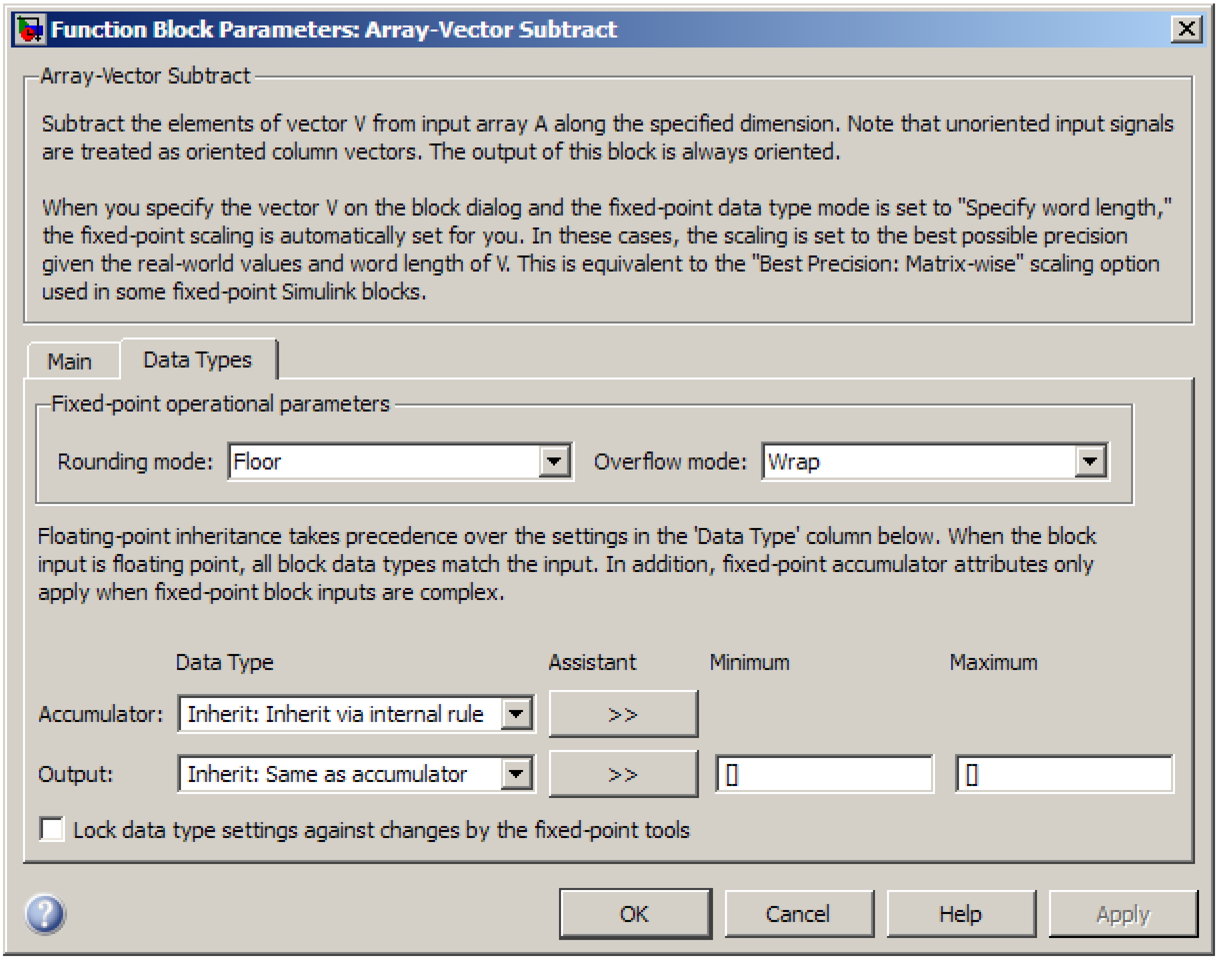
Vector (V) source

Specify the source of the vector, *V*. The vector can come from the `Input port` or from a `Dialog parameter`.

Vector (V)

Specify the vector, *V*. This parameter is visible only when you select `Dialog parameter` for the **Vector (V) source** parameter.

The **Data Types** pane of the Array-Vector Subtract block dialog appears as follows.



Note: Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.

Note: The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when both of the following conditions exist:

- **Accumulator data type** is `Inherit: Inherit via internal rule`
- **Output data type** is `Inherit: Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.

Overflow mode

Select the overflow mode for fixed-point operations.

Vector (V)

Use this parameter to specify the word and fraction lengths for the elements of the vector, *V*. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Note The **Vector (V)** parameter on the **Data Types** pane is only visible when you select `Dialog parameter` for the **Vector (V) source** parameter on the **Main** pane of the block mask. When the vector comes in through the block's input port, the data type and scaling of its elements are inherited from the driving block.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-80 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-80 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
V	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

See Also

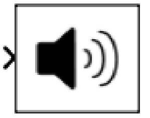
`Array-Vector Add`	DSP System Toolbox
`Array-Vector Divide`	DSP System Toolbox
`Array-Vector Multiply`	DSP System Toolbox

Introduced in R2007b

Audio Device Writer

Play to sound card

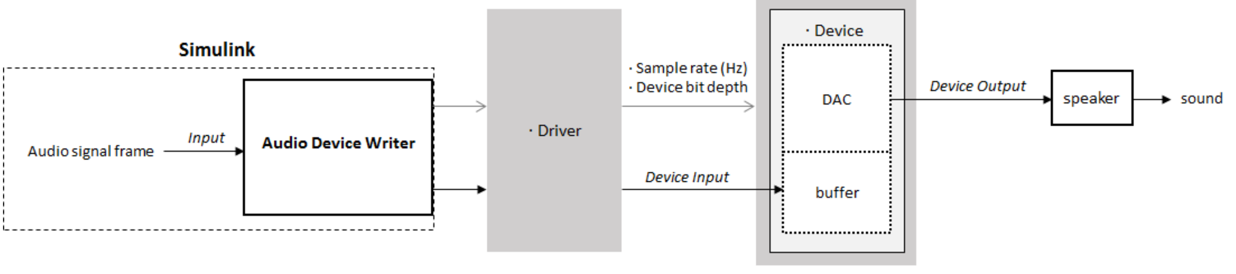
Library: DSP System Toolbox / Sinks



Description

The Audio Device Writer block writes audio samples to an audio output device.

Parameters of the Audio Device Writer block specify the driver, the device, and device attributes such as sample rate and bit depth.



Data Flow of Audio Device Writer Block

- An audio signal frame is input to the Audio Device Writer block.
- The Audio Device Writer block uses the specified driver to pass the frame (device input) to your specified audio device buffer.
- The audio device performs digital-to-analog conversion at the specified sample rate and bit depth.
- The audio device outputs an analog chunk to your speaker.

Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

If input to the Audio Device Writer block is of data type `double` or `single`, the block clips values outside the range [–1, 1]. For other data types, the allowed input range is [min, max] of the specified data type.

Data Types: `single` | `double` | `int16` | `int32` | `uint8`

Output

`Port_1` — Number of samples underrun

scalar

This port outputs the number of samples underrrun while writing a frame of data (one input matrix).

Dependencies

To enable this port, select the **Output number of samples underrun** parameter.

Data Types: `uint32`

Parameters

Main Tab

`Driver` — Driver used to access your audio device

`DirectSound` (default) | `ASIO` | `WASAPI`

- ASIO drivers do not come pre-installed on Windows machines. To use the `ASIO` driver option, install an ASIO driver outside of MATLAB.

Note: If **Driver** is set to `ASIO`, open the ASIO UI outside of MATLAB to set the sound card buffer size to the frame size (number of rows) input to the Audio Device Writer block. See the documentation of your ASIO driver for more information.

- WASAPI drivers are supported for exclusive-mode only.

ASIO and WASAPI drivers do not provide sample rate conversion. For ASIO and WASAPI drivers, supply an audio stream with a sample rate supported by your audio device.

This parameter applies only on Windows machines. Linux machines always use the ALSA driver. Mac machines always use the CoreAudio driver.

To specify nondefault **Driver** values, you must install Audio System Toolbox™. If the toolbox is not installed, specifying nondefault **Driver** values returns an error.

`Device` — Device used to play audio samples

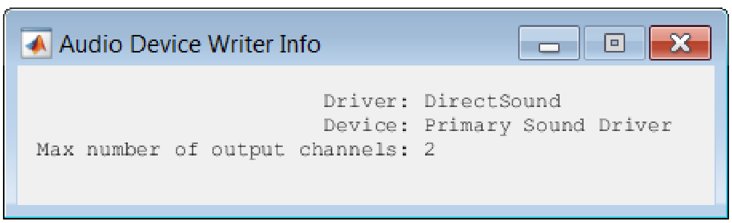
default audio device (default)

The device list is populated with devices available on your computer.

`Info` — View information about your audio output configuration

button

This button opens a dialog box that lists your selected audio driver, the full name of your audio device, and the maximum output channels for your configuration. For example:



`Inherit sample rate from input` — Specify source of input sample rate

on (default) | off

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Sample rate (Hz)**.

`Sample rate (Hz)` — Sample rate used by device to play audio data
`44100` (default) | positive scalar

The possible range of **Sample rate (Hz)** depends on your audio hardware.

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Advanced Tab

`Device bit depth` — Data type used by device to perform digital-to-analog conversion
`16-bit integer` (default) | `8-bit integer` | `24-bit integer` | `32-bit float`

Before performing digital-to-analog conversion, the input data is cast to a data type specified by this parameter.

Note: To specify a nondefault **Device bit depth**, you must install Audio System Toolbox. If the toolbox is not installed, specifying a nondefault **Device bit depth** returns an error.

`Use default mapping between columns of input of this block and sound card's output channels` — Toggle channel mapping source
on (default) | off

When you select this parameter, the block uses the default mapping between columns of the matrix input to this block and the channels of your device. When you clear this parameter, you specify the mapping in **Device output channels**.

`Device output channels` — Specify nondefault channel mapping
`[1:MaximumOutputChannels]` (default) | scalar | vector

Nondefault mapping between columns of matrix input to the Audio Device Writer block and channels of output device, specified as a scalar or vector. For example:

If **Device output channels** is specified as `1:3`, then:

- The first column of the input matrix maps to channel 1.
- The second column of the input matrix maps to channel 2.

- The third column of the input matrix maps to channel 3.

If **Device output channels** is specified as `[3,1,2]`, then:

- The first column of the input matrix maps to channel 3.
- The second column of the input matrix maps to channel 1.
- The third column of the input matrix maps to channel 2.

Note: To selectively map between columns of the input matrix and your sound card's output channels, you must install Audio System Toolbox. If the toolbox is not installed, specifying nondefault values for **Device output channels** returns an error.

Dependencies

To enable this parameter, clear the **Use default mapping between columns of input of this block and sound card's output channels** parameter.

`Output number of samples underrun` — Specify output port for number of samples underrun
off (default) | on

When you select this parameter, an output port is added to the block. The port outputs the number of samples underrrun while writing a frame of data (one input matrix).

Model Examples

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

The executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

See Also

See Also

System Objects

audioDeviceWriter | `audioplayer` | `sound`

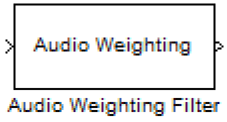
Topics

“How To Run a Generated Executable Outside MATLAB”

Introduced in R2016a

Audio Weighting Filter

Design audio weighting filter



Note: The Audio Weighting Filter block will be removed from DSP System Toolbox in a future release. Existing instances of the block continue to run. For new code, use the Audio Weighting Filter block from Audio System Toolbox instead.

Library

Filtering / Filter Designs

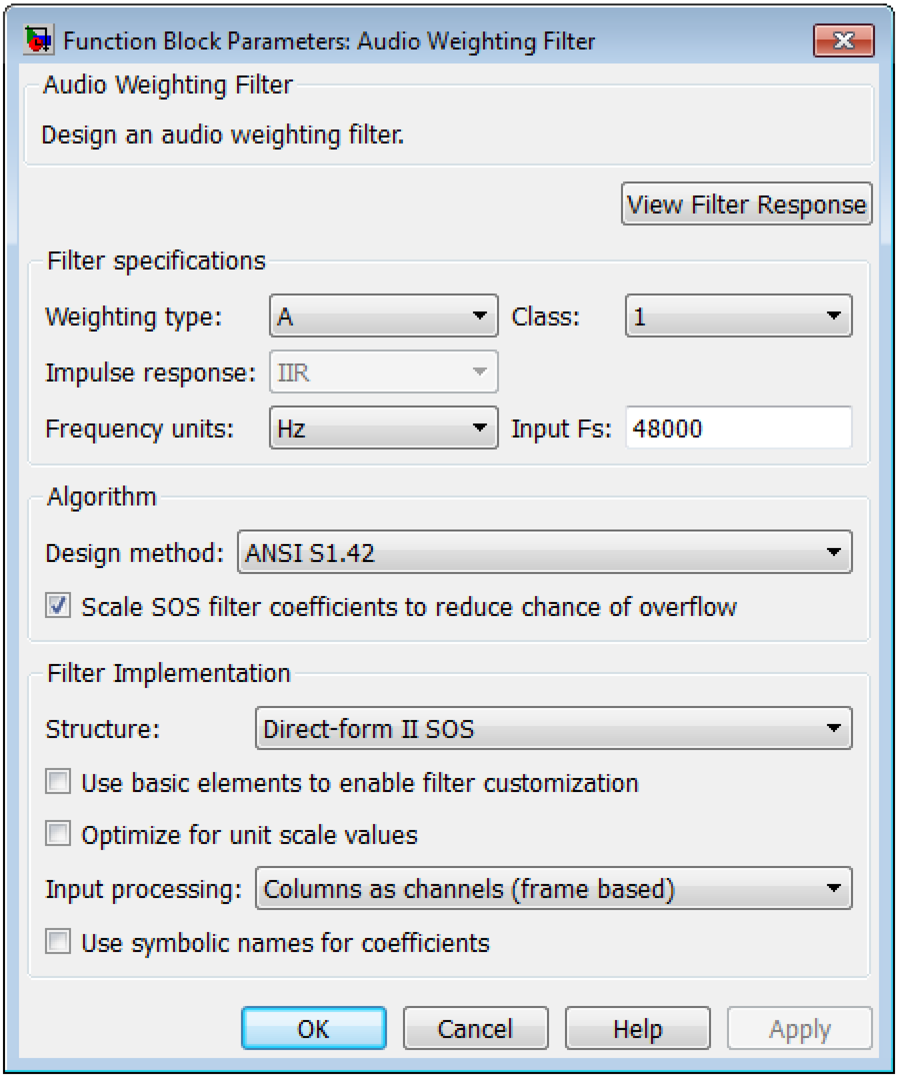
`dspfdesign`

Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See "Audio Weighting Filter Design — Main Pane" on page 5-714 for more information about the parameters of this block. The **Data Types** and **Code** panes are not available for blocks in the DSP System Toolbox Filter Designs library.



View Filter Response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Weighting type

The weighting type defines the frequency response of the filter. The valid weighting types for this filter are `A`, `C` , `C-message`, `ITU-R 468–4`, and `ITU-T 0.41`. For definitions of the available weighting types, see the `fdesign.audioweighting` reference page.

Class

The filter class describes the frequency-dependent tolerances specified in the relevant standards [1], [2]. There are two possible class values: `1` and `2`. Class 1 weighting filters have stricter tolerances than class 2 filters. The filter class value does not affect the design. The class value is only used to provide a specification mask in `fvtool` for the analysis of the filter design. The default value of this parameter is `1`.

The filter class is only applicable for `A` weighting and `C` weighting filters.

Impulse response

Specify the impulse response type as one of `IIR` or `FIR`. For A, C , C-message, and ITU-R 468–4 filter, `IIR` is the only option. For a ITU-T 0.41 weighting filter, `FIR` is the only option.

Frequency units

Specify the frequency units as Hertz (Hz), kilohertz (kHz), megahertz (MHz), or gigahertz (GHz). Normalized frequency designs are not supported for audio weighting filters. The default value of this parameter is `Hz`.

Input Fs

Specify the input sampling frequency. The units correspond to the setting of the **Frequency units** parameter.

Algorithm

Design Method

Valid design methods depend on the weighting type. For type A and C weighting filters, the only valid design type is `ANSI S1.42`. This is an IIR design method that follows ANSI standard S1.42–2001. For a C message filter, the only valid design method is `Bell 41009`, which is an IIR design method following the Bell System Technical Reference PUB 41009. For a ITU-R 468–4 weighting filter, you can design an IIR or FIR filter. If you choose an IIR design, the design method is `IIR least p-norm`. If you choose an FIR design, the design method choices are `Equirriple` or `Frequency Sampling`. For an ITU-T 0.41 weighting filter, the available FIR design methods are `Equirriple` or `Frequency Sampling`.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. For audio weighting IIR filter designs, you can choose direct form I or II biquad (SOS). You can also choose to implement these structures in transposed form.

For FIR designs, you can choose a direct form, direct-form transposed, direct-form symmetric, or direct-form asymmetric structure.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

For more information about sample- and frame-based processing, see “Sample- and Frame-Based Concepts”.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
Output	• Double-precision floating point • Single-precision floating point

References

[1] *American National Standard Design Response of Weighting Networks for Acoustical Measurements*, ANSI S1.42-2001, Acoustical Society of America, New York, NY, 2001.

[2] *Electroacoustics Sound Level Meters Part 1: Specifications*, IEC 61672-1, First Edition 2002-05.

See Also

`fdesign.audioweighting` | `filterbuilder` | `fvtool`

Topics

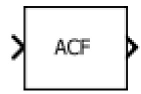
Audio Weighting Filters Example

Introduced in R2011b

Autocorrelation

Autocorrelation of *N*-D array

Library: DSP System Toolbox / Statistics



Description

The Autocorrelation block computes the autocorrelation along the first dimension of an *N*-D input array. The computation can be done in the time domain or frequency domain. You can specify the domain through the **Computation domain** parameter. In the time domain, the input signal is convolved with its time-reversed complex conjugate. In the frequency domain, the block computes the autocorrelation by taking the Fourier transform of the input signal, multiplying the Fourier transform with its conjugate, and computing the inverse Fourier transform of the product. In this domain, depending on the input length, the block can require fewer computations. For information on these two computation methods, see "Algorithms" on page 2-104.

You can specify the maximum lag for autocorrelation using the **Compute all non-negative lags** and **Maximum non-negative lag (less than input length)** parameters.

The block accepts fixed-point signals when you set the **Computation domain** to `Time`.

Ports

Input

`Port_1` — Data input

vector | matrix | *N*-D array

Data input. The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input can be a fixed-point signal when you set the **Computation domain** to `Time`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed_point`

Complex Number Support: Yes

Output

`Port_1` — Autocorrelated output
vector | matrix | *N*-D array

Autocorrelated output of the data input.

- When the input is an *M*-by-*N* matrix, *u*, the output, *y*, is an (*l*+1)-by-*N* matrix. *l* is the maximum positive lag for autocorrelation.
- When the input is an *N*-D array, the block outputs an *N*-D array. The size of the first dimension is *l*+1, and the sizes of all other dimensions match those of the input array. For example, when the input is an *M*-by-*N*-by-*P* array, the block outputs an (*l*+1)-by-*N*-by-*P* array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed_point`
Complex Number Support: Yes

Parameters

Main Tab

`Compute all non-negative lags` — Compute autocorrelation over all nonnegative lags
on (default) | off

When you select this parameter, the Autocorrelation block computes the autocorrelation over all nonnegative lags in the range [0, `length(input)` – 1]. When you clear this parameter, the block computes the autocorrelation using the lags in the range [0, *l*], where *l* is the value you specify in **Maximum non-negative lag (less than input length)**.

`Maximum non-negative lag (less than input length)` — Maximum positive lag
`1` (default) | integer greater than or equal to 0 and less than input length

Maximum positive lag for autocorrelation, specified as an integer that is greater than or equal to 0 and less than the input length.

Dependencies

To enable this parameter, clear the **Compute all non-negative lags** parameter.

`Scaling` — Scaling of the output

`None` (default) | `Biased` | `Unbiased` | `Unity at zero-lag`

Scaling applied to the output.

- `None` — Generates the raw autocorrelation $y_{i,j}$ without normalization.
- `Biased` — Generates the biased estimate of the autocorrelation.

  $$y_{i,j}^{biased} = \frac{y_{i,j}}{M}$$

- `Unbiased` — Generates the unbiased estimate of the autocorrelation.

  $$y_{i,j}^{unbiased} = \frac{y_{i,j}}{M-i}$$

- `Unity at zero-lag` — Normalizes the estimate of the autocorrelation for each channel so that the zero-lag sum, the first element in each column, is identically 1.

  $$y_{0,j} = 1$$

`Computation domain` — Domain in which the block computes the autocorrelation

`Time` (default) | `Frequency`

- `Time` — Computes the convolutions in the time domain, which minimizes the memory usage.
- `Frequency` — Computes the autocorrelation in frequency domain. For more information, see “Algorithms” on page 2-104.

To autocorrelate fixed-point signals, set this parameter to `Time`.

Data Types Tab

Note Fixed-point signals are supported for the time domain only. To use these parameters, on the **Main** tab, set **Computation domain** to `Time`.

`Rounding mode` — Method of rounding operation

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Select the rounding mode for fixed-point operations.

Note: The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numerical results when all these conditions are met:

- **Product output** data type is `Inherit: Inherit via internal rule`.
- **Accumulator** data type is `Inherit: Inherit via internal rule`.
- **Output** data type is `Inherit: Same as accumulator`.

With these data type settings, the block operates in full-precision mode.

`Saturate on integer overflow` — Saturate on integer overflow

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations.

`Product output` — Product output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Same as input` | `fixdt([],16,0)`

Product output specifies the data type of the output of a product operation in the Autocorrelation block. For more information on the product output data type, see “Multiplication Data Types” and the 'Fixed-Point Conversion' section in “Fixed Point” on page 2- .

- `Inherit: Inherit via internal rule` — The block inherits the product output data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.
- `Inherit: Same as input` — The block specifies the product output data type to be the same as the input data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink).

`Accumulator` — Accumulator data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Same as input` | `Inherit: Same as product output` | `fixdt([],16,0)`

Accumulator specifies the data type of the output of an accumulation operation in the Autocorrelation block. For illustrations on how to use the accumulator data type in this block, see the 'Fixed-Point Conversion' section in "Fixed Point" on page 2- .

- `Inherit: Inherit via internal rule` — The block inherits the accumulator data type based on an internal rule. For more information on this rule, see "Inherit via Internal Rule".
- `Inherit: Same as input` — The block specifies the accumulator data type to be the same as the input data type.
- `Inherit: Same as product output` — The block specifies the accumulator data type to be the same as the product output data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink).

`Output` — Output data type

`Inherit: Same as accumulator` (default) | `Inherit: Same as input` | `Inherit: Same as product output` | `fixdt([],16,0)`

Output specifies the data type of the output of the Autocorrelation block. For more information on the output data type, see the 'Fixed-Point Conversion' section in "Fixed Point" on page 2- .

- `Inherit: Same as input` — The block specifies the output data type to be the same as the input data type.

- `Inherit: Same as product output` — The block specifies the output data type to be the same as the product output data type.
- `Inherit: Same as accumulator` — The block specifies the output data type to be the same as the accumulator data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink).

`Output Minimum` — Minimum value the block can output

[ ] (default) | scalar

Specify the minimum value the block can output. Simulink software uses this minimum value to perform:

- Simulation range checking. See "Signal Ranges" (Simulink).
- Automatic scaling of fixed-point data types.

`Output Maximum` — Maximum value block can output

[ ] (default) | scalar

Specify the maximum value the block can output. Simulink software uses this maximum value to perform:

- Simulation range checking. See "Signal Ranges" (Simulink).
- Automatic scaling of fixed-point data types.

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block dialog box.

Definitions

Autocorrelation

Autocorrelation is the correlation of a signal with itself at different points in time.

For a deterministic discrete-time sequence, *x(n)*, the autocorrelation is computed using the following relationship:

$$r_x(h) = \sum_{n=0}^{N-h-1} x^*(n)x(n+h) \quad h = 0,1,\ldots,N-1$$

where h is the lag and * denotes the complex conjugate. If the input is a length N realization of a WSS stationary random process, $r_x(h)$ is an estimate of the theoretical autocorrelation:

$$\rho_x(h) = E\{x^*(n)x(n+h)\}$$

where E{ } is the expectation operator. The `Unity at zero-lag` normalization divides each sequence value by the autocorrelation or autocorrelation estimate at zero lag.

$$\frac{\rho_x(h)}{\rho_x(0)} = \frac{E\{x^*(n)x(n+h)\}}{E\{|x(0)|^2\}}$$

The most commonly used estimate of the theoretical autocorrelation of a WSS random process is the biased estimate:

$$\hat{\rho}_x(h) = \frac{1}{N} \sum_{k=0}^{N-h-1} x^*(n)x(n+h)$$

Algorithms

Time domain Computation

When you set the computation domain to time, the algorithm computes the autocorrelation of the input signal in the time domain. The input signal can be a fixed-point signal in this domain.

The autocorrelation sequence, y, is computed using this equation:

$$y_{i,j} = \sum_{k=0}^{M-l-1} u^*_{k,j} u_{(k+i),j} \qquad 0 \le i \le l$$

- $y_{0,j}$ is the zero-lag element in the jth column of the input.
- i is the index of the lag.
- j is the index of the input data column.
- * denotes the complex conjugate.
- M is the number of elements in each column.
- l is the maximum positive lag for autocorrelation. When you choose to compute the autocorrelation with all nonnegative lags, $l=M–1$. Otherwise, l is the maximum nonnegative integer lag value specified.
- u is an M-by-N input matrix.

Frequency Domain Computation

When you set the computation domain to frequency, the algorithm computes the autocorrelation in the frequency domain.

In this domain, the algorithm computes the autocorrelation sequence by taking the Fourier transform of the input signal, multiplying the Fourier transform with its complex conjugate, and taking the inverse Fourier transform of the product. In this domain, depending on the input length, the algorithm can require fewer computations.

Extended Capabilities

C/C++ Code Generation

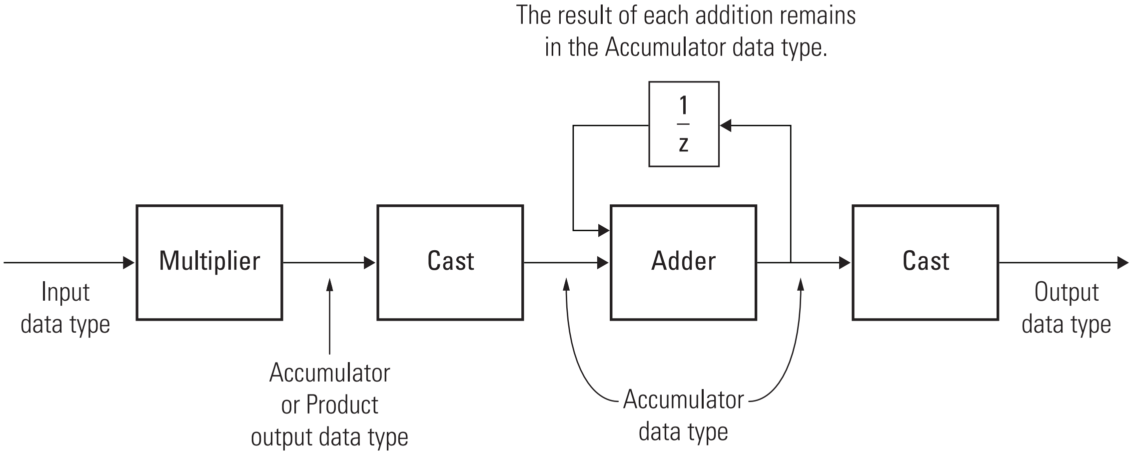
Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion

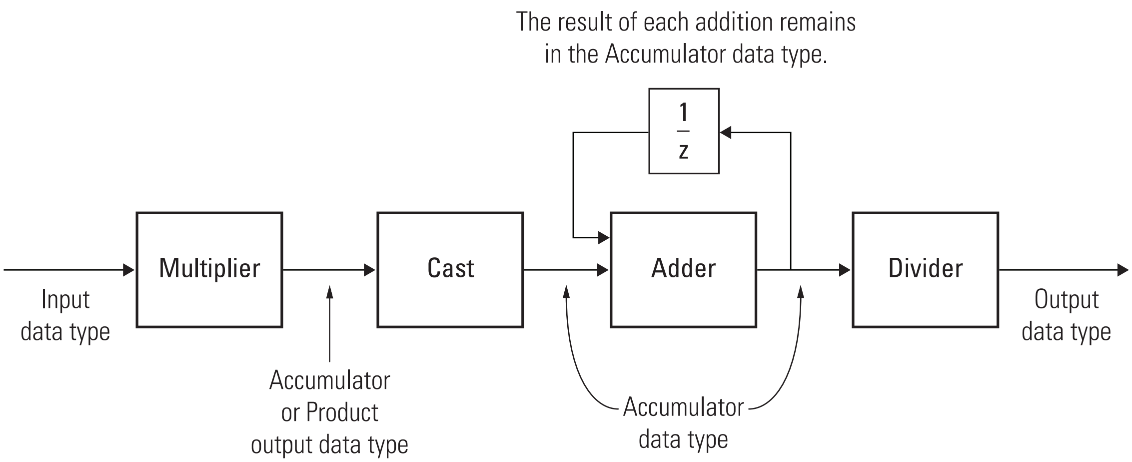
Design and simulate fixed-point algorithms using Fixed-Point Designer™.

These diagrams show the data types that the Autocorrelation block uses for fixed-point signals (time domain only).

Signal flow when Scaling is "None"



Signal flow when Scaling is other than "None"



You can set the product output, accumulator, and output data types on the **Data Types** tab of the block.

When the input is real, the output of the multiplier is in the product output data type. When the input is complex, the output of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types".

See Also

See Also

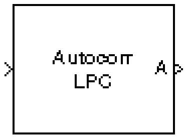
System Objects
dsp.Autocorrelator | dsp.Crosscorrelator

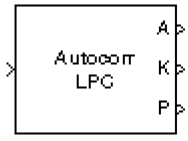
Blocks
`Correlation`

Introduced before R2006a

Autocorrelation LPC

Determine coefficients of Nth-order forward linear predictors





Library

Estimation / Linear Prediction

`dsplp`

Description

The Autocorrelation LPC block determines the coefficients of an *N-step forward linear predictor* for the time-series in each length-M input channel, u, by minimizing the prediction error in the least squares sense. A linear predictor is an FIR filter that predicts the next value in a sequence from the present and past inputs. This technique has applications in filter design, speech coding, spectral analysis, and system identification.

The Autocorrelation LPC block can output the prediction error for each channel as polynomial coefficients, reflection coefficients, or both. It can also output the prediction error power for each channel. The input u can be an unoriented vector, column vector, or a matrix. Row vectors are not valid inputs. The block treats all M-by-N matrix inputs as N channels of length M.

When you select **Inherit prediction order from input dimensions**, the prediction order, N, is inherited from the input dimensions. Otherwise, you can use the **Prediction order** parameter to specify the value of N. Note that N must be a scalar with a value less than the length of the input channels or the block produces an error.

When **Output(s)** is set to A, port A is enabled. For each channel, port A outputs an (N+1)-by-1 column vector, $a = [1\ a_2\ a_3\ \ldots\ a_{N+1}]^T$, containing the coefficients of an Nth-order moving average (MA) linear process that predicts the next value, $\hat{u}_{M+1}$, in the input time-series.

$$\hat{u}_{M+1} = -(a_2 u_M) - (a_3 u_{M-1}) - \ldots - (a_{N+1} u_{M-N+1})$$

When **Output(s)** is set to K, port K is enabled. For each channel, port K outputs a length-N column vector whose elements are the prediction error reflection coefficients. When **Output(s)** is set to `A and K`, both port A and K are enabled, and each port outputs its respective set of prediction coefficients for each channel.

When you select **Output prediction error power (P)**, port P is enabled. The prediction error power is output at port P as a vector whose length is the number of input channels.

Algorithm

The Autocorrelation LPC block computes the least squares solution to

$$\min_{\tilde{a} \in \Re^n} \| U\tilde{a} - b \|$$

where $\|\cdot\|$ indicates the 2-norm and

$$U = \begin{bmatrix} u_1 & 0 & \cdots & 0 \\ u_2 & u_1 & \ddots & \vdots \\ \vdots & u_2 & \ddots & 0 \\ \vdots & \vdots & \ddots & u_1 \\ \vdots & \vdots & \vdots & u_2 \\ \vdots & \vdots & \vdots & \vdots \\ u_M & \vdots & \vdots & \vdots \\ 0 & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_M \end{bmatrix}, \tilde{a} = \begin{bmatrix} a_2 \\ \vdots \\ a_n + 1 \end{bmatrix}, b = \begin{bmatrix} u_2 \\ u_3 \\ \vdots \\ u_M \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Solving the least squares problem via the normal equations

$$U^* U \tilde{a} = U^* b$$

leads to the system of equations

$$\begin{bmatrix} r_1 & r_2^* & \cdots & r_n^* \\ r_2 & r_1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & r_2^* \\ r_n & \cdots & r_2 & r_1 \end{bmatrix} \begin{bmatrix} a_2 \\ a_3 \\ \vdots \\ a_{n+1} \end{bmatrix} = \begin{bmatrix} -r_2 \\ -r_3 \\ \vdots \\ -r_{n+1} \end{bmatrix}$$

where $r = [r_1\ r_2\ r_3\ \ldots\ r_{n+1}]^T$ is an autocorrelation estimate for u computed using the Autocorrelation block, and * indicates the complex conjugate transpose. The normal equations are solved in $O(n^2)$ operations by the Levinson-Durbin block.

Note that the solution to the LPC problem is very closely related to the Yule-Walker AR method of spectral estimation. In that context, the normal equations above are referred to as the Yule-Walker AR equations.

Parameters

Output(s)

The type of prediction coefficients output by the block. The block can output polynomial coefficients (`A`), reflection coefficients (`K`), or both (`A and K`).

Output prediction error power (P)

When selected, enables port `P`, which outputs the output prediction error power.

Inherit prediction order from input dimensions

When selected, the block inherits the prediction order from the input dimensions.

Prediction order (N)

Specify the prediction order, N, which must be a scalar. This parameter is disabled when you select the **Inherit prediction order from input dimensions** parameter.

References

Haykin, S. *Adaptive Filter Theory*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1996.

Ljung, L. *System Identification: Theory for the User*. Englewood Cliffs, NJ: Prentice Hall, 1987. Pgs. 278-280.

Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

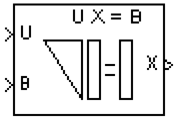
See Also

`Autocorrelation`	DSP System Toolbox
`Levinson-Durbin`	DSP System Toolbox
`Yule-Walker Method`	DSP System Toolbox
`lpc`	Signal Processing Toolbox

Introduced before R2006a

Backward Substitution

Solve $UX=B$ for X when U is upper triangular matrix



Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

`dspsolvers`

Description

The Backward Substitution block solves the linear system $UX=B$ by simple backward substitution of variables, where:

- U is the upper triangular M-by-M matrix input to the U port.
- B is the M-by-N matrix input to the B port.

The M-by-N output matrix X is the solution of the equations. The block does not check the rank of the inputs.

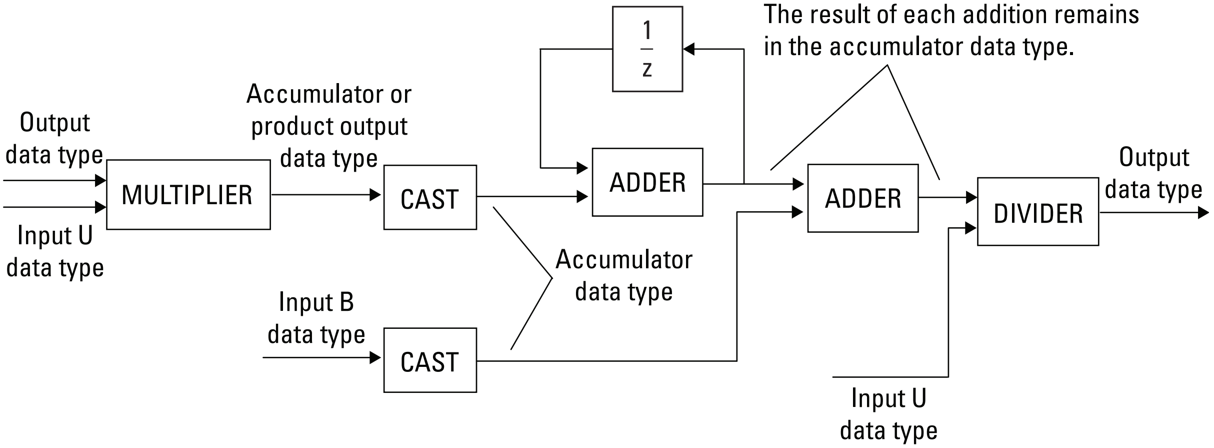
The block uses only the elements in the *upper triangle* of input U and ignores the lower elements. When you select the **Input U is unit-upper triangular** check box, the block assumes the elements on the diagonal of U are 1s. This is useful when matrix U is the result of another operation, such as an LDL decomposition, that uses the diagonal elements to represent the D matrix.

The block treats a length-M vector input at port B as an M-by-1 matrix.

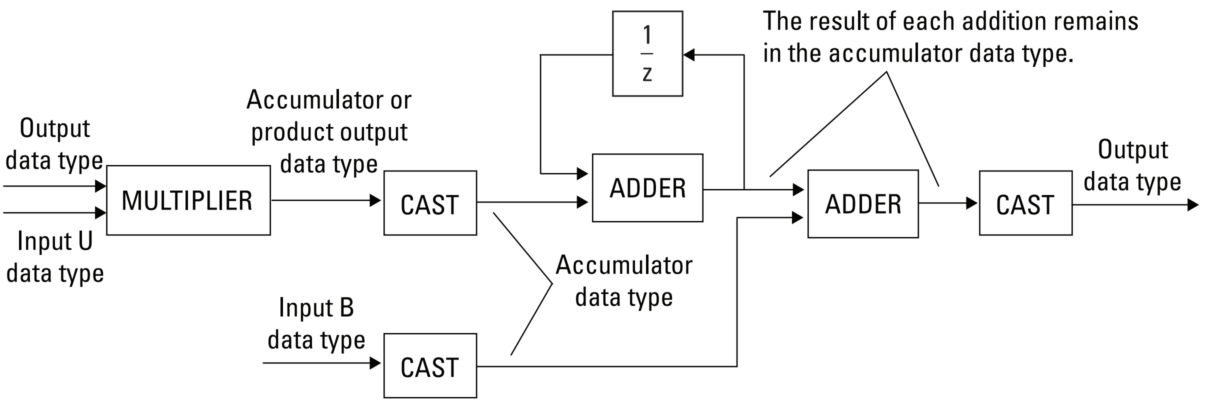
Fixed-Point Data Types

The following diagram shows the data types used within the Backward Substitution block for fixed-point signals.

When input U is not unit-upper triangular:



When input U is unit-upper triangular:

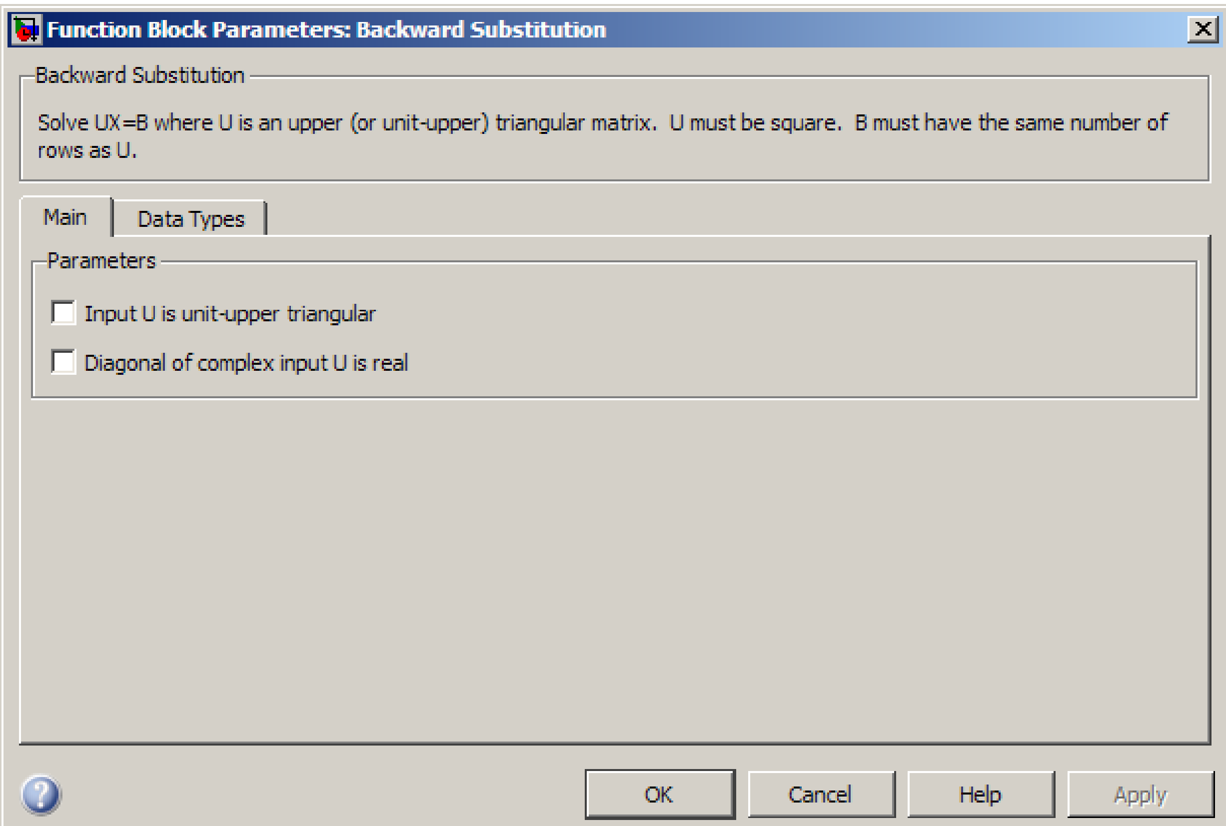


You can set the product output, accumulator, and output data types in the block dialog as discussed in the following section.

The output of the multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types".

Dialog Box

The **Main** pane of the Backward Substitution block dialog box appears as follows.



Input U is unit-upper triangular

Select this check box only when all elements on the diagonal of *U* have a value of 1. When you do so, the block optimizes its behavior by skipping an unnecessary divide operation.

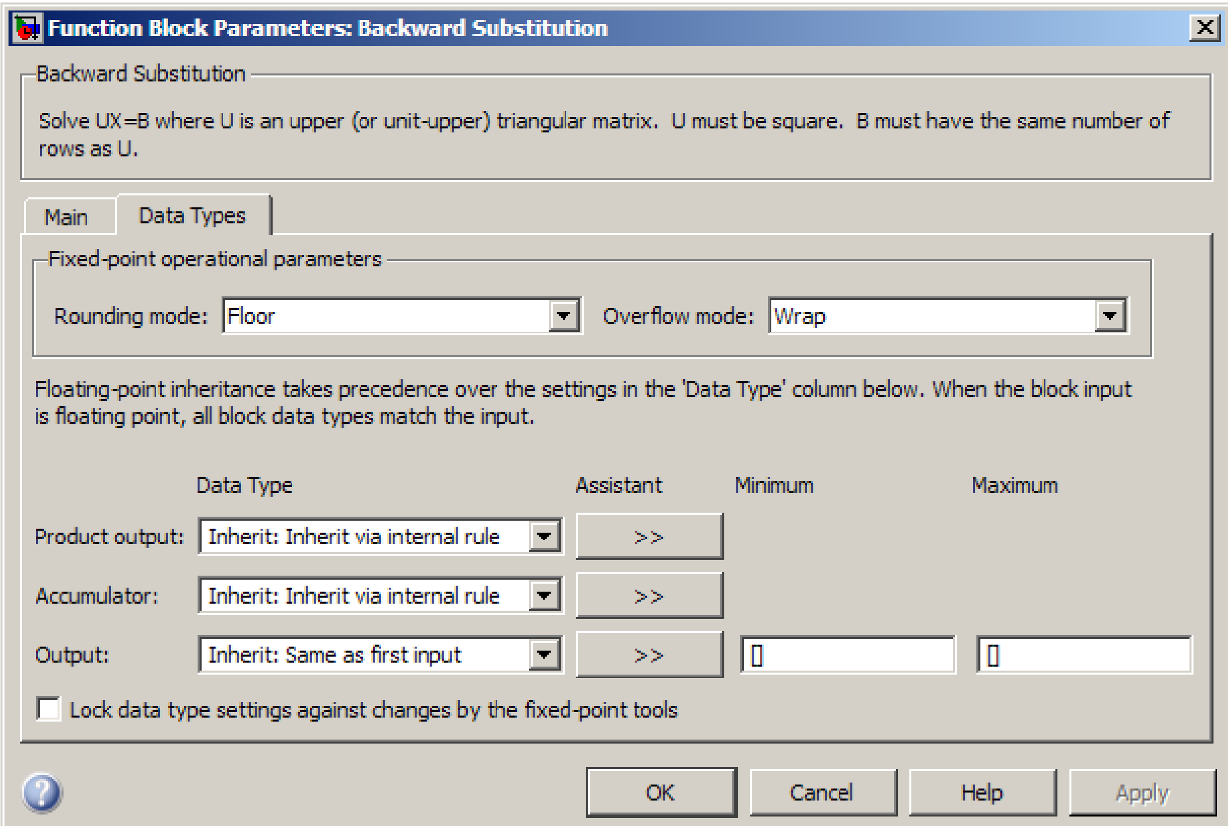
Do not select this check box if there are any elements on the diagonal of *U* that do not have a value of 1. When you clear the **Input U is unit-upper triangular** check box, the block always performs the necessary divide operation.

Diagonal of complex input U is real

Select to optimize simulation speed when the diagonal elements of complex input U are real. This parameter is only visible when **Input U is unit-upper triangular** is not selected.

Note: When U is a complex fixed-point signal, you must select either **Input U is unit-upper triangular** or **Diagonal of complex input U is real**. In such a case, any imaginary part of the diagonal of U is ignored.

The **Data Types** pane of the Backward Substitution block dialog appears as follows.



Note: Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-112 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-112 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-112 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as first input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Minimum

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
U	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
B	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
X	• Double-precision floating point • Single-precision floating point • Fixed point (signed only)

Port	Supported Data Types
	• 8-, 16-, and 32-bit signed integers

See Also

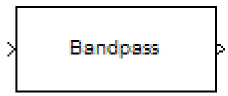
`Cholesky Solver`	DSP System Toolbox
`Forward Substitution`	DSP System Toolbox
`LDL Solver`	DSP System Toolbox
`Levinson-Durbin`	DSP System Toolbox
`LU Solver`	DSP System Toolbox
`QR Solver`	DSP System Toolbox

See “Linear System Solvers” for related information.

Introduced before R2006a

Bandpass Filter

Design bandpass filter



Library

Filtering / Filter Designs

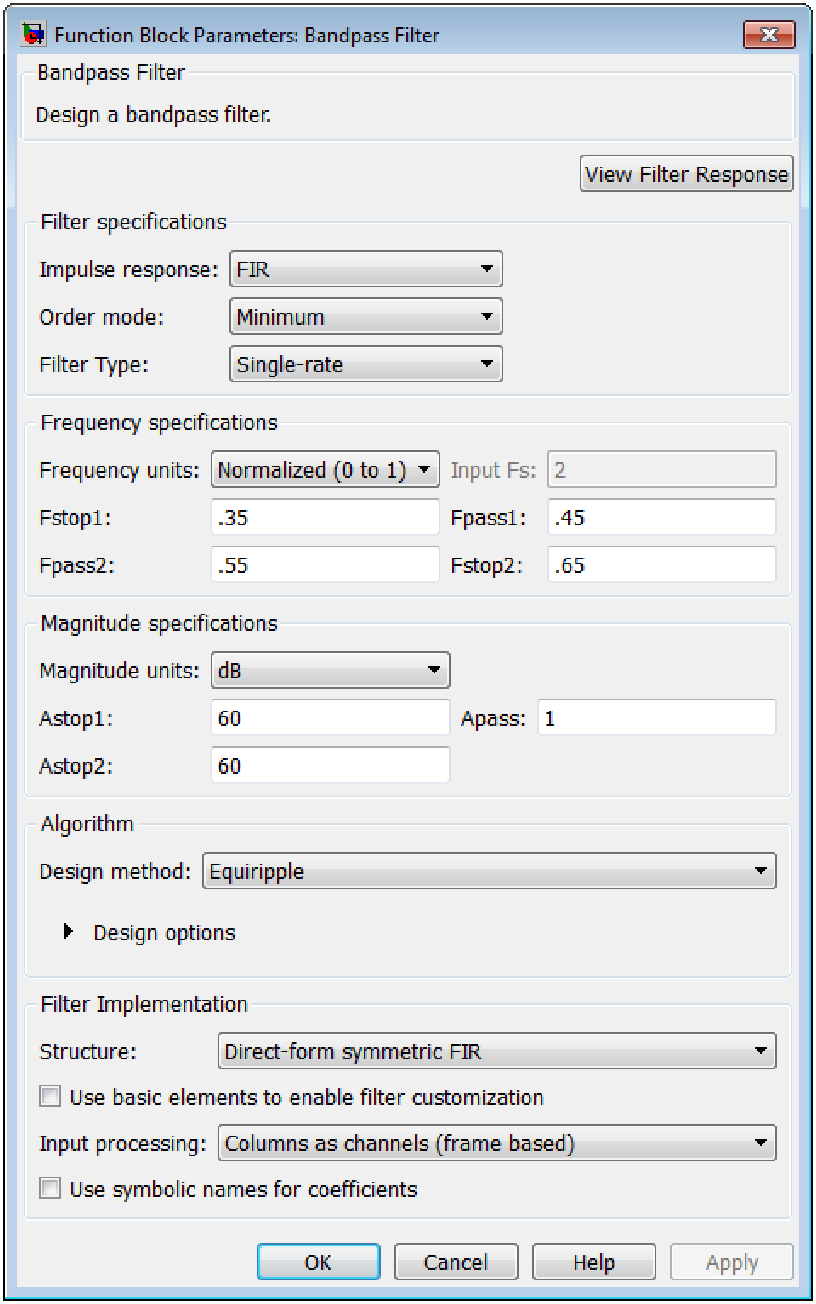
`dspfdesign`

Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See “Bandpass Filter Design — Main Pane” on page 5-716 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.


Function Block Parameters: Bandpass Filter
Bandpass Filter
Design a bandpass filter.
View Filter Response
Filter specifications
Impulse response: FIR
Order mode: Minimum
Filter Type: Single-rate
Frequency specifications
Frequency units: Normalized (0 to 1)
Input Fs: 2
Fstop1: .35
Fpass1: .45
Fpass2: .55
Fstop2: .65
Magnitude specifications
Magnitude units: dB
Astop1: 60
Apass: 1
Astop2: 60
Algorithm
Design method: Equiripple
Design options
Filter Implementation
Structure: Direct-form symmetric FIR
Use basic elements to enable filter customization
Input processing: Columns as channels (frame based)
Use symbolic names for coefficients
OK
Cancel
Help
Apply

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop-down list. `FIR` is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note: The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select `Minimum` (the default) or `Specify`. Selecting `Specify` enables the **Order** option so you can enter the filter order. When you set the **Impulse response** to `IIR`, you can specify different numerator and denominator orders. To specify a different denominator order, you must select the **Denominator order** check box.

Order

Enter the filter order. This option is enabled only if you set the **Order mode** to `Specify`.

Denominator order

Select this check box to specify a different denominator order. This option is enabled only if you set the **Impulse response** to `IIR` and the **Order mode** to `Specify`.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

Decimation Factor

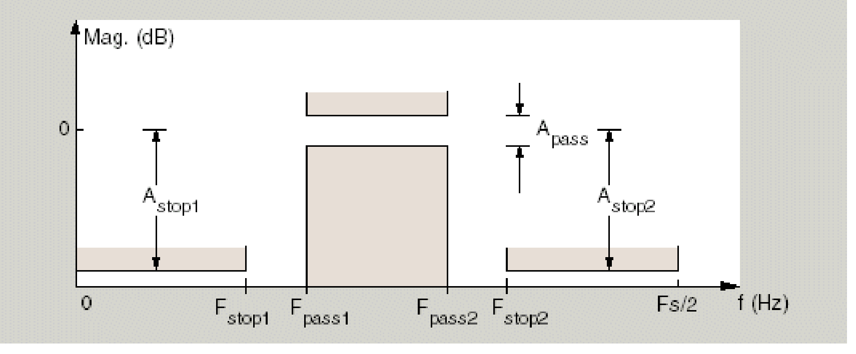
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, regions between specification values such as Fstop1 and Fpass1 represent transition regions where the filter response is not constrained.

Frequency constraints

When **Order mode** is `Specify`, select the filter features that the block uses to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- `Passband edges` — For IIR filters, define the filter by specifying frequencies for the edges of the passband.
- `Stopband edges` — For IIR filters, define the filter by specifying frequencies for the edges of the stopbands.
- `3 dB points` — For IIR filters, define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point three decibels below the passband value.
- `3 dB points and passband width` — For IIR filters, define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.
- `3 dB points and stopband widths` — For IIR filters, define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.
- `6 dB points` — For FIR filters, define the filter response by specifying the locations of the 6 dB points. The 6 dB point is the frequency for the point six decibels below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fstop1

Enter the frequency at the edge of the end of the first stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fpass1

Enter the frequency at the edge of the start of the passband. Specify the value in either normalized frequency units or the absolute units you selected for **Frequency units**.

Fpass2

Enter the frequency at the edge of the end of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop2

Enter the frequency at the edge of the start of the second stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

F3dB1

When **Frequency constraints** is `3 dB points`, `3 dB points and passband width`, or `3 dB points and stopband width`, specify the lower-frequency 3 dB point.

F3dB2

When **Frequency constraints** is `3 dB points`, `3 dB points and passband width`, or `3 dB points and stopband width`, specify the higher-frequency 3 dB point.

F6dB1

When **Frequency constraints** is `6 dB points`, specify the lower-frequency 6 dB point.

F6dB2

When **Frequency constraints** is `6 dB points`, specify the higher-frequency 6 dB point.

Passband width

When **Frequency constraints** is `3 dB points and passband width`, specify the width of the passband, in units corresponding to the **Frequency units** parameter.

Stopband width

When **Frequency constraints** is `3 dB points and stopband width`, specify the width of the stopband, in units corresponding to the **Frequency units** parameter.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude constraints

This option is only available when you specify the order of your filter design. The options for **Magnitude constraints** depend on the value of the **Frequency constraints**. When you set the **Frequency constraints** parameter to `Unconstrained`, **Magnitude constraints** must also be set to `Unconstrained`. When **Frequency constraints** is not set to `Unconstrained`, some combination of the following options will be available for the **Magnitude constraints** parameter: `Unconstrained`, `Passband ripple`, `Passband ripple and stopband attenuation` or `Stopband attenuation`.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications.

From the drop-down list, select one of the following options:

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in dB (decibels). This is the default setting.
- `Squared` — Specify the magnitude in squared units.

Astop1

Enter the filter attenuation in the first stopband in the units you choose for **Magnitude units**, either linear or decibels.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop2

Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is `Equiripple`.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Specify the phase constraint of the filter as `Linear`, `Maximum`, or `Minimum`.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note: Generally, **Minimum order** designs are not available for IIR filters.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2006b

Bandstop Filter

Design bandstop filter

Library

Filtering / Filter Designs

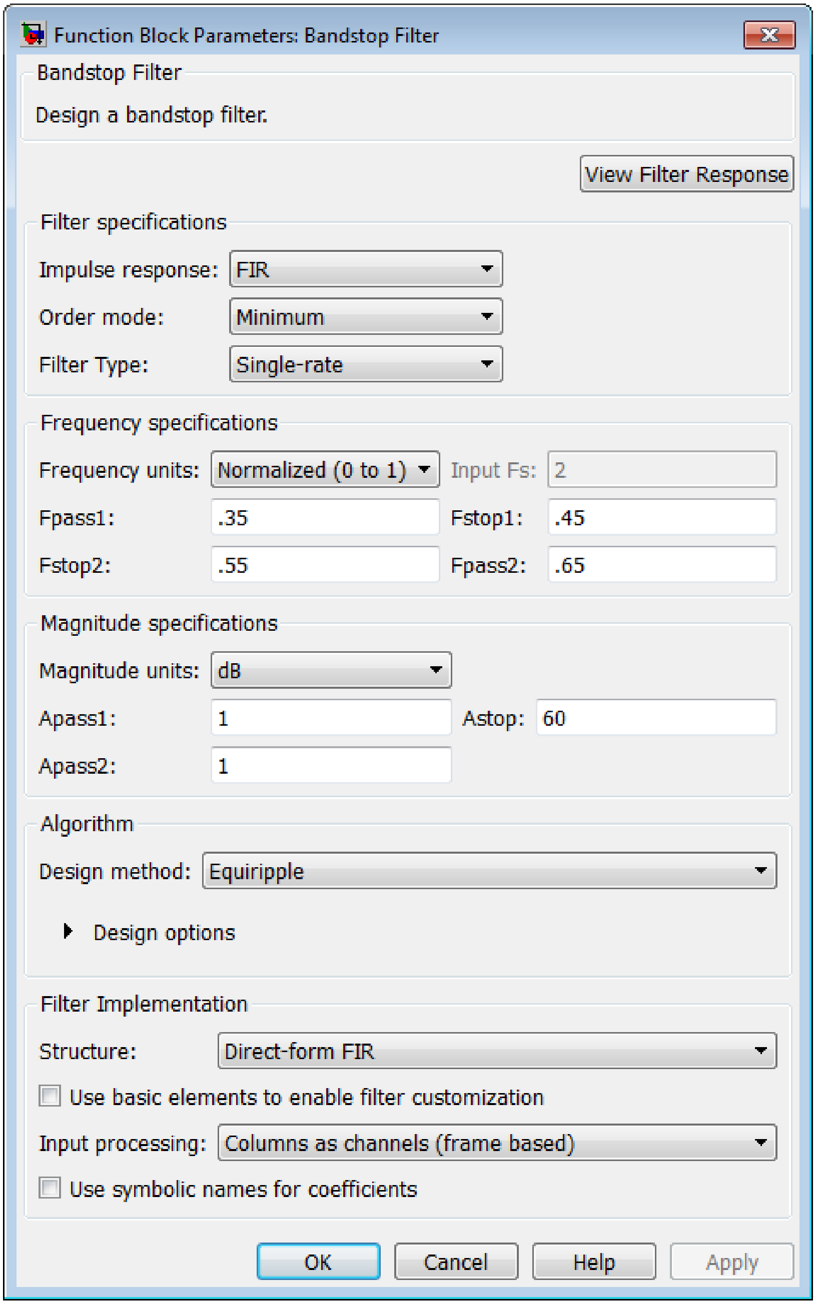
`dspfdesign`

Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See “Bandstop Filter Design — Main Pane” on page 5-723 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.


Function Block Parameters: Bandstop Filter
Bandstop Filter
Design a bandstop filter.
View Filter Response
Filter specifications
Impulse response: FIR
Order mode: Minimum
Filter Type: Single-rate
Frequency specifications
Frequency units: Normalized (0 to 1)
Input Fs: 2
Fpass1: .35
Fstop1: .45
Fstop2: .55
Fpass2: .65
Magnitude specifications
Magnitude units: dB
Apass1: 1
Astop: 60
Apass2: 1
Algorithm
Design method: Equiripple
Design options
Filter Implementation
Structure: Direct-form FIR
Use basic elements to enable filter customization
Input processing: Columns as channels (frame based)
Use symbolic names for coefficients
OK
Cancel
Help
Apply

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop-down list. `FIR` is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note: The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select `Minimum` (the default) or `Specify`. Selecting `Specify` enables the **Order** option so you can enter the filter order. When you set the **Impulse response** to `IIR`, you can specify different numerator and denominator orders. To specify a different denominator order, you must select the **Denominator order** check box.

Order

Enter the filter order. This option is enabled only if you set the **Order mode** to `Specify`.

Denominator order

Select this check box to specify a different denominator order. This option is enabled only if you set the **Impulse response** to `IIR` and the **Order mode** to `Specify`.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

Decimation Factor

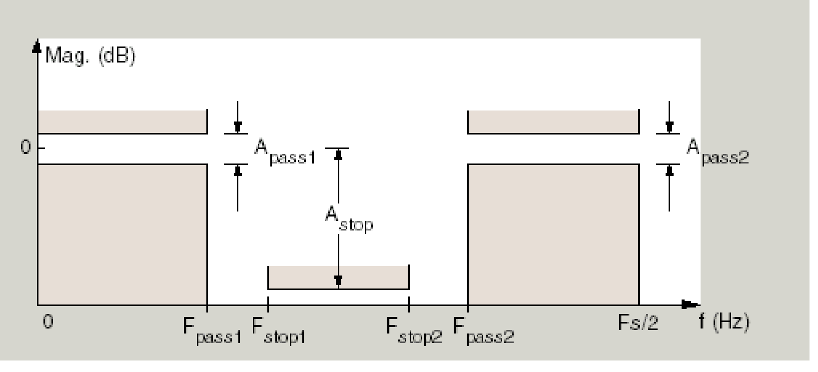
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



Frequency constraints

When **Order mode** is `Specify`, select the filter features that the block uses to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- `Passband edges` — For IIR filters, define the filter by specifying frequencies for the edges of the passband.
- `Stopband edges` — For IIR filters, define the filter by specifying frequencies for the edges of the stopbands.
- `3 dB points` — For IIR filters, define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point three decibels below the passband value.
- `3 dB points and passband width` — For IIR filters, define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.
- `3 dB points and stopband widths` — For IIR filters, define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.
- `6 dB points` — For FIR filters, define the filter response by specifying the locations of the 6 dB points. The 6 dB point is the frequency for the point six decibels below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

When you design an interpolator, Fs represents the sampling frequency at the filter output rather than the filter input. This option is available only when you set **Filter type** is `interpolator`.

Fpass1

Enter the frequency at the edge of the end of the first passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop1

Enter the frequency at the edge of the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop2

Enter the frequency at the edge of the end of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fpass2

Enter the frequency at the edge of the start of the second passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

F3dB1

When **Frequency constraints** is `3 dB points`, `3 dB points and passband width`, or `3 dB points and stopband width`, specify the lower-frequency 3 dB point.

F3dB2

When **Frequency constraints** is `3 dB points`, `3 dB points and passband width`, or `3 dB points and stopband width`, specify the higher-frequency 3 dB point.

F6dB1

When **Frequency constraints** is `6 dB points`, specify the lower-frequency 6 dB point.

F6dB2

When **Frequency constraints** is `6 dB points`, specify the higher-frequency 6 dB point.

Passband width

When **Frequency constraints** is `3 dB points and passband width`, specify the width of the passband, in units corresponding to the **Frequency units** parameter.

Stopband width

When **Frequency constraints** is `3 dB points and stopband width`, specify the width of the stopband, in units corresponding to the **Frequency units** parameter.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude constraints

This option is only available when you specify the order of your filter design. The options for **Magnitude constraints** depend on the value of the **Frequency constraints**. Depending on the value of the **Frequency constraints** parameter, some combination of the following options will be available for the **Magnitude constraints** parameter: `Unconstrained`, `Passband ripple and stopband attenuation`, or `Constrained bands`.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Apass1

Enter the filter ripple allowed in the first passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels

Apass2

Enter the filter ripple allowed in the second passband in the units you choose for **Magnitude units**, either linear or decibels

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is `Equiripple`.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Specify the phase constraint of the filter as `Linear`, `Maximum`, or `Minimum`.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2006b

Binary File Reader

Read data from binary files
Library: DSP System Toolbox / Sources

Description

The Binary File Reader block reads multichannel signal data from a binary file. The block reads the header that precedes the data. The **File header** parameter specifies the structure of the header. You can specify the type, size, and complexity of the data through the block parameters. You can also export the header to the base workspace by clicking on the **Export header to base workspace** button.

The first time you read the file, the reader reads the header, followed by the data. On subsequent calls, the reader reads the remaining data. Once the end of the file is reached, the reader returns zeros of the specified data type, size, and complexity. The reader can read signal data from a binary file that is not created by the `Binary File Writer` block.

Ports

Output

`data` — Binary file data

column vector | row vector | matrix

The reader block reads the binary data from the file specified in the **File name** parameter. The data output from the block has dimensions **Samples per frame**-by-**Number of channels**. The block can read floating-point data and integer data. The input data can be real or complex. When the data is complex, the block reads the data as interleaved real and imaginary components. The reader assumes the default endianness of the host machine.

This port is unnamed until you select the **Output end-of-file indicator** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

`EOF` — End-of-file indicator
boolean scalar

When the block reaches the end of the file, the port outputs a `1`. Otherwise, the port outputs a `0`.

This port is unnamed until you select the **Output end-of-file indicator** parameter.

Data Types: `Boolean`

Parameters

`File name` — Name of the file
`'Untitled.bin'` (default) | character vector

Name of the file from which the block reads the data. If the file is not on the MATLAB path, then specify the full path for the file.

`File header` — Size of the header
`struct([])` (default) | structure

The structure specifies the prototype of the file header, that is, the size of the header and the data type of the field values. The structure can have an arbitrary number of fields. Each field of the structure must be a real matrix of a built-in type. For example, if **File header** is set to `struct('field1',1:10,'field2',single(1))`, the block assumes that the header is formed by 10 real double-precision values followed by 1 single-precision value. If the file contains no header, you can set this parameter to an empty structure, `struct([])`.

`Export header to base workspace` — Retrieve the header
button

To retrieve the file header, click **Export header to base workspace**. The block exports the file header to the base workspace.

`Storage data type` — Storage class of data in file
`'double'` (default) | `'single'` | `'int8'` | `'int16'` | `'int32'` | `'int64'` | `'uint8'` | `'uint16'` | `'uint32'` | `'uint64'`

Storage class of data in file, specified as a character vector. This parameter defines the data type of the matrix the block outputs.

`Samples per frame` — Number of samples per output frame
`1024` (default) | positive integer

Samples per frame specifies the number of rows of the output matrix that the block outputs. The output matrix has dimensions **Samples per frame**-by-**Number of channels**. Once the end of the file is reached, the block returns zeros of the specified data type, size, and complexity.

`Data is complex` — Specify data complexity
off (default) | on

When you select this parameter, the reader treats the data as complex data. The block reads the data as interleaved real and imaginary components. Configure the block to read the data as a 2-by-2 matrix. The block reads `[1 5 2 6 3 7 4 8]` as `[1 2; 3 4]+1j*[5 6; 7 8]`. When you do not select this parameter, the block reads the data as `[1 5; 2 6]`.

`Number of channels` — Number of channels
`1` (default) | positive integer

Number of channels specifies the number of columns of the output matrix that the block outputs. This parameter defines the number of consecutive interleaved data samples stored in the file for each time instant. The size of the data is **Samples per frame**-by-**Number of channels**. Once the end of the file is reached, if the output matrix is not full, the block fills the matrix with zeros to make it a full-sized matrix.

`Output end-of-file indicator` — End-of-file indicator
off (default) | on

When you select this parameter, an additional output port named **EOF** appears on the block. When the block reaches the end of the file, the port outputs a `1`. Otherwise, the port outputs a `0`.

`Sample time (s)` — Sample time
1 (default) | numeric and nonnegative or -1

Sample time (s) controls the sample time at the output port of the block. This value represents $1/F_s$, where F_s is the sampling rate of the signal data. The Simulink sample time at the output port is **Samples per frame** × **Sample time (s)**.

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Blocks
`Audio Device Writer` | `Binary File Writer`

System Objects
dsp.BinaryFileWriter | dsp.BinaryFileReader | dsp.MatFileWriter

Introduced in R2016b

Binary File Writer

Write data to binary files

Library: DSP System Toolbox / Sources

Description

The Binary File Writer block writes multichannel signal data to a binary file. The block specifies the name of the file and the structure of the header that precedes the signal data. If there is no header to write, the block specifies an empty structure, `struct([])`. The first time you write to the file, the block writes the header, followed by the data. On subsequent calls, the block writes the remaining data. If the header is empty, then no header is written.

The block writes the data in a row-major format. For example, if the input array is [`1 2 4 5; 8 7 9 2`], the block writes the data as [`1 2 4 5 8 7 9 2`].

Ports

Input

`Port_1` — Data to write

column vector | row vector | matrix

The writer block writes the data to the file specified in the **File name** parameter. If the **File header** structure is not empty, then the writer writes the header before writing the data. The block can write floating-point data and integer data. The input data can be real or complex. When the data is complex, the block writes the data as interleaved real and imaginary components. The writer assumes the default endianness of the host machine.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Complex Number Support: Yes

Parameters

`File name` — Name of the file

`'Untitled.bin'` (default) | character vector

Name of the file to which the block writes the data.

`File header` — Size of the header

`struct([])` (default) | structure

The structure can have an arbitrary number of fields. Each field of the structure must be a real matrix of a built-in type. For example, if **File header** is set to `struct('field1',1:10,'field2',single(1))`, the block writes a header formed by 10 double-precision values, `(1:10)`, followed by 1 single precision value, `single(1)`. If there is no header to write, set this parameter to an empty structure, `struct([])`.

`Simulate using` — Type of simulation to run

`Code generation` (default) | `Interpreted execution`

- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Blocks
`Audio Device Writer` | `Binary File Reader`

System Objects
dsp.BinaryFileReader | dsp.BinaryFileWriter | dsp.MatFileReader

Introduced in R2016b

Biquad Filter

Model biquadratic IIR (SOS) filters

Library

Filtering / Filter Implementations

`dsparch4`

Description

The Biquad Filter block independently filters each channel of the input signal with the specified biquadratic IIR filter. When you specify the filter coefficients in the dialog box, the block implements static filters with fixed coefficients. When you provide the filter coefficients through an input port, you can tune the coefficients during simulation.

The block filters an M-by-N input matrix as follows:

- When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each column as a separate channel. In this mode, the block creates M instances of the same filter, each with its own independent state buffer. Each of the M filters process N input samples at every Simulink time step.
- When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats each element as a separate channel. In this mode, the block creates $M*N$ instances of the same filter, each with its own independent state buffer. Each filter processes one input sample at every Simulink time step.

This block supports variable-size input. It means that while the block is simulating, the frame size (number of rows) can change. The output dimensions always equal the dimensions of the input signal. The outputs of this block numerically match the outputs of the dsp.BiquadFilter System object™.

The Biquad Filter block supports the Simulink state logging feature. See “States” (Simulink) for more information.

Coefficient Source and Filter Structures

The Biquad Filter block can operate in three different modes. Select the mode in the **Coefficient source** group box. If you select:

- **Dialog parameters** — Enter information about the filter such as structure and coefficients in the block mask. In this mode, you can choose the following filter structures in the **Filter structure** parameter:
  - `Direct form I`
  - `Direct form I transposed`
  - `Direct form II`
  - `Direct form II transposed`
- **Input port(s)** — Enter information about the filter structure in the block mask using the **Filter structure** parameter. The filter coefficients come into the block via input ports. The following additional ports appear on the block icon:
  - `Num` — Specify numerator coefficients
  - `Den` — Specify denominator coefficients
  - `g` — Specify scale values

  `Num` must be a 3–by–N numeric matrix, `Den` must be a 2–by–N numeric matrix, and `g` must be a 1–by–(N+1) numeric vector, where N is the number of biquad filter sections. The object assumes the first denominator coefficients of each section to be 1. This configuration is applicable when the `SOSMatrixSource` property is 'Input port' and the `ScaleValuesInputPort` property is true. The reason you would need to specify `Num` and `Den` instead of the `SOSMatrix`, is that in Fixed-Point operation, the numerators, and denominators can have different fraction lengths. Therefore, there is a need to be able to pass the data of the numerator with a fixed-point type different from that of the denominator.
- **Filter object** — Specify the filter using a dsp.BiquadFilter System object.

Specifying the SOS Matrix and Scale Values

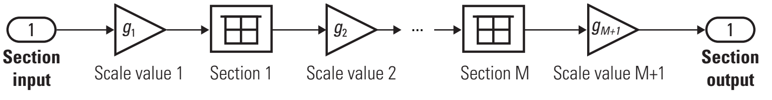
The **SOS matrix (Mx6)** is an M-by-6 matrix, where M is the number of sections in the second-order section filter. Each row of the SOS matrix contains the numerator and denominator coefficients (b_{ik} and a_{ik}) of the corresponding section in the filter.

$$\begin{bmatrix} b_{01} & b_{11} & b_{21} & a_{01} & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & a_{02} & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0M} & b_{1M} & b_{2M} & a_{0M} & a_{1M} & a_{2M} \end{bmatrix}$$

`ss2sos` and `tf2sos` functions from Signal Processing Toolbox convert a state-space or transfer function description of your filter into the second-order section description used by this block.

The **Scale values** parameter specifies the scale values the block uses between each SOS section. You can specify a real-valued scalar or a vector of length M+**1**.

- If you enter a scalar, the value specifies the gain value before the first section of the second-order filter. The rest of the gain values default to `1`.
- If you enter a vector of M+1 values, each value specifies a separate section of the filter. For example, the first element is the first gain value, the second element is the second gain value, and so on.



Select the **Optimize unity scale values** check box to optimize your simulation when one or more scale values equal 1. Selecting this option removes the unity gains so that the values are treated like Simulink lines or wires. In some fixed-point cases, when there are unity scale values, selecting this parameter also omits certain casts. Refer to “Filter Structure Diagrams” on page 2-170 for more information.

Specifying Initial Conditions

The Biquad Filter block initializes the internal filter states to zero by default. Optionally, use the **Initial conditions** or **Initial conditions on zeros side** and **Initial conditions on poles side** parameters to specify nonzero initial states for the filter delays.

To determine the number of initial conditions you must specify and how to specify them, see the following table on valid initial conditions.

Valid Initial Conditions

Initial Condition	Description
Scalar	The block initializes all delay elements in the filter to the scalar value.
Vector or matrix (for applying different delay elements to each channel)	Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel. Where *M* is the number of sections and *N* is the number of input channels: • The vector length must equal the number of delay elements in the filter, *M**2. • The matrix must have the same number of rows as the number of delay elements in the filter (*M**2)**N*. The matrix must also have one column for each channel of the input signal.

Fixed-Point Data Types

See the “Filter Structure Diagrams” on page 2-170 section for diagrams showing the data types the biquad filter block uses when processing fixed-point signals.

Examples

Open an example model by typing `ex_biquad_filter_ref` at the MATLAB command line.

Dialog Box

Coefficient Source

The Biquad Filter block can operate in three different modes. Select the mode in the **Coefficient source** group box. If you select:

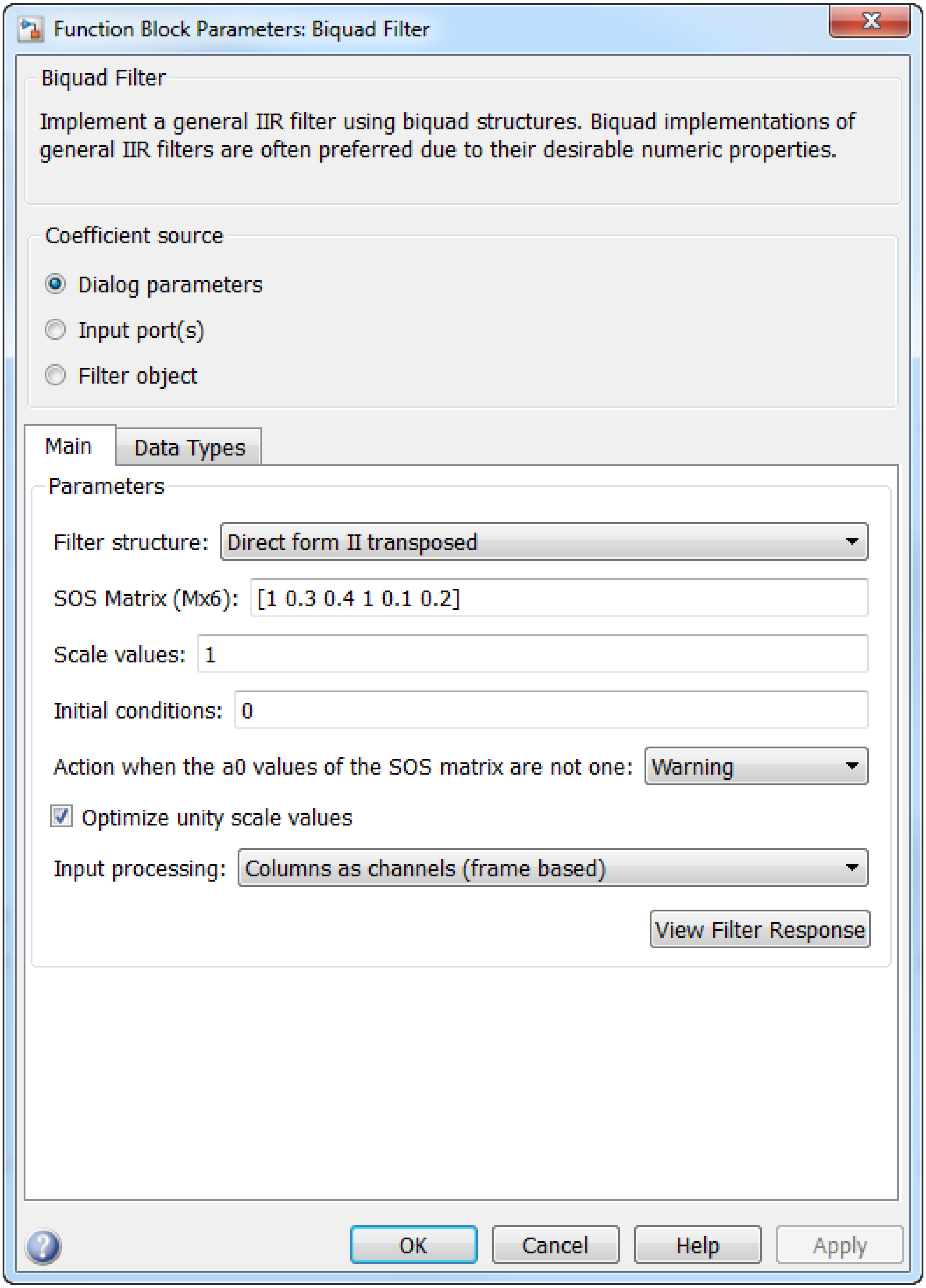
- **Dialog parameters** — Enter information about the filter such as structure and coefficients in the block mask. In this mode, you can choose the following filter structures in the **Filter structure** parameter:
  - `Direct form I`

  - `Direct form I transposed`
  - `Direct form II`
  - `Direct form II transposed`
- **Input port(s)** — Enter information about the filter structure in the block mask using the **Filter structure** parameter. The filter coefficients come into the block via input ports. The following additional ports appear on the block icon:
  - `Num` — Specify numerator coefficients
  - `Den` — Specify denominator coefficients
  - `g` — Specify scale values

  `Num` must be a 3–by–N numeric matrix, `Den` must be a 2–by–N numeric matrix, and `g` must be a 1–by–(N+1) numeric vector, where N is the number of biquad filter sections. The object assumes the first denominator coefficients of each section to be 1. This configuration is applicable when the `SOSMatrixSource` property is 'Input port' and the `ScaleValuesInputPort` property is true. The reason you would need to specify `Num` and `Den` instead of the `SOSMatrix`, is that in Fixed-Point operation, the numerators, and denominators can have different fraction lengths. Therefore, there is a need to be able to pass the data of the numerator with a fixed-point type different from that of the denominator.
- **Filter object** — Specify the filter using a dsp.BiquadFilter System object.

Specify Filter Characteristics in Dialog

The **Main** pane of the Biquad Filter block dialog appears as follows when **Dialog parameters** is selected in the **Coefficient source** group box.



Function Block Parameters: Biquad Filter
Biquad Filter
Implement a general IIR filter using biquad structures. Biquad implementations of general IIR filters are often preferred due to their desirable numeric properties.
Coefficient source
Dialog parameters
Input port(s)
Filter object
Main
Data Types
Parameters
Filter structure:
Direct form II transposed
SOS Matrix (Mx6):
[1 0.3 0.4 1 0.1 0.2]
Scale values:
1
Initial conditions:
0
Action when the a0 values of the SOS matrix are not one:
Warning
Optimize unity scale values
Input processing:
Columns as channels (frame based)
View Filter Response
OK
Cancel
Help
Apply

Filter structure

Select the filter structure.

This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected.

SOS Matrix

Specify an M-by-6 matrix, where M is the number of sections in the second-order section filter. Each row of the SOS matrix contains the numerator and denominator coefficients (b_{ik} and a_{ik}) of the corresponding section in the filter.

$$\begin{bmatrix} b_{01} & b_{11} & b_{21} & a_{01} & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & a_{02} & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0M} & b_{1M} & b_{2M} & a_{0M} & a_{1M} & a_{2M} \end{bmatrix}$$

This parameter is only visible when **Dialog parameters** is selected.

Scale values

The **Scale values** parameter specifies the scalar or vector of M+1 scale values to be used between SOS sections.

- When you enter a scalar, the value specifies the gain value before the first section of the second-order filter. The rest of the gain values default to 1.
- When you enter a vector of M+1 values, each value specifies a separate section of the filter. For example, the first element is the first gain value, the second element is the second gain value, and so on.

  This parameter is only visible when **Dialog parameters** is selected.

Initial conditions

Specify the initial conditions of the filter states. To learn how to specify initial conditions, see “Specifying Initial Conditions” on page 2-149.

This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected and the filter structure is `Direct form II` or `Direct form II transposed`.

Initial conditions on zeros side

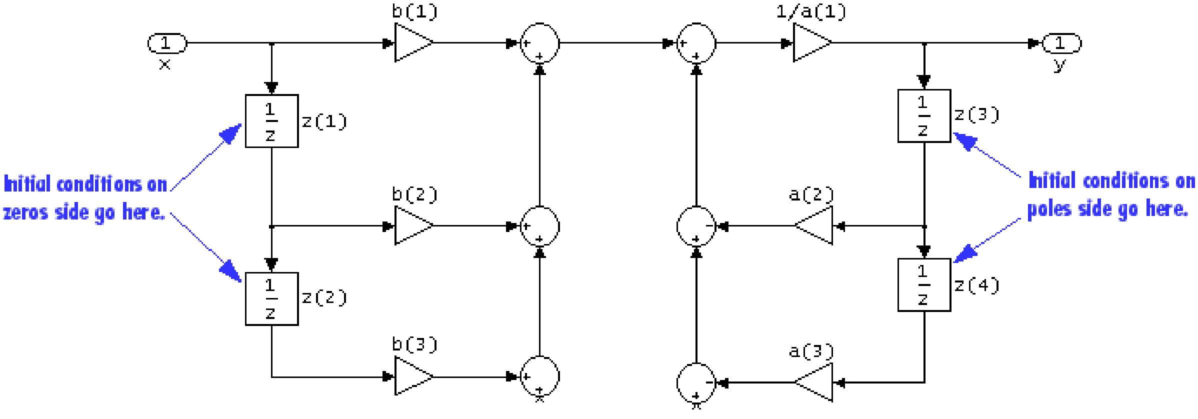
Specify the initial conditions for the filter states on the side of the filter structure with the zeros (b_0, b_1,b_2, ...); see the next diagram.

This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected and the filter structure is `Direct form I` or `Direct form I transposed`. To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 2-149.

Initial conditions on poles side

Specify the initial conditions for the filter states on the side of the filter structure with the poles (a_0, a_1,a_2, ...); see the next diagram.

This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected and the filter structure is `Direct form I` or `Direct form I transposed`. To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 2-149.



Action when the a0 values of the SOS matrix are not one

Specify the action the block should perform when the SOS matrix a_{0j} values do not equal one; `None`, `Error`, or `Warn`.

This parameter is only visible when **Dialog parameters** is selected.

Optimize unity scale values

Select this check box to optimize your simulation when one or more scale values equal 1. Selecting this option removes the unity gains so that the values are treated like Simulink lines or wires. In some fixed-point cases when there are unity scale values, selecting this parameter also omits certain casts. Refer to "Filter Structure Diagrams" on page 2-170 for more information.

This parameter is only visible when **Dialog parameters** is selected.

Input processing

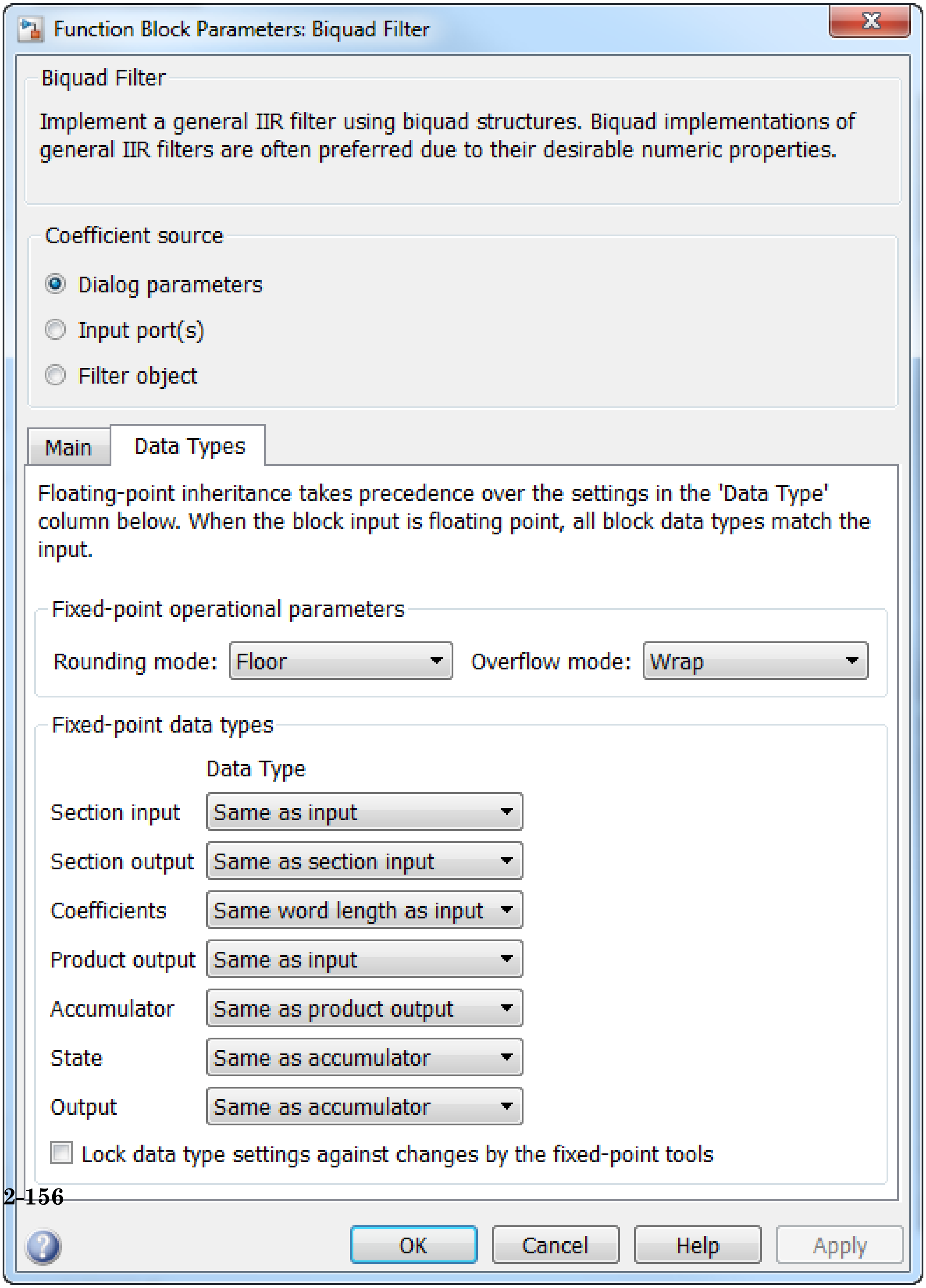
Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the filter specified in the dialog. For more information on FVTool, see the Signal Processing Toolbox documentation.

The **Data Types** pane of the Biquad Filter block dialog appears as follows when you select **Dialog parameters** in the **Coefficient source** group box.


Function Block Parameters: Biquad Filter
Biquad Filter
Implement a general IIR filter using biquad structures. Biquad implementations of general IIR filters are often preferred due to their desirable numeric properties.
Coefficient source
Dialog parameters
Input port(s)
Filter object
Main
Data Types
Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input.
Fixed-point operational parameters
Rounding mode: Floor
Overflow mode: Wrap
Fixed-point data types
Data Type
Section input
Same as input
Section output
Same as section input
Coefficients
Same word length as input
Product output
Same as input
Accumulator
Same as product output
State
Same as accumulator
Output
Same as accumulator
Lock data type settings against changes by the fixed-point tools
OK
Cancel
Help
Apply

Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; instead, they always round to `Nearest`.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; instead, they are always saturated.

Multiplicand

Choose how you specify the word and fraction lengths of the multiplicand data type of a `Direct form I transposed` filter structure. See "Fixed-Point Data Types" on page 2-150 and the "Direct Form I Transposed" on page 2-173 filter structure diagram for illustrations depicting the use of the multiplicand data type in this block.

This parameter is only visible when the **Filter structure** parameter is set to `Direct form I transposed`. When you select:

- `Same as output` — Word length and fraction length characteristics of **Multiplicand** data type match with those of the output of the block.
- `Binary point scaling` — Enter the word length and the fraction length of the multiplicand, in bits.
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the multiplicand. This block requires power-of-two slope and a bias of zero.

Section input

Choose how you specify the word and fraction lengths of the fixed-point data type going into each section of a biquadratic filter. See "Fixed-Point Data Types" on page 2-150 for illustrations depicting the use of the section input data type in this block. When you select:

- `Same as input` — Word length and fraction length characteristics of **Section input** data type match with those of the input to the block.
- `Binary point scaling` — Enter the word and fraction lengths of the section input, in bits.
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the section input. This block requires power-of-two slope and a bias of zero.

Section output

Choose how you specify the word and fraction lengths of the fixed-point data type coming out of each section of a biquadratic filter. See "Fixed-Point Data Types" on page 2-150 for illustrations depicting the use of the section output data type in this block. When you select:

- `Same as section input` — Word length and fraction length characteristics of **Section output** data type match with those of the input to the block.
- `Binary point scaling` — Enter the word and fraction lengths of the section output, in bits.
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the section output. This block requires power-of-two slope and a bias of zero.

Coefficients

Choose how you specify the word and fraction lengths of the filter coefficients (numerator, denominator, and scale value) when **Dialog parameters** is selected in the **Coefficient source** group box. See “Fixed-Point Data Types” on page 2-150 for illustrations depicting the use of the coefficient data types in this block. When you select:

- `Same word length as input` — Word length of the filter coefficients matches with that of the input to the block. In this mode, the block automatically sets the fraction length of the coefficients to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- `Specify word length` — Enter the word length of the coefficients, in bits. In this mode, the block automatically sets the fraction length of the coefficients to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- `Binary point scaling` — Enter the word length and the fraction length of the coefficients, in bits. If applicable, you can enter separate fraction lengths for the numerator and denominator coefficients.
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the coefficients. If applicable, you can enter separate slopes for the numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.
- The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; instead, they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See “Fixed-Point Data Types” on page 2-150 and

"Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. When you select:

- `Same as input` (default) — **Product output** word length and fraction length characteristics match those of the input to the block.
- `Inherit via internal rule` — **Product output** word length and fraction lengths are computed based on full-precision rules. These rules prevent quantization from occurring within the block. Bits are added, as needed, so that no roundoff or overflow occurs. For more details, see "Inherit via Internal Rule".
- `Binary point scaling` — Enter the word length and the fraction length of the product output, in bits. If applicable, you can enter separate fraction lengths for the numerator and denominator product output data type
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the product output. If applicable, you can enter separate slopes for the numerator and denominator product output data type. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See "Fixed-Point Data Types" on page 2-150 and "Multiplication Data Types" for illustrations depicting the use of the accumulator data type in this block. When you select:

- `Same as input` — **Accumulator** word and fraction length characteristics match those of the input to the block.
- `Same as product output` — **Accumulator** word and fraction length characteristics match those of the product output.
- `Binary point scaling` — Enter the word length and the fraction length of the accumulator, in bits. If applicable, you can enter separate fraction lengths for the numerator and denominator accumulator data type.
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the accumulator. If applicable, you can enter separate slopes for the numerator and denominator accumulator data type. This block requires power-of-two slope and a bias of zero.

State

Use this parameter to specify how you would like to designate the state word and fraction lengths when **Dialog parameters** is selected in the **Coefficient source** group box. See "Fixed-Point Data Types" on page 2-150 for illustrations depicting the use of the state data type in this block.

This parameter is not visible for `Direct form I` and `Direct form I transposed filter` structures. When you select:

- `Same as input` — **State** word and fraction length characteristics match those of the input to the block.
- `Same as accumulator` — **State** word and fraction length characteristics match those of the accumulator.
- `Binary point scaling` — Enter the word length and the fraction length of the state, in bits. If applicable, you can enter separate fraction lengths for the numerator and denominator state data type.
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the state. If applicable, you can enter separate slopes for the numerator and denominator state data type. This block requires power-of-two slope and a bias of zero.

Output

Choose how you specify the output word length and fraction length. See "Fixed-Point Data Types" on page 2-150 for illustrations depicting the use of the output data type in this block. When you select:

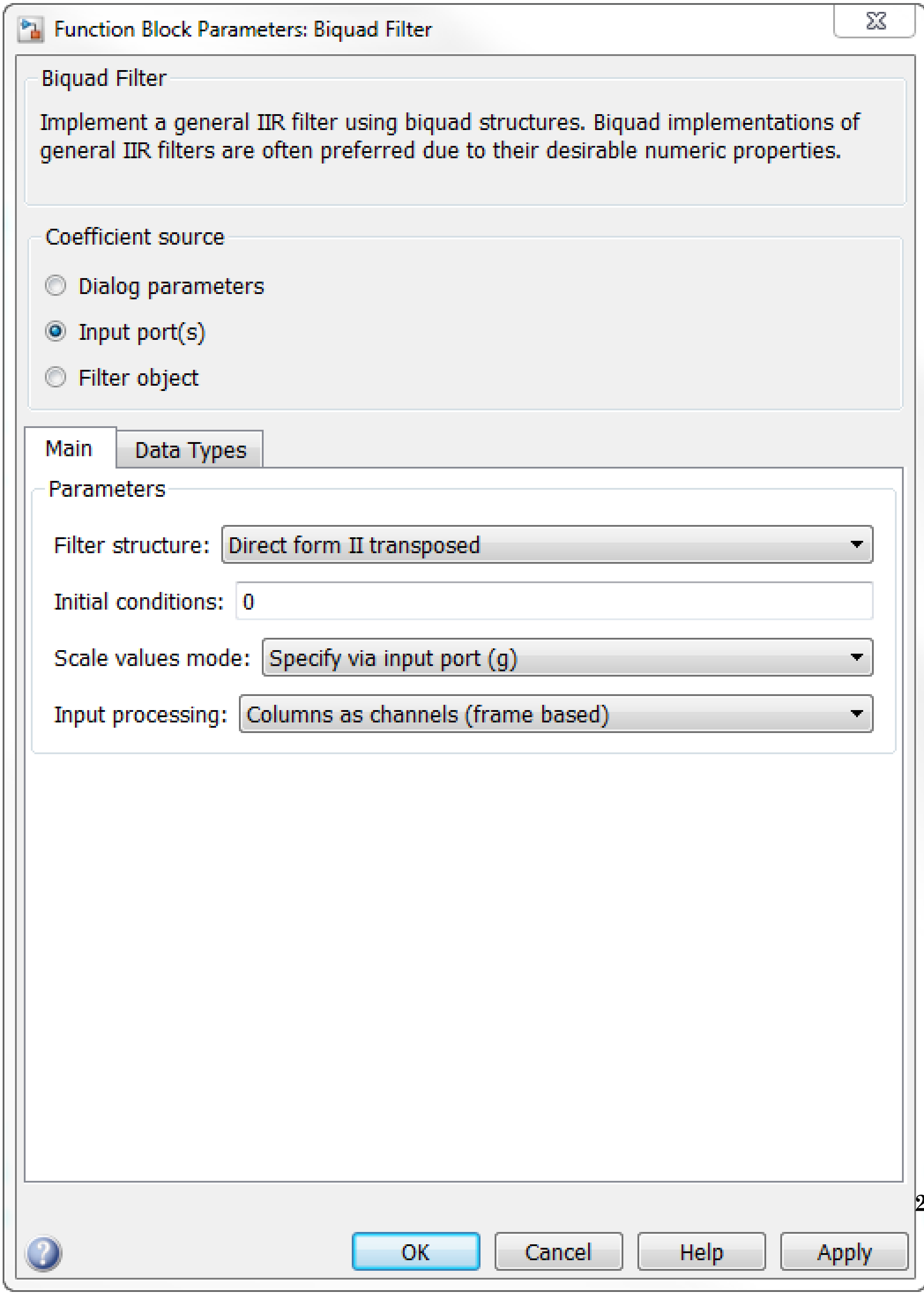
- `Same as input` — **Output** word and fraction length characteristics match those of the input to the block.
- `Same as accumulator` — **Output** word and fraction length characteristics match those of the accumulator.
- `Binary point scaling` — Enter the word length and the fraction length of the output, in bits.
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Specify Filter Characteristics Via Input Port

The **Main** pane of the Biquad Filter block dialog appears as follows when **Input port(s)** is selected in the **Coefficient source** group box.


Function Block Parameters: Biquad Filter
Biquad Filter
Implement a general IIR filter using biquad structures. Biquad implementations of general IIR filters are often preferred due to their desirable numeric properties.
Coefficient source
Dialog parameters
Input port(s)
Filter object
Main
Data Types
Parameters
Filter structure:
Direct form II transposed
Initial conditions:
0
Scale values mode:
Specify via input port (g)
Input processing:
Columns as channels (frame based)
OK
Cancel
Help
Apply

Filter structure

Select the filter structure.

This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected in the **Coefficient source** group box.

Initial conditions

Specify the initial conditions of the filter states. To learn how to specify initial conditions, see “Specifying Initial Conditions” on page 2-149.

This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected and the filter structure is `Direct form II` or `Direct form II transposed`.

Initial conditions on zeros side

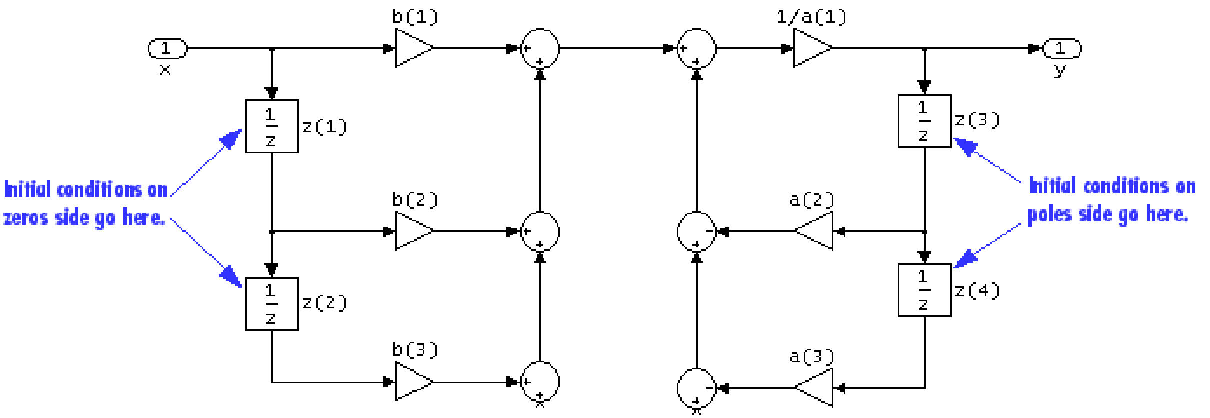
Specify the initial conditions for the filter states on the side of the filter structure with the zeros (b_0, b_1,b_2, ...); see the next diagram.

This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected and the filter structure is `Direct form I` or `Direct form I transposed`. To learn how to specify initial conditions, see “Specifying Initial Conditions” on page 2-149.

Initial conditions on poles side

Specify the initial conditions for the filter states on the side of the filter structure with the poles (a_0, a_1,a_2, ...); see the next diagram.

This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected and the filter structure is `Direct form I` or `Direct form I transposed`. To learn how to specify initial conditions, see “Specifying Initial Conditions” on page 2-149.



Scale values mode

Choose how to specify the scale values to use between filter sections. When you select `Specify via input port (g)`, you enter the scale values as a 2-D vector at port g. When you select `Assume all are unity and optimize`, all scale values are removed and treated like Simulink lines or wires.

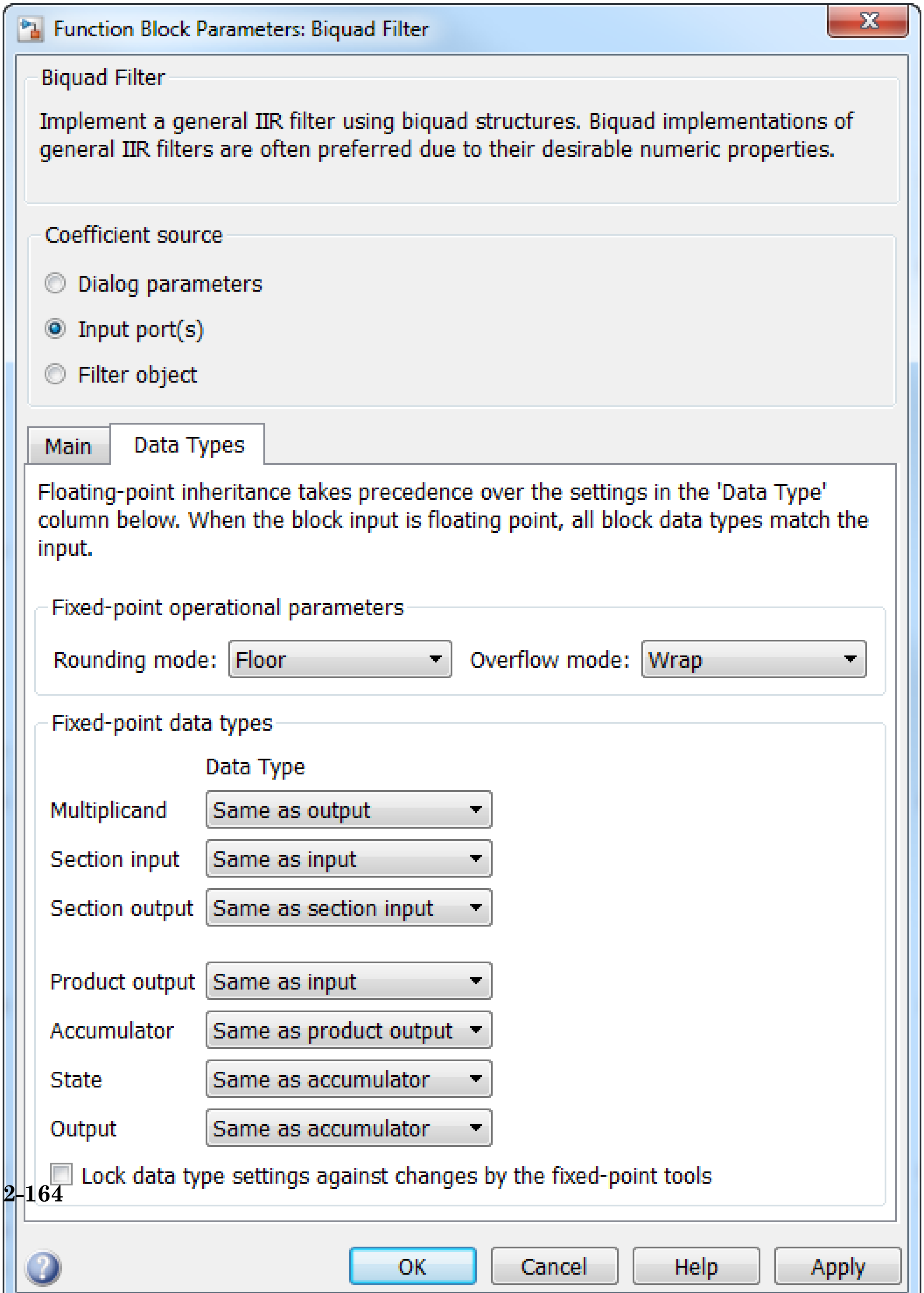
This parameter is only visible when **Input port(s)** is selected.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

The **Data Types** pane of the Biquad Filter block dialog appears as follows when you select **Input port(s)** in the **Coefficient source** group box.


Function Block Parameters: Biquad Filter
Biquad Filter
Implement a general IIR filter using biquad structures. Biquad implementations of general IIR filters are often preferred due to their desirable numeric properties.
Coefficient source
Dialog parameters
Input port(s)
Filter object
Main
Data Types
Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input.
Fixed-point operational parameters
Rounding mode:
Floor
Overflow mode:
Wrap
Fixed-point data types
Data Type
Multiplicand
Same as output
Section input
Same as input
Section output
Same as section input
Product output
Same as input
Accumulator
Same as product output
State
Same as accumulator
Output
Same as accumulator
Lock data type settings against changes by the fixed-point tools
OK
Cancel
Help
Apply

Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; instead, they always round to `Nearest`.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; instead, they are always saturated.

Multiplicand

Choose how you specify the word and fraction lengths of the multiplicand data type of a `Direct form I transposed` filter structure. See "Fixed-Point Data Types" on page 2-150 and the "Direct Form I Transposed" on page 2-173 filter structure diagram for illustrations depicting the use of the multiplicand data type in this block.

This parameter is only visible when the **Filter structure** parameter is set to `Direct form I transposed`. When you select:

- `Same as output` — Word length and fraction length characteristics of **Multiplicand** data type match with those of the output of the block.
- `Binary point scaling` — Enter the word length and the fraction length of the multiplicand, in bits.
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the multiplicand. This block requires power-of-two slope and a bias of zero.

Section input

Choose how you specify the word and fraction lengths of the fixed-point data type going into each section of a biquadratic filter. See "Fixed-Point Data Types" on page 2-150 for illustrations depicting the use of the section input data type in this block. When you select:

- `Same as input` — Word length and fraction length characteristics of **Section input** data type match with those of the input to the block.
- `Binary point scaling` — Enter the word and fraction lengths of the section input, in bits.
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the section input. This block requires power-of-two slope and a bias of zero.

Section output

Choose how you specify the word and fraction lengths of the fixed-point data type coming out of each section of a biquadratic filter. See "Fixed-Point Data Types" on page 2-150 for illustrations depicting the use of the section output data type in this block. When you select:

- `Same as section input` — Word length and fraction length characteristics of **Section output** data type match with those of the input to the block.
- `Binary point scaling` — Enter the word and fraction lengths of the section output, in bits.
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the section output. This block requires power-of-two slope and a bias of zero.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See “Fixed-Point Data Types” on page 2-150 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. When you select:

- `Same as input` (default) — **Product output** word length and fraction length characteristics match those of the input to the block.
- `Inherit via internal rule` — **Product output** word length and fraction lengths are computed based on full-precision rules. These rules prevent quantization from occurring within the block. Bits are added, as needed, so that no roundoff or overflow occurs. For more details, see “Inherit via Internal Rule”.
- `Binary point scaling` — Enter the word length and the fraction length of the product output, in bits. If applicable, you can enter separate fraction lengths for the numerator and denominator product output data type.
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the product output. If applicable, you can enter separate slopes for the numerator and denominator product output data type. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See “Fixed-Point Data Types” on page 2-150 and “Multiplication Data Types” for illustrations depicting the use of the accumulator data type in this block. When you select:

- `Same as input` — **Accumulator** word and fraction length characteristics match those of the input to the block.

- `Same as product output` — **Accumulator** word and fraction length characteristics match those of the product output.
- `Binary point scaling` — Enter the word length and the fraction length of the accumulator, in bits. If applicable, you can enter separate fraction lengths for the numerator and denominator accumulator data type.
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the accumulator. If applicable, you can enter separate slopes for the numerator and denominator accumulator data type. This block requires power-of-two slope and a bias of zero.

State

Use this parameter to specify how you would like to designate the state word and fraction lengths. See "Fixed-Point Data Types" on page 2-150 for illustrations depicting the use of the state data type in this block.

This parameter is not visible for `Direct form I` structure. When you select:

- `Same as input` — **State** word and fraction length characteristics match those of the input to the block.
- `Same as accumulator` — **State** word and fraction length characteristics match those of the accumulator.
- `Binary point scaling` — Enter the word length and the fraction length of the state, in bits. If applicable, you can enter separate fraction lengths for the numerator and denominator state data type.
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the state. If applicable, you can enter separate slopes for the numerator and denominator state data type. This block requires power-of-two slope and a bias of zero.

Output

Choose how you specify the output word length and fraction length. See "Fixed-Point Data Types" on page 2-150 for illustrations depicting the use of the output data type in this block. When you select:

- `Same as input` — **Output** word and fraction length characteristics match those of the input to the block.
- `Same as accumulator` — **Output** word and fraction length characteristics match those of the accumulator.

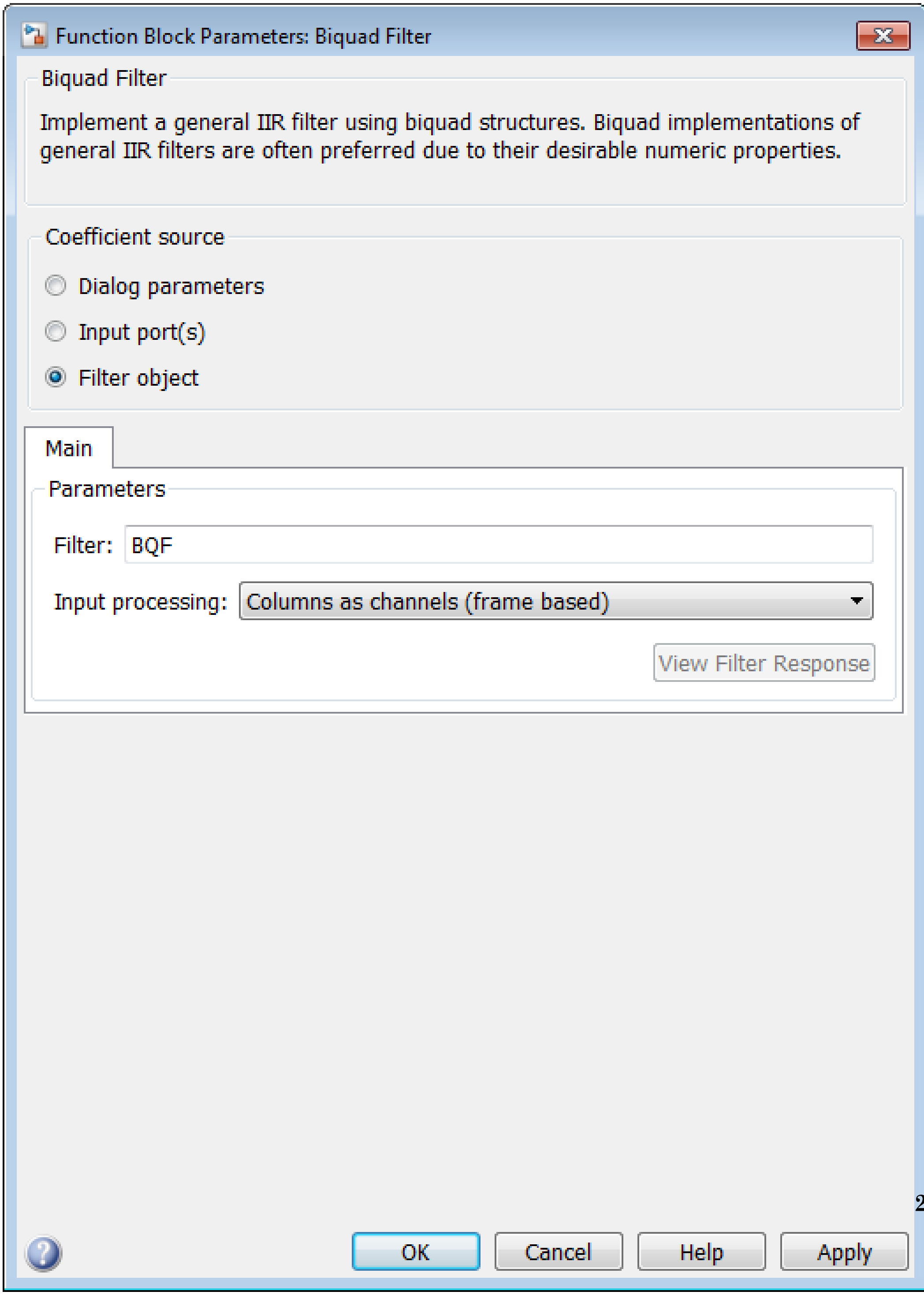
- `Binary point scaling` — Enter the word length and the fraction length of the output, in bits.
- `Slope and bias scaling` — Enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Specify Discrete-Time Filter Object

The **Main** pane of the Biquad Filter block dialog appears as follows when **Filter object** is selected in the **Coefficient source** group box.


Function Block Parameters: Biquad Filter
Biquad Filter
Implement a general IIR filter using biquad structures. Biquad implementations of general IIR filters are often preferred due to their desirable numeric properties.
Coefficient source
Dialog parameters
Input port(s)
Filter object
Main
Parameters
Filter:
BQF
Input processing:
Columns as channels (frame based)
View Filter Response
OK
Cancel
Help
Apply

Filter

Specify the name of the discrete-time filter object that you want the block to implement. You must specify the filter as a dsp.BiquadFilter System object.

You can define the System object in the block mask or in a MATLAB workspace variable.

For information on creating System objects, see “Define Basic System Objects” (MATLAB).

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the filter object specified in the **Filter object** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note: If you specify a filter in the **Filter object** parameter, you must click the **Apply** button to apply the filter before using the **View filter response** button.

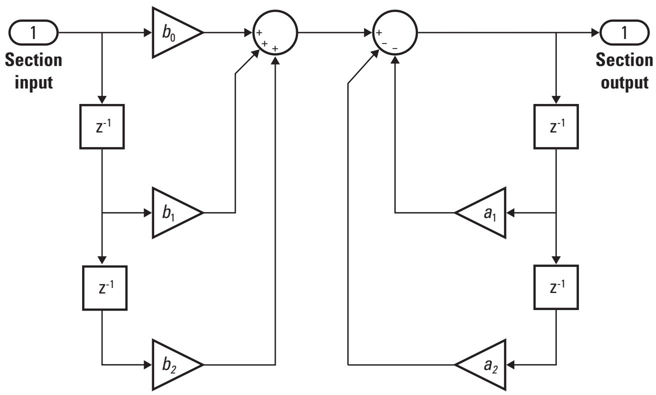
Filter Structure Diagrams

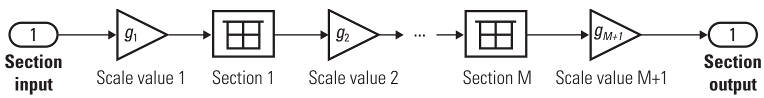
The diagrams in the following sections show the filter structures supported by the Biquad Filter block. They also show the data types used in the filter structures for fixed-point signals. You can set the data types shown in these diagrams in the block dialog box. This is discussed in “Dialog Box” on page 2-150.

- “Direct Form I” on page 2-171
- “Direct Form I Transposed” on page 2-173
- “Direct Form II” on page 2-176

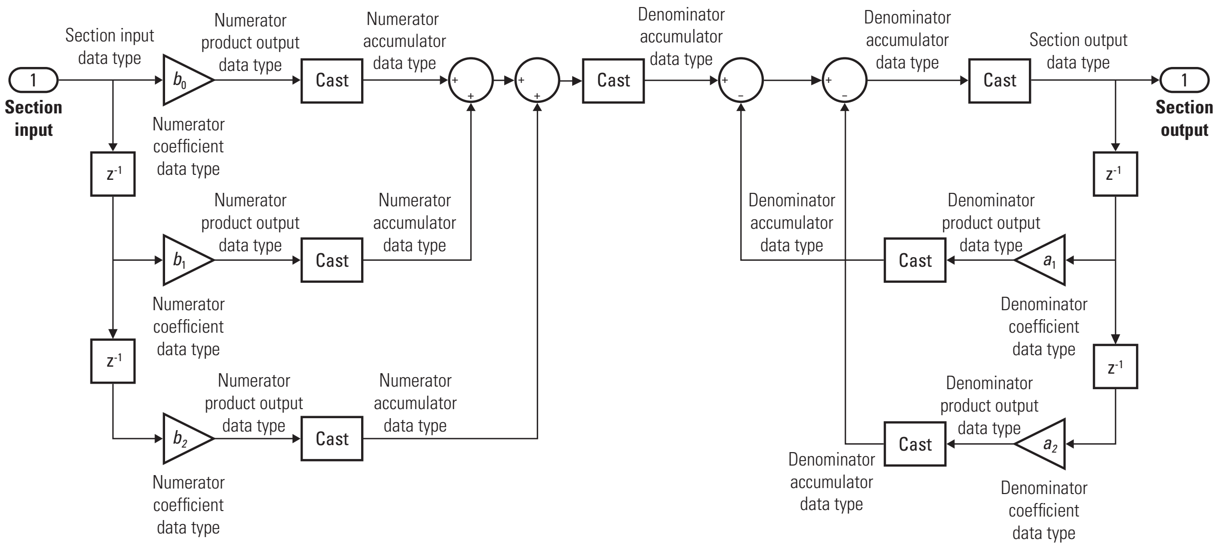
- "Direct Form II Transposed" on page 2-179

Direct Form I



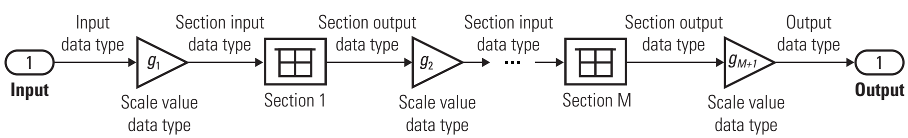


The following diagram shows the data types for one section of the filter for fixed-point signals.

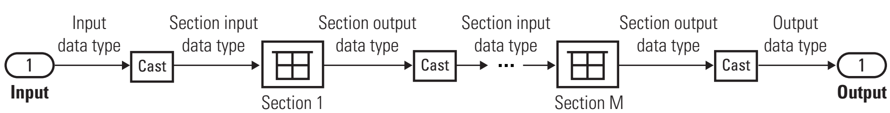


The following diagrams show the fixed-point data types between filter sections.

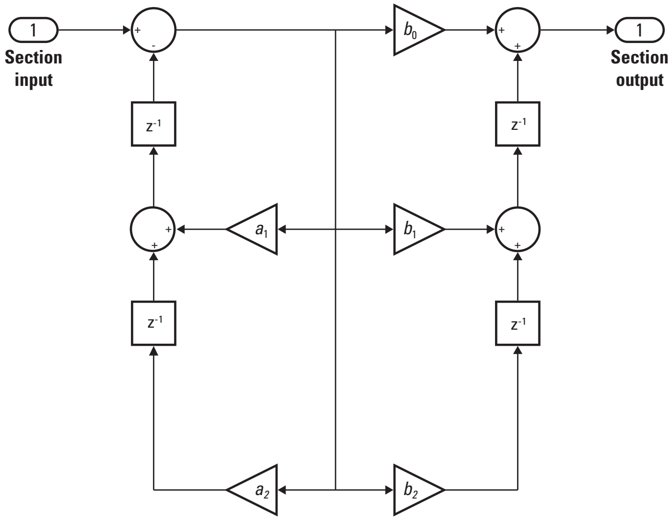
When the data is not optimized:

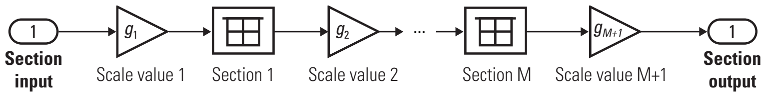


When you select **Optimize unity scale values** and scale values equal 1:

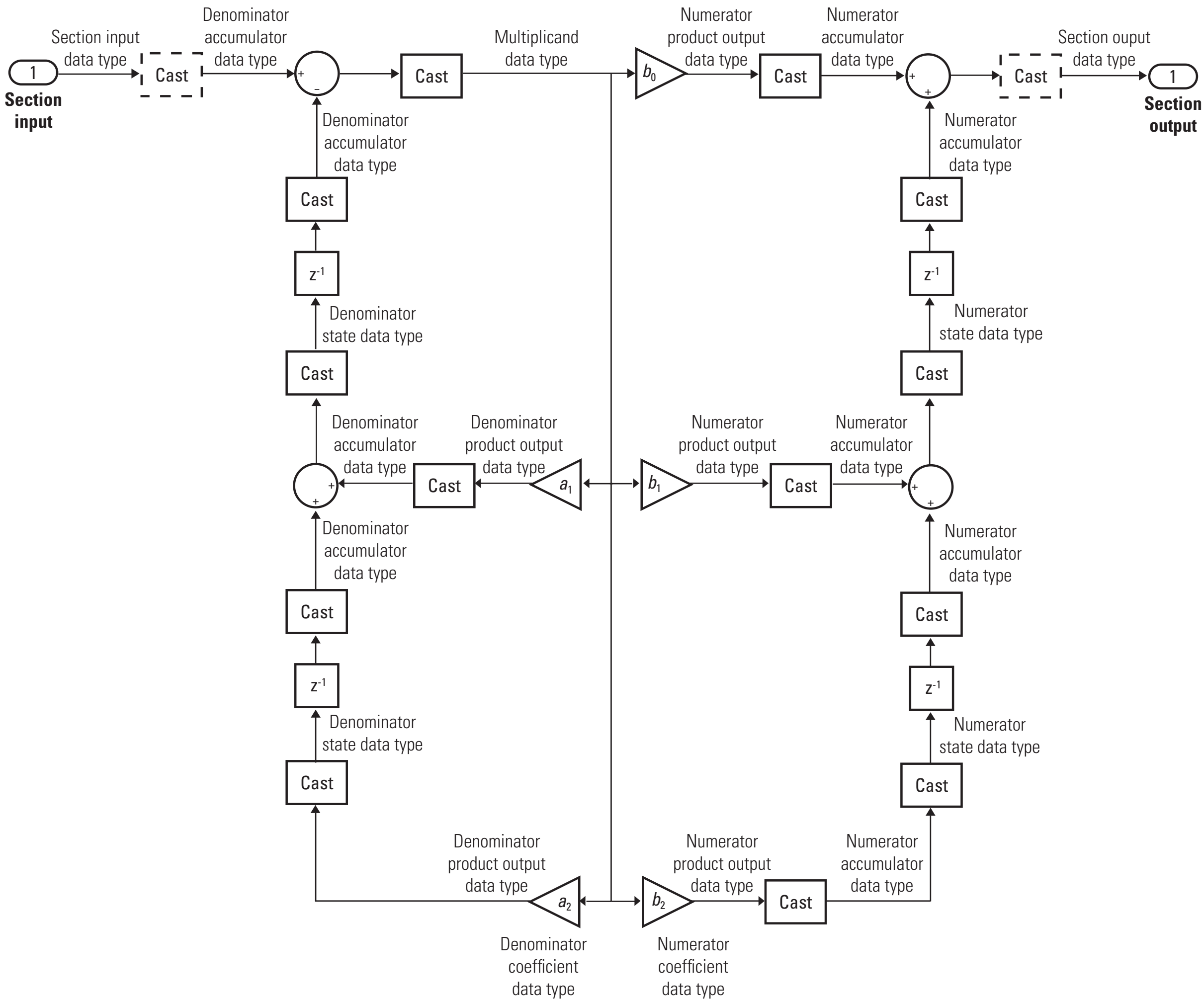


Direct Form I Transposed





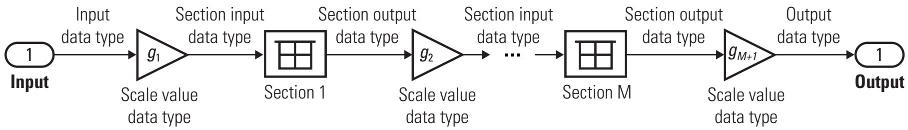
The following diagram shows the data types for one section of the filter for fixed-point signals.



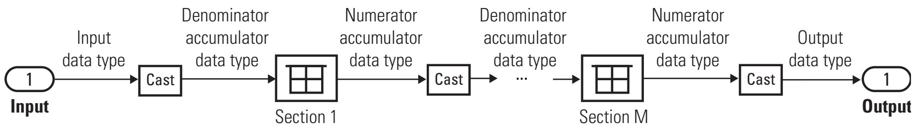
The dashed casts are omitted when **Optimize unity scale values** is selected and scale values equal one.

The following diagrams show the fixed-point data types between filter sections.

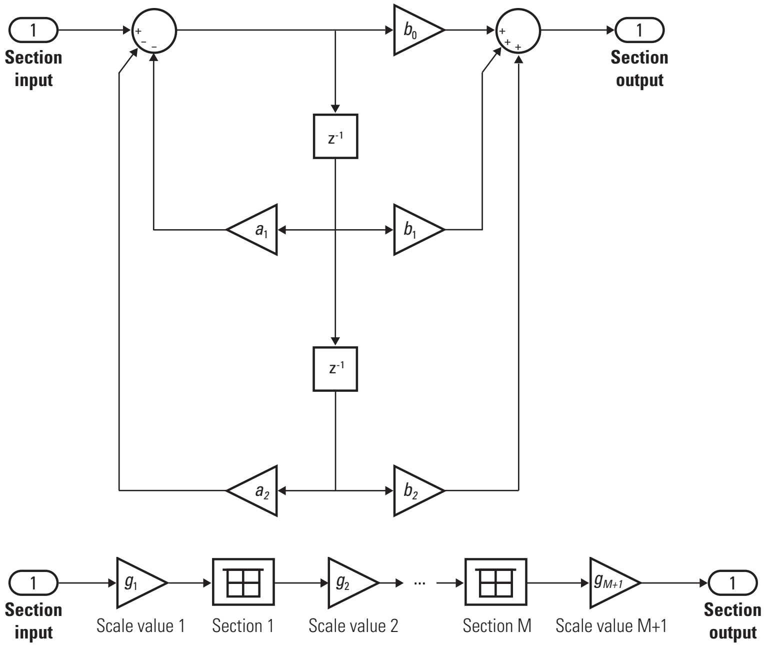
When the data is not optimized:



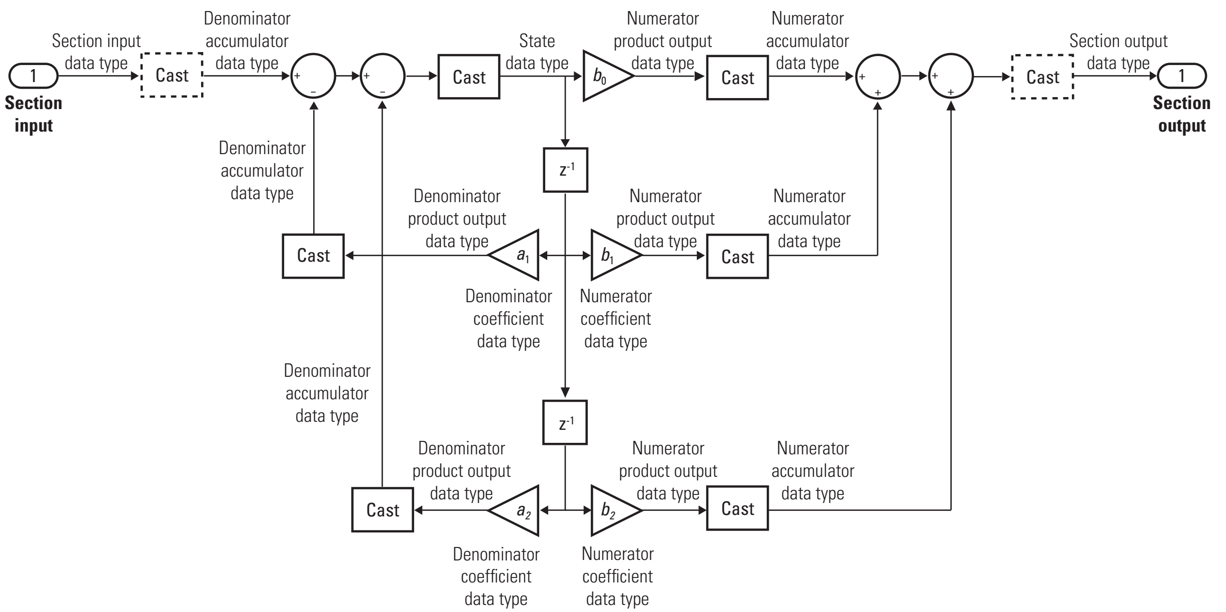
When you select **Optimize unity scale values** and scale values equal 1:



Direct Form II



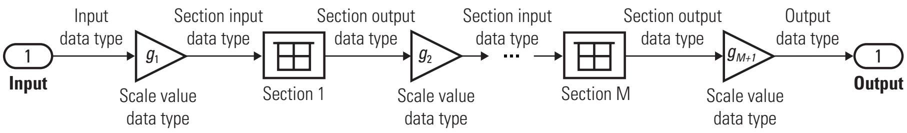
The following diagram shows the data types for one section of the filter for fixed-point signals.



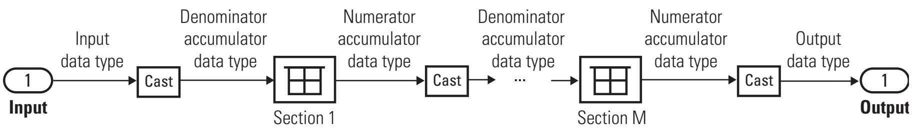
The dashed casts are omitted when **Optimize unity scale values** is selected and scale values equal one.

The following diagrams show the fixed-point data types between filter sections.

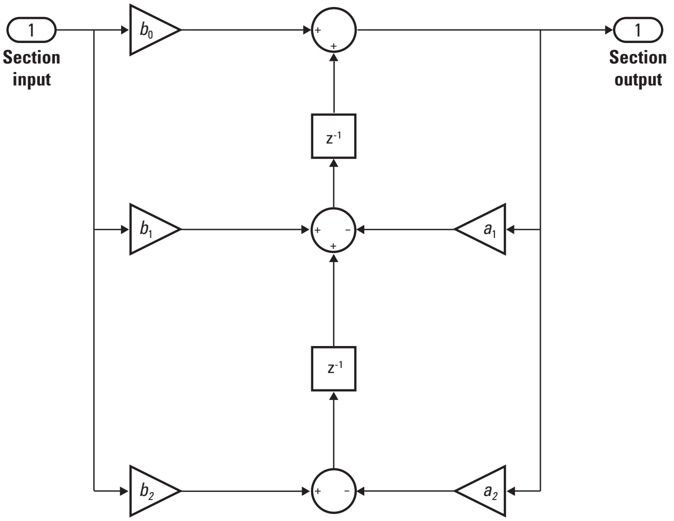
When the data is not optimized:

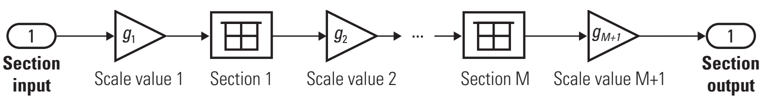


When you select **Optimize unity scale values** and scale values equal 1:

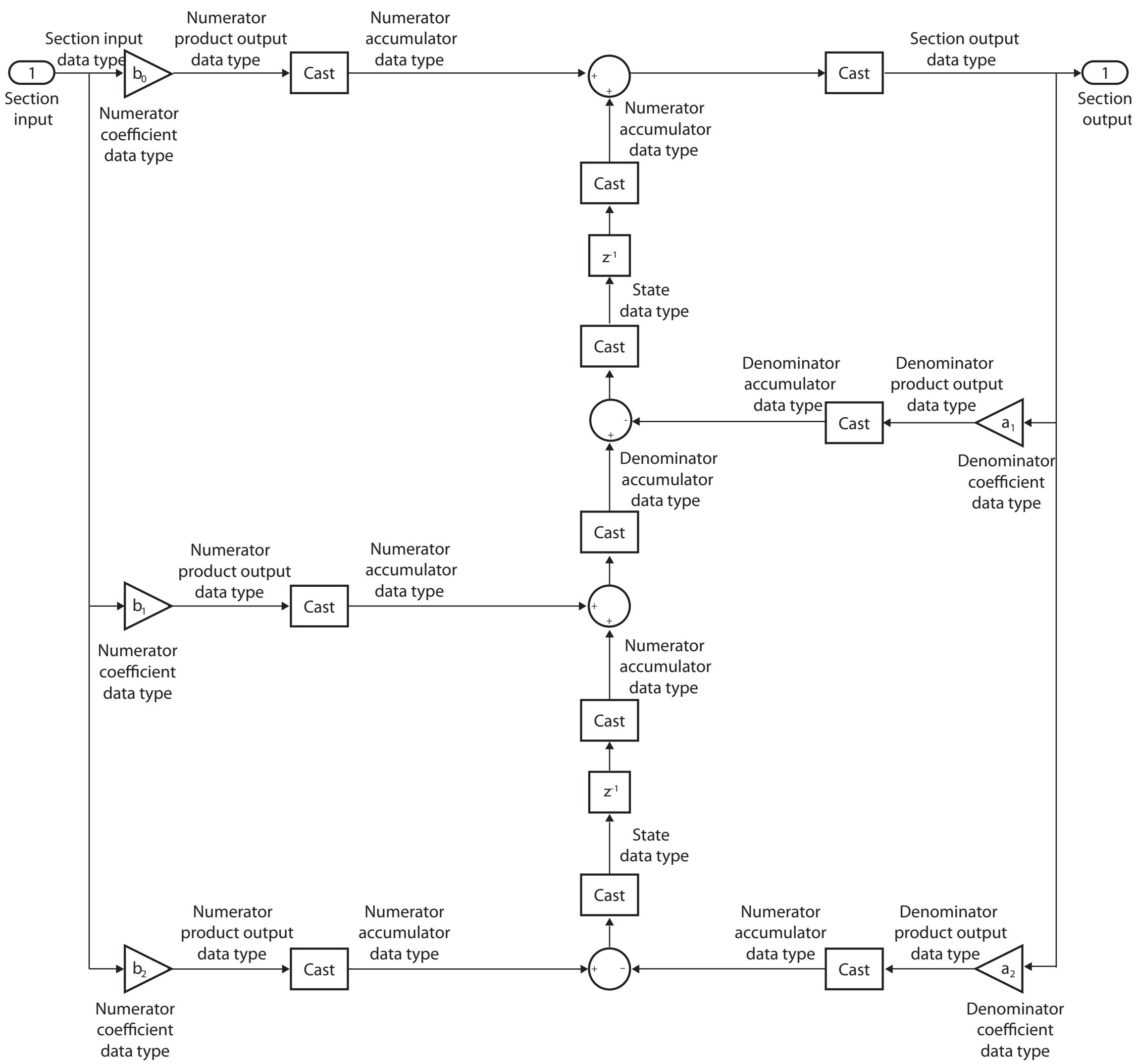


Direct Form II Transposed



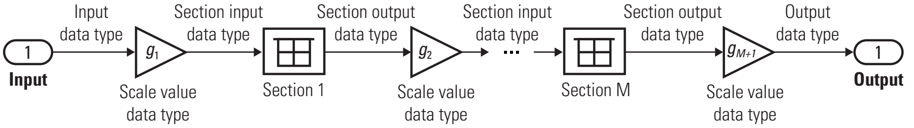


The following diagram shows the data types for one section of the filter for fixed-point signals.

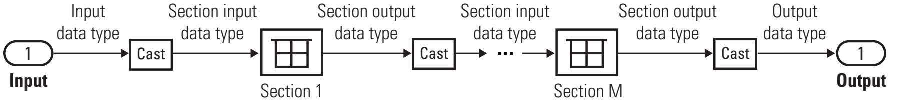


The following diagrams show the fixed-point data types between filter sections.

When the data is not optimized:



When you select **Optimize unity scale values** and scale values equal 1:



HDL Code Generation

This block supports HDL code generation using HDL Coder™. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Biquad Filter.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

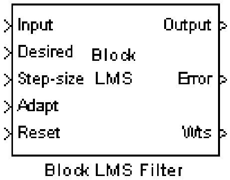
See Also

Discrete FIR Filter | `Variable Bandwidth FIR Filter` | `Variable Bandwidth IIR Filter` | dsp.BiquadFilter

Introduced in R2008b

Block LMS Filter

Compute output, error, and weights using LMS adaptive algorithm



Library

Filtering / Adaptive Filters

`dspadpt3`

Description

The Block LMS Filter block implements an adaptive least mean-square (LMS) filter, where the adaptation of filter weights occurs once for every block of samples. The block estimates the filter weights, or coefficients, needed to minimize the error, $e(n)$, between the output signal, $y(n)$, and the desired signal, $d(n)$. Connect the signal you want to filter to the Input port. The input signal can be a scalar or a column vector. Connect the signal you want to model to the `Desired` port. The desired signal must have the same data type, complexity, and dimensions as the input signal. The `Output` port outputs the filtered input signal. The `Error` port outputs the result of subtracting the output signal from the desired signal.

The block calculates the filter weights using the Block LMS adaptive filter algorithm. This algorithm is defined by the following equations.

$$
\begin{aligned}
n &= kN + i \\
y(n) &= \mathbf{w}^T(k-1)\mathbf{u}(n) \\
e(n) &= d(n) - y(n) \\
\mathbf{w}(k) &= \mathbf{w}(k-1) + f(\mathbf{u}(n), e(n), \mu)
\end{aligned}
$$

The weight update function for the Block LMS adaptive filter algorithm is defined as

$$f(\mathbf{u}(n), e(n), \mu) = \mu \sum_{i=0}^{N-1} \mathbf{u}^*(kN+i)e(kN+i)$$

The variables are as follows.

Variable	Description
n	The current time index
i	The iteration variable in each block, $0 \le i \le N-1$
k	The block number
N	The block size
$\mathbf{u}(n)$	The vector of buffered input samples at step n
$\mathbf{w}(n)$	The vector of filter-tap estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at time n
$d(n)$	The desired response at time n
μ	The adaptation step size

Use the **Filter length** parameter to specify the length of the filter weights vector.

The **Block size** parameter determines how many samples of the input signal are acquired before the filter weights are updated. The number of rows in the input must be an integer multiple of the **Block size** parameter.

The adaptation **Step-size (mu)** parameter corresponds to μ in the equations. You can either specify a step-size using the input port, Step-size, or enter a value in the Block Parameters: Block LMS Filter dialog box.

Use the **Leakage factor (0 to 1)** parameter to specify the leakage factor, $0 < 1 - \mu\alpha \le 1$, in the leaky LMS algorithm shown below.

$$\mathbf{w}(k) = (1 - \mu\alpha)\mathbf{w}(k-1) + f(\mathbf{u}(n), e(n), \mu)$$

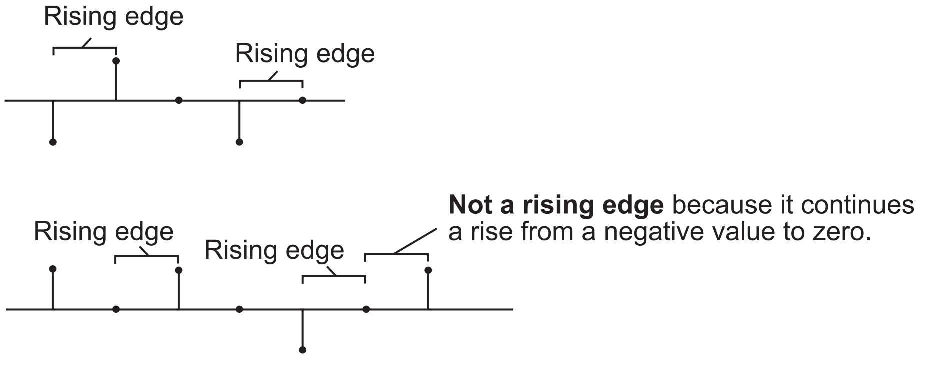
Enter the initial filter weights as a vector or a scalar in the **Initial value of filter weights** text box. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value

When you select the **Adapt port** check box, an `Adapt` port appears on the block. When the input to this port is greater than zero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain at their current values.

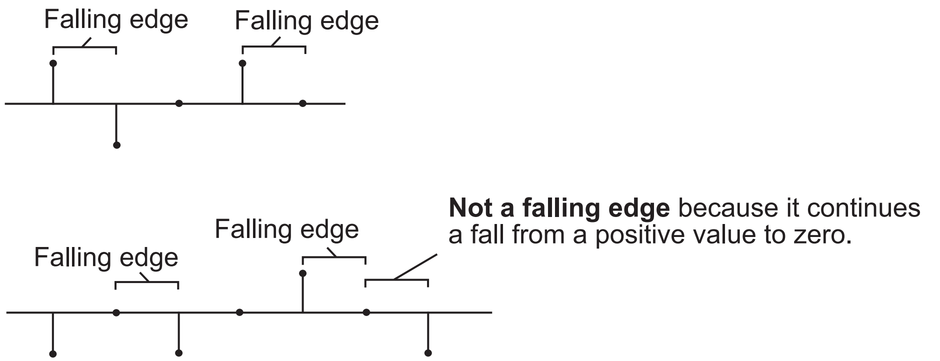
When you want to reset the value of the filter weights to their initial values, use the **Reset input** parameter. The block resets the filter weights whenever a reset event is detected at the Reset port. The reset signal rate must be the same rate as the data signal input.

From the **Reset input** list, select `None` to disable the Reset port. To enable the Reset port, select one of the following from the **Reset input** list:

- `Rising edge` — Triggers a reset operation when the Reset input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure).



- `Falling edge` — Triggers a reset operation when the Reset input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- `Either edge` — Triggers a reset operation when the Reset input is a `Rising edge` or `Falling edge` (as described above)
- `Non-zero sample` — Triggers a reset operation at each sample time that the Reset input is not zero

Select the **Output filter weights** check box to create a `Wts` port on the block. For each iteration, the block outputs the current updated filter weights from this port.

Parameters

Filter length

Enter the length of the FIR filter weights vector.

Block size

Enter the number of samples to acquire before the filter weights are updated. The number of rows in the input must be an integer multiple of the **Block size**.

Specify step-size via

Select `Dialog` to enter a value for mu in the Block parameters: LMS Filter dialog box. Select `Input port` to specify mu using the Step-size input port.

Step-size (mu)

Enter the step-size. Tunable (Simulink).

Leakage factor (0 to 1)

Enter the leakage factor, $0 < 1 - \mu\alpha \leq 1$. Tunable (Simulink).

Initial value of filter weights

Specify the initial values of the FIR filter weights.

Adapt port

Select this check box to enable the Adapt input port.

Reset port

Select this check box to enable the Reset input port.

Output filter weights

Select this check box to export the filter weights from the Wts port.

References

Hayes, M. H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Desired	• Must be the same as Input
Step-size	• Must be the same as Input
Adapt	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Reset	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Port	Supported Data Types
Output	• Same as Input
Error	• Same as Input
Wts	• Same as Input

See Also

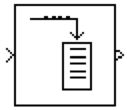
Fast Block LMS Filter	DSP System Toolbox
Kalman Adaptive Filter (Obsolete)	DSP System Toolbox
LMS Filter	DSP System Toolbox
RLS Filter	DSP System Toolbox

See “Adaptive Filters in Simulink” for related information.

Introduced before R2006a

Buffer

Buffer input sequence to smaller or larger frame size



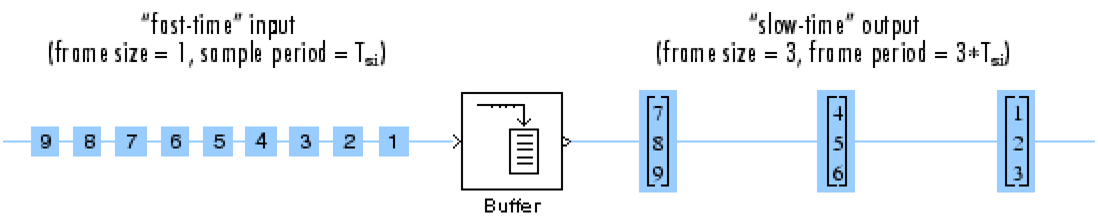
Library

Signal Management / Buffers

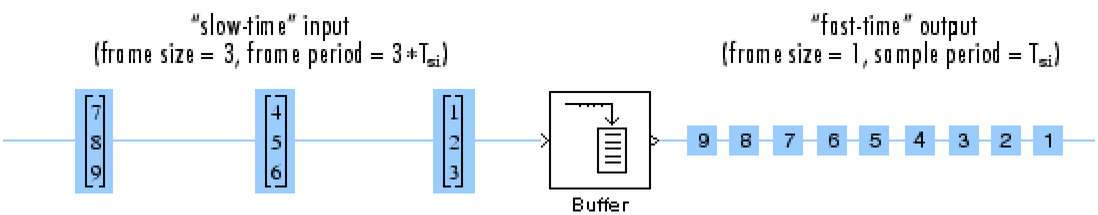
`dspbuff3`

Description

The Buffer block always performs frame-based processing. The block redistributes the data in each column of the input to produce an output with a different frame size. Buffering a signal to a larger frame size yields an output with a *slower* frame rate than the input. For example, consider the following illustration for scalar input.



Buffering a signal to a smaller frame size yields an output with a *faster* frame rate than the input. For example, consider the following illustration of scalar output.



The block coordinates the output *frame size* and *frame rate* of nonoverlapping buffers such that the sample period of the signal is the same at both the input and output: $T_{so} = T_{si}$.

This block supports triggered subsystems when the block input and output rates are the same.

Buffering Single Channel Signals

The following table shows the output dimensions of the Buffer block when the input is a single-channel signal. M_o is the value of the **Output buffer size** parameter.

Input Dimensions	Output Dimensions
1-by-1 (scalar)	M_o-by-1
M_i-by-1 (column vector)	M_o-by-1

The input frame period is $M_i \cdot T_{si}$, where M_i is the input frame size and T_{si} is the input sample period. The output frame period is $(M_o - L)T_{si}$, where L is the value of the **Buffer overlap** parameter and T_{si} is the input sample period. When you set the **Buffer overlap** parameter to $M_o - 1$, the output frame period equals the input sample period.

Buffering Multichannel Signals

The following table shows the output dimensions of the Buffer block when the input is a multichannel signal. M_o is the value of the **Output buffer size** parameter and can be greater or less than the input frame size, M_i. The block buffers each of the N input channels independently.

Input Dimensions	Output Dimensions
1-by-N (row vector)	M_o-by-N
M_i-by-N (matrix)	M_o-by-N

The input frame period is $M_i \cdot T_{si}$, where M_i is the input frame size and T_{si} is the input sample period. The output frame period is $(M_o - L)T_{si}$, which equals the sequence sample period when the **Buffer overlap** is $M_o - 1$. Thus, the output sample period is related to the input sample period by

$$T_{so} = \frac{(M_o - L)T_{si}}{M_i}$$

Buffering with Overlap or Underlap

The **Buffer overlap** parameter, L, specifies the amount of overlap or underlap in each successive output frame. To overlap the data in the buffer, specify a value of L in the range $0 \le L < M_o$, where M_o is the value of the **Output buffer size** parameter. The block takes L samples (rows) from the current output and repeats them in the next output. In cases of overlap, the block acquires $M_o - L$ *new* input samples before propagating the buffered data to the output.

When $L < 0$, you are buffering the signal with underlap. The block discards L input samples after the buffer fills and outputs the buffer with period $(M_o - L)T_{si}$, which is longer than in the zero-overlap case.

The output frame period is $(M_o - L)T_{si}$, which equals the input sequence sample period, T_{si}, when the **Buffer overlap** is $M_o - 1$.

Latency

Zero-tasking latency means that the first input sample, received at $t = 0$, appears as the first output sample. In the Simulink single-tasking mode, the Buffer block has zero-tasking latency for the following special cases:

- Scalar input and output ($M_o = M_i = 1$) with zero or negative **Buffer overlap** ($L \le 0$)
- Input frame size is an integer multiple of the output frame size

  $$M_i = kM_o$$

  where k is an integer with zero **Buffer overlap** ($L = 0$); notable cases of this include

- Any input frame size M_i with scalar output ($M_o = 1$) and zero **Buffer overlap** ($L = 0$)
- Equal input and output frame sizes ($M_o = M_i$) with zero **Buffer overlap** ($L = 0$)

For all cases of single-tasking operation other than those listed above, the Buffer block's buffer is initialized to the value(s) specified by the **Initial conditions** parameter. The block reads from this buffer to generate the first D output samples, where

$$D = \begin{cases} M_o + L & (L \geq 0) \\ M_o & (L < 0) \end{cases}$$

The dimensions of the **Initial conditions** parameter depend on the **Buffer overlap**, L, and whether the input is single-channel or multichannel:

- When $L \neq 0$, the **Initial conditions** parameter must be a scalar.
- When $L = 0$, the **Initial conditions** parameter can be a scalar, or it can be a vector with the following constraints:
  - For single-channel inputs, the **Initial conditions** parameter can be a vector of length M_o if M_i is 1, or a vector of length M_i if M_o is 1.
  - For multichannel inputs, the **Initial conditions** parameter can be a vector of length $M_o * N$ if M_i is 1, or a vector of length $M_i * N$ if M_o is 1.

For all *multitasking* operations, use the `rebuffer_delay` function to compute the exact delay, in samples, that the Buffer block introduces for a given combination of buffer size and buffer overlap.

For general buffering between arbitrary frame sizes, the **Initial conditions** parameter must be a scalar value, which is then repeated across all elements of the initial output(s). However, in the special case where the input is a 1-by-N row vector, and the output of the block is an M_o-by-N matrix, **Initial conditions** can be

- An M_o-by-N matrix
- A length-M_o vector to be repeated across all columns of the initial output(s)
- A scalar to be repeated across all elements of the initial output(s)

In the special case where the output is a 1-by-N row vector, which is the result of unbuffering an M_i-by-N matrix, the **Initial conditions** can be

- A vector containing M_i samples to output sequentially for each channel during the first M_i sample times
- A scalar to be repeated across all elements of the initial output(s)

Note For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

Behavior in Enabled Subsystems

The Buffer block cannot be used in an enabled subsystem under the following conditions:

- In a multirate multitasking environment
- When the **Buffer overlap** parameter is set to a negative value

The Buffer block has an internal reservoir that temporarily stores data. When the Buffer block is used in an enabled subsystem, there is the possibility that the reservoir can overrun or underrun. The block implements safeguards against these occurrences.

Overrun occurs when more data enters into the buffer than it can hold. For example, consider buffering a scalar input to a frame of size three with a buffer that accepts input every second and outputs every three seconds. If you place this buffer inside an enabled subsystem that is disabled every three seconds at t = 3s, t = 6s, and so on, the buffer accumulates data in its internal reservoir without being able to empty it. This condition results in overrun.

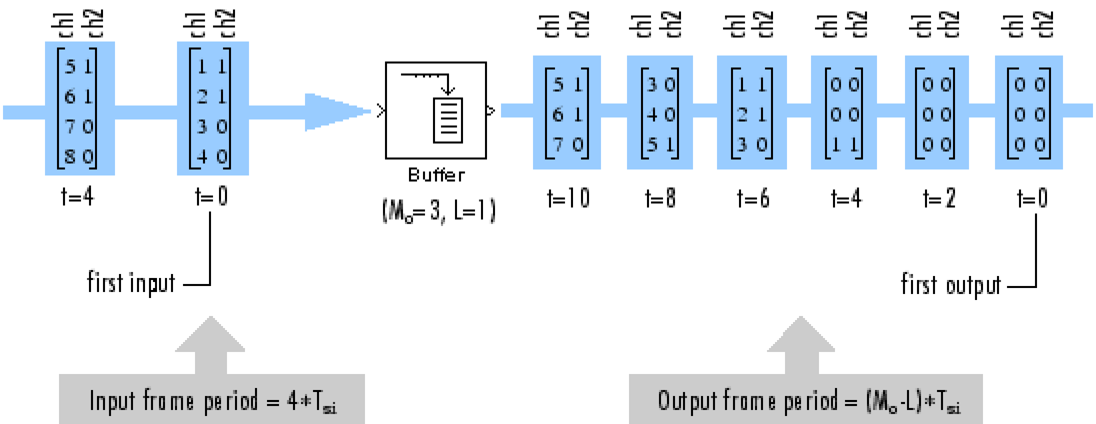
Underrun occurs when the buffer runs out of data to output. For example, again consider buffering a scalar input to a frame size of three with a buffer that accepts input every second and outputs every three seconds. If you place this buffer inside an enabled subsystem that is disabled at t = 10s, t = 11s, t = 13s, t = 14s, t = 16s, and t = 17s, its internal reservoir becomes drained, and there is no data to output at t = 18s. This condition results in underrun.

To protect from overrun and underrun, the Buffer block keeps a record of the amount of data in its internal reservoir. When the Buffer block reads data, the amount of data in its reservoir goes up. When data is output from the Buffer block, the amount of data in its reservoir goes down. To protect from overrun, the oldest samples in the reservoir are discarded whenever amount of data in the reservoir is larger than the actual buffer size. To protect from underrun, the most recent samples are repeated whenever an output is due and there is no data in the reservoir.

Examples

Buffering a two-channel input with overlap

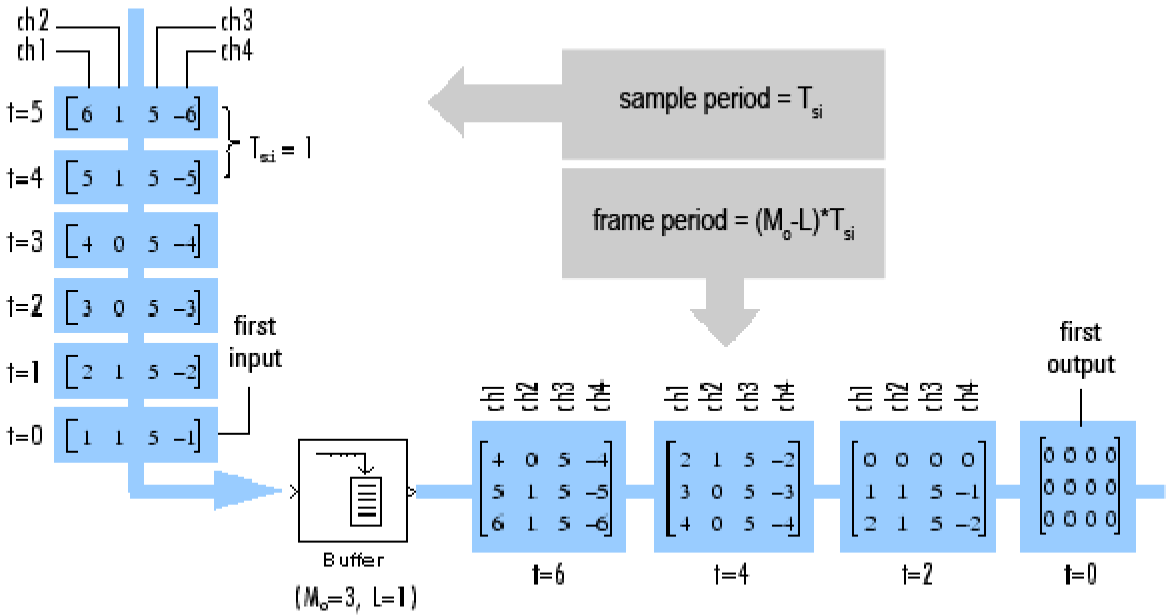
The ex_buffer_tut4 model buffers a two-channel input using an **Output buffer size** of three and a **Buffer overlap** of one. The following diagram illustrates the inputs and outputs of the Buffer block.



Notice that the output is delayed by eight samples. This is latency occurs because of the parameter settings in this model, and because the model is running in Simulink multitasking mode. The first eight output samples therefore adopt the value specified for the **Initial conditions** parameter, which in this case is set to zero. You can use the `rebuffer_delay` function to determine the latency of the Buffer block for any combination of frame size and overlap values.

Buffering a four-channel input with overlap

The ex_buffer_tut3 buffers a four-channel input using a **Output buffer size** of three and a **Buffer overlap** of one. The following diagram illustrates the inputs and outputs of the Buffer block.



Notice that the input vectors do not begin appearing at the output until the second row of the second matrix. This is due to latency in the Buffer block. The first output matrix (all zeros in this example) reflects the value of the **Initial conditions** parameter, while the first row of zeros in the second output is a result of the one-sample overlap between consecutive output frames.

You can use the `rebuffer_delay` function with a frame size of 1 to precisely compute the delay (in samples). For the previous example,

```
d = rebuffer_delay(1,3,1)
d =
     4
```

This agrees with the four samples of delay (zeros) per channel shown in the previous figure.

Parameters

Output buffer size

Specify the number of consecutive samples, M_o, from each channel to buffer into the output frame.

Buffer overlap

Specify the number of samples, L, by which consecutive output frames overlap.

Initial conditions

Specify the value of the block's initial output for cases of nonzero latency; a scalar, vector, or matrix.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

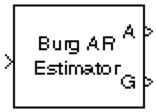
`Delay Line`	DSP System Toolbox
`Unbuffer`	DSP System Toolbox
`rebuffer_delay`	DSP System Toolbox

See “Convert Sample and Frame Rates in Simulink” and “Buffering and Frame-Based Processing” for more information.

Introduced before R2006a

Burg AR Estimator

Compute estimate of autoregressive (AR) model parameters using Burg method



Library

Estimation / Parametric Estimation

`dspparest3`

Description

The Burg AR Estimator block uses the Burg method to fit an autoregressive (AR) model to the input data by minimizing (least squares) the forward and backward prediction errors while constraining the AR parameters to satisfy the Levinson-Durbin recursion.

The input must be a column vector or an unoriented vector, which is assumed to be the output of an AR system driven by white noise. This input represents a frame of consecutive time samples from a single-channel signal. The block computes the normalized estimate of the AR system parameters, $A(z)$, independently for each successive input frame.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

When you select the **Inherit estimation order from input dimensions** parameter, the order, p, of the all-pole model is one less than the length of the input vector. Otherwise, the order is the value specified by the **Estimation order** parameter.

The **Output(s)** parameter allows you to select between two realizations of the AR process:

- A — The top output, A, is a column vector of length p+1 with the same frame status as the input, and contains the normalized estimate of the AR model polynomial coefficients in descending powers of z.

```
[1 a(2) ... a(p+1)]
```

- `K` — The top output, K, is a column vector of length p with the same frame status as the input, and contains the reflection coefficients (which are a secondary result of the Levinson recursion).
- `A and K` — The block outputs both realizations.

The scalar gain, G, is provided at the bottom output (`G`).

The following table compares the features of the Burg AR Estimator block to the Covariance AR Estimator, Modified Covariance AR Estimator, and Yule-Walker AR Estimator blocks.

	Burg AR Estimator	Covariance AR Estimator	Modified Covariance AR Estimator	Yule-Walker AR Estimator
Characteristics	Does not apply window to data	Does not apply window to data	Does not apply window to data	Applies window to data
	Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion	Minimizes the forward prediction error in the least squares sense	Minimizes the forward and backward prediction errors in the least squares sense	Minimizes the forward prediction error in the least squares sense (also called “autocorrelation method”)
Advantages	Always produces a stable model			Always produces a stable model
Disadvantages		May produce unstable models	May produce unstable models	Performs relatively poorly for short data records
Conditions for Nonsingularity		Order must be less than or equal to half the input frame size	Order must be less than or equal to 2/3 the input frame size	Because of the biased estimate, the autocorrelation matrix is

	Burg AR Estimator	Covariance AR Estimator	Modified Covariance AR Estimator	Yule-Walker AR Estimator
				guaranteed to positive-definite, hence nonsingular

Parameters

Output(s)

The realization to output, model coefficients, reflection coefficients, or both.

Inherit estimation order from input dimensions

When selected, sets the estimation order p to one less than the length of the input vector.

Estimation order

The order of the AR model, p. This parameter is enabled when you do not select **Inherit estimation order from input dimensions**.

References

Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
A	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
G	• Double-precision floating point • Single-precision floating point

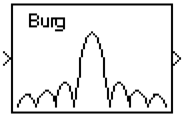
See Also

`Burg Method`	DSP System Toolbox
`Covariance AR Estimator`	DSP System Toolbox
`Modified Covariance AR Estimator`	DSP System Toolbox
`Yule-Walker AR Estimator`	DSP System Toolbox
`arburg`	Signal Processing Toolbox

Introduced before R2006a

Burg Method

Power spectral density estimate using Burg method



Library

Estimation / Power Spectrum Estimation

`dspspect3`

Description

The Burg Method block estimates the power spectral density (PSD) of the input frame using the Burg method. This method fits an autoregressive (AR) model to the signal by minimizing (least squares) the forward and backward prediction errors. Such minimization occurs with the AR parameters constrained to satisfy the Levinson-Durbin recursion.

The input must be a column vector or an unoriented vector. This input represents a frame of consecutive time samples from a single-channel signal. The block outputs a column vector containing the estimate of the power spectral density of the signal at N_{fft} equally spaced frequency points. The frequency points are in the range $[0,F_s)$, where F_s is the sampling frequency of the signal.

When you select the **Inherit estimation order from input dimensions** parameter, the order of the all-pole model is one less than the input frame size. Otherwise, the **Estimation order** parameter specifies the order. The block computes the spectrum from the FFT of the estimated AR model parameters.

Selecting the **Inherit FFT length from estimation order** parameter specifies that N_{fft} is one greater than the estimation order. Clearing the **Inherit FFT length from estimation order** check box, allows you to use the **FFT length** parameter to specify N_{fft} as a power of 2. The block zero-pads or wraps the input to N_{fft} before computing the FFT. The output is always sample based.

When you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must hold:

- The input to the block is the original signal, with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

The Burg Method and Yule-Walker Method blocks return similar results for large frame sizes. The following table compares the features of the Burg Method block to the Covariance Method, Modified Covariance Method, and Yule-Walker Method blocks.

	Burg	Covariance	Modified Covariance	Yule-Walker
Characteristics	Does not apply window to data	Does not apply window to data	Does not apply window to data	Applies window to data
	Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion	Minimizes the forward prediction error in the least squares sense	Minimizes the forward and backward prediction errors in the least squares sense	Minimizes the forward prediction error in the least squares sense (also called *autocorrelation method*)
Advantages	High resolution for short data records	Better resolution than Y-W for short data records (more accurate estimates)	High resolution for short data records	Performs as well as other methods for large data records
	Always produces a stable model	Able to extract frequencies from	Able to extract frequencies from	Always produces a stable model

	Burg	Covariance	Modified Covariance	Yule-Walker
		data consisting of p or more pure sinusoids	data consisting of p or more pure sinusoids	
			Does not suffer spectral line-splitting	
Disadvantages	Peak locations highly dependent on initial phase	May produce unstable models	May produce unstable models	Performs relatively poorly for short data records
	May suffer spectral line-splitting for sinusoids in noise, or when order is very large	Frequency bias for estimates of sinusoids in noise	Peak locations slightly dependent on initial phase	Frequency bias for estimates of sinusoids in noise
	Frequency bias for estimates of sinusoids in noise		Minor frequency bias for estimates of sinusoids in noise	
Conditions for Nonsingularity		Order must be less than or equal to half the input frame size	Order must be less than or equal to 2/3 the input frame size	Because of the biased estimate, the autocorrelation matrix is guaranteed to be positive-definite, hence nonsingular

Examples

The dspsacomp example compares the Burg method with several other spectral estimation methods.

Parameters

Inherit estimation order from input dimensions

Selecting this check box sets the estimation order to one less than the length of the input vector.

Estimation order

The order of the AR model. This parameter becomes visible only when you clear the **Inherit estimation order from input dimensions** check box.

Inherit FFT length from estimation order

When selected, the FFT length is one greater than the estimation order. To specify the number of points on which to perform the FFT, clear the **Inherit FFT length from estimation order** check box. You can then specify a power-of-two FFT length using the **FFT length** parameter.

FFT length

Enter the number of data points on which to perform the FFT, N_{fft}. When N_{fft} is larger than the input frame size, the block zero-pads each frame as needed. When N_{fft} is smaller than the input frame size, the block wraps each frame as needed. This parameter becomes visible only when you clear the **Inherit FFT length from input dimensions** check box.

Inherit sample time from input

If you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must hold:

- The input to the block is the original signal, with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

Sample time of original time series

Specify the sample time of the original time-domain signal. This parameter becomes visible only when you clear the **Inherit sample time from input** check box.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

References

[1] Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[2] Orfanidis, S. J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[3] Orfanidis, S. J. *Optimum Signal Processing: An Introduction*. 2nd ed. New York, NY: Macmillan, 1985.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- When the FFT length is not a power of two, the executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "How To Run a Generated Executable Outside MATLAB".
- When the FFT length is a power of two, you can generate standalone C and C++ code from this block.

See Also

See Also

Blocks
`Burg AR Estimator` | `Covariance Method` | `Modified Covariance Method` | `Short-Time FFT` | `Yule-Walker Method`

Topics
"Spectral Analysis"

Introduced before R2006a

Channelizer

Polyphase FFT analysis filter bank
Library: DSP System Toolbox / Filtering / Multirate Filters

Description

The Channelizer block separates a broadband input signal into multiple narrow subbands using an FFT-based analysis filter bank. The filter bank uses a prototype lowpass filter and is implemented using a polyphase structure. You can specify the filter coefficients directly or through design parameters. When you specify the design parameters, the filter is designed using the `designMultirateFIR` function.

This block accepts variable-size inputs. That is, during the simulation, you can change the size of each input channel. The number of channels cannot change.

Ports

Input

`Port_1` — Broadband signal

L-by-1 column vector | L-by-N matrix

Input broadband signal, which the channelizer splits into multiple narrow bands. The number of rows in the input signal must be a multiple of the number of frequency bands of the filter bank. Each column of the input corresponds to a separate channel.

Data Types: `single` | `double`
Complex Number Support: Yes

Output

`Port_1` — Multiple narrowband signals

L/M-by-M | L/M-by-M-by-N array

Multiple narrow subbands of the input broadband signal. Each narrow band signal forms a column in the output.

If the input is one of the following:

- L-by-1 column vector — The output is a L/M-by-M matrix. M is the number of frequency bands.
- L-by-N matrix — The output is a L/M-by-M-by-N matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

`Number of frequency bands` — Number of frequency bands
`8` (default) | positive integer greater than 1

Number of frequency bands into which the block separates the input broadband signal. This parameter indicates the FFT length and the decimation factor used by the algorithm.

`Polyphase filter specification` — Filter design parameters or coefficients
`Number of taps per band and stopband attenuation` (default) | `Coefficients`

- `Number of taps per band and stopband attenuation` — Specify the filter design parameters through the **Number of filter taps per frequency band** and **Stopband attenuation (dB)** parameters. When you specify the design parameters, the filter is designed using the `designMultirateFIR` function.
- `Coefficients` — Specify the filter coefficients directly using the **Prototype lowpass filter coefficients** parameter.

`Number of filter taps per frequency band` — Number of filter coefficients per frequency band
`12` (default) | positive integer

Number of filter coefficients that each polyphase branch uses. The number of polyphase branches matches the number of frequency bands. The total number of filter coefficients for the prototype lowpass filter is given by **Number of frequency bands** × **Number of filter taps per frequency band**. For a given stopband attenuation, increasing the number of taps per band narrows the transition width of the filter. As a result, there is more usable bandwidth for each frequency band, at the expense of increased computation.

Dependencies

To enable this parameter, set **Polyphase filter specification** to `Number of taps per band and stopband attenuation`.

`Stopband attenuation (dB)` — Stopband attenuation

`80` (default) | positive real scalar

Stopband attenuation of the lowpass filter, in dB. This value controls the maximum amount of aliasing from one frequency band to the next. Larger is the stopband attenuation, smaller is the passband ripple.

Dependencies

To enable this parameter, set **Polyphase filter specification** to `Number of taps per band and stopband attenuation`.

`Prototype lowpass filter coefficients` — Coefficients of the prototype lowpass filter

`rcosdesign(0.25,6,8,'sqrt')` (default) | row vector

Coefficients of the prototype lowpass filter. The default value is the coefficients vector `rcosdesign(0.25,6,8,'sqrt')` returns. There must be at least one coefficient per frequency band. If the length of the lowpass filter is less than the number of frequency bands, the object zero-pads the coefficients.

Tunable: Yes

Dependencies

To enable this parameter, set **Polyphase filter specification** to `Coefficients`.

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

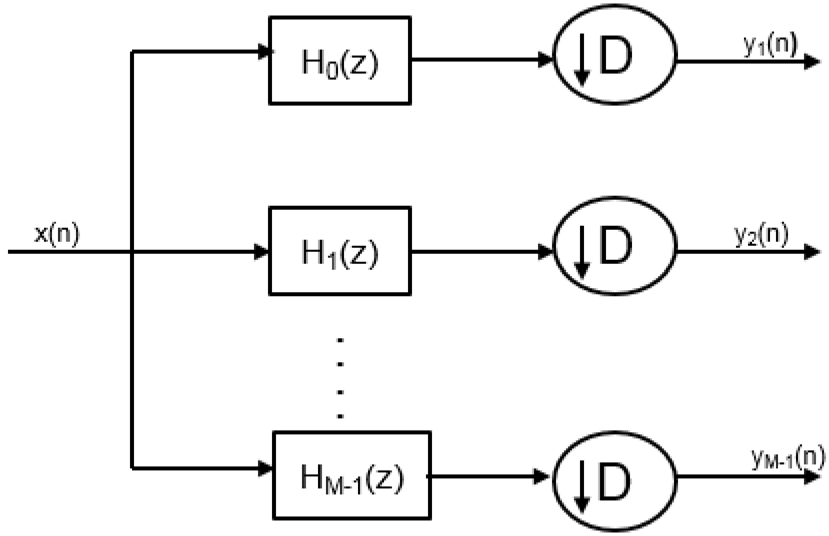
  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Model Examples

Definitions

Analysis Filter Bank

The analysis filter bank consists of a series of parallel bandpass filters that split an input broadband signal, x(n), into a series of narrow subbands. Each bandpass filter retains a different portion of the input signal. After the bandwidth is reduced by one of the bandpass filters, the signal is downsampled to a lower sampling rate commensurate with the new bandwidth.



Prototype Lowpass Filter

To implement the analysis filter bank efficiently, the channelizer uses a prototype lowpass filter. This filter has an impulse response of h[n], a normalized two-sided bandwidth of $2\pi/M$, and a cutoff frequency of π/M. M is the number of frequency bands, that is, the branches of the analysis filter bank. This value corresponds to the FFT length that the filter bank uses. M can be high, in the order of 2048 or more. The stopband attenuation determines the minimum level of interference (aliasing) from one frequency band to another. The passband ripple must be small so that the input signal is not distorted in the passband.

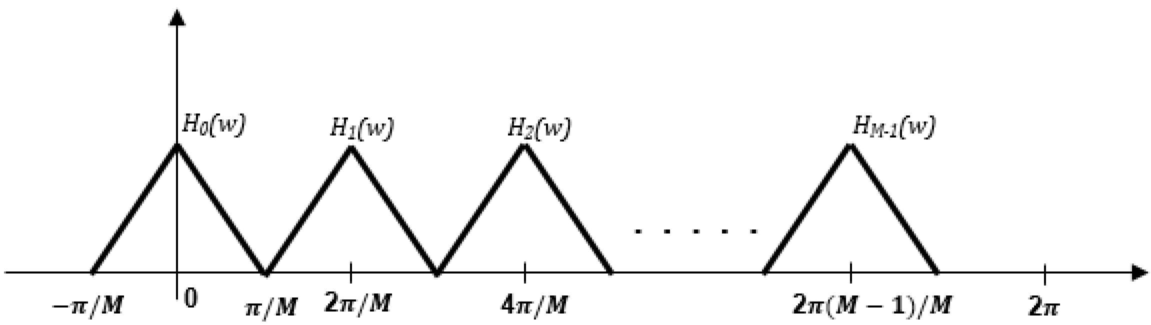
The prototype lowpass filter models the first branch of the filter bank. The other $M-1$ branches are modeled by filters that are modulated versions of the prototype filter. The modulation factor is given by $e^{-jw_k n}, \quad w_k = 2\pi k / M, \quad k = 0, 1, \ldots, M-1$.

Using Prototype Lowpass Filter

The transfer function of the modulated kth bandpass filter is given by:

$$H_k(z) = H_0(ze^{-jw_k}), \quad w_k = 2\pi k / M, \quad k = 1, 2, \ldots, M-1$$

This figure shows the frequency response of M filters.

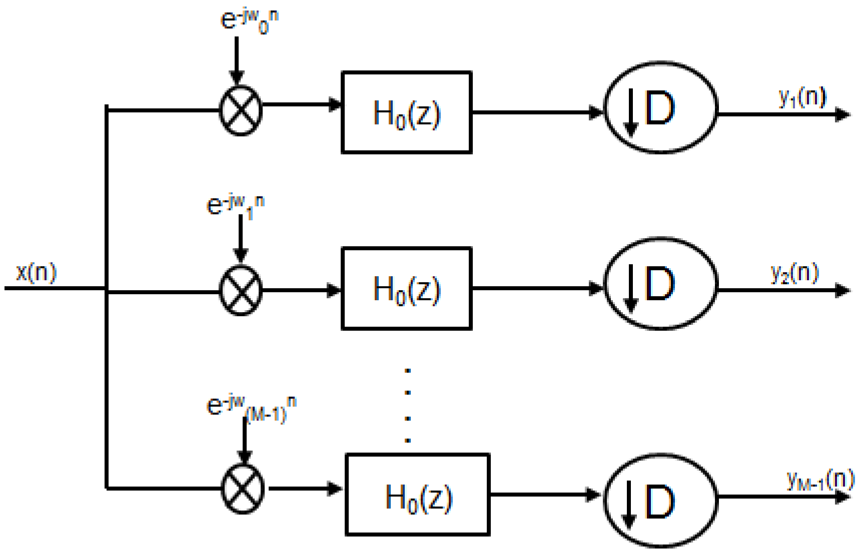


To obtain the frequency response characteristics of the filter $H_k(z)$, where $k = 1,...,M-1$, uniformly shift the frequency response of the prototype filter, $H_0(z)$, by multiples of $2\pi/M$. Each subband filter, $H_k(z)$, $\{k = 1,...,M-1\}$, is derived from the prototype filter.

Shift Narrow Subbands to Baseband

The frequency components in the input signal, $x(n)$, are translated in frequency to baseband by multiplying $x(n)$ with the complex exponentials, $e^{-jw_k n}$, *where* $w_k = 2\pi k / M$, *and* $k = 1,2,..,M-1$, where, $w_k = 2\pi k / M$, and $k = 1,2,\dots,M-1$. The resulting product signals are passed through the lowpass filters, $H_0(z)$. The output of the lowpass filter is relatively narrow in bandwidth. Downsample the signal commensurate with the new bandwidth. Choose a decimation factor, $D \le M$, the number of branches of the analysis filter bank.

The figure shows an analysis filter bank that uses the prototype lowpass filter.



$y_1(n)$, $y_2(n)$, ..., $y_{M-1}(n)$ are narrow subband signals translated into baseband.

Algorithm

Polyphase Implementation

The analysis filter bank can be implemented efficiently using the polyphase structure.

To derive the polyphase structure, start with the transfer function of the prototype lowpass filter.

$$H_0(z) = b_0 + b_1 z^{-1} + \ldots + b_N z^{-N}$$

N+1 is the length of the prototype filter.

You can rearrange this equation as follows:

$$H_0(z) = \begin{matrix} \left(b_0 + b_M z^{-M} + b_{2M} z^{-2M} + .. + b_{N-M+1} z^{-(N-M+1)}\right) + \\ z^{-1}\left(b_1 + b_{M+1} z^{-M} + b_{2M+1} z^{-2M} + .. + b_{N-M+2} z^{-(N-M+1)}\right) + \\ \vdots \\ z^{-(M-1)}\left(b_{M-1} + b_{2M-1} z^{-M} + b_{3M-1} z^{-2M} + .. + b_N z^{-(N-M+1)}\right) \end{matrix}$$

M is the number of polyphase components.

You can write this equation as:

$$H_0(z) = E_0(z^M) + z^{-1} E_1(z^M) + \ldots + z^{-(M-1)} E_{M-1}(z^M)$$

$E_0(z^M)$, $E_1(z^M)$, ..., $E_{M-1}(z^M)$ are polyphase components of the prototype lowpass filter, H_0(z).

The other filters in the filter bank, H_k(z), where k = 1, ..., M-1, are modulated versions of this prototype filter.

You can write the transfer function of the kth modulated bandpass filter as $H_k(z) = H_0(ze^{-jw_k})$. Replacing z with ze^{-jw_k},

$$H_k(z) = h_0 + h_1 e^{-jwk} z^{-1} + h_2 e^{-j2wk} z^{-2} \ldots + h_N e^{-jNwk} z^{-N}$$

N+1 is the length of the kth filter.

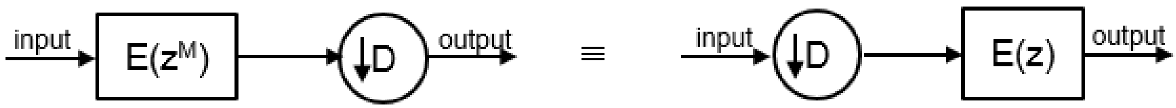
In polyphase form, the equation is as follows:

$$H_k(z) = \begin{bmatrix} 1 & e^{-jw_k} & e^{-j2w_k} & \ldots & e^{-j(M-1)w_k} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1} E_1(z^M) \\ \vdots \\ z^{-(M-1)} E_{M-1}(z^M) \end{bmatrix}$$

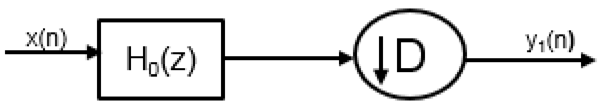
For all M channels in the filter bank, the MIMO transfer function, H(z), is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{-jw_1} & e^{-j2w_1} & \dots & e^{-j(M-1)w_1} \\ & & \vdots & & \\ 1 & e^{-jw_{M-1}} & e^{-j2w_{M-1}} & \dots & e^{-j(M-1)w_{M-1}} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1}E_1(z^M) \\ \vdots \\ z^{-(M-1)}E_{M-1}(z^M) \end{bmatrix}$$

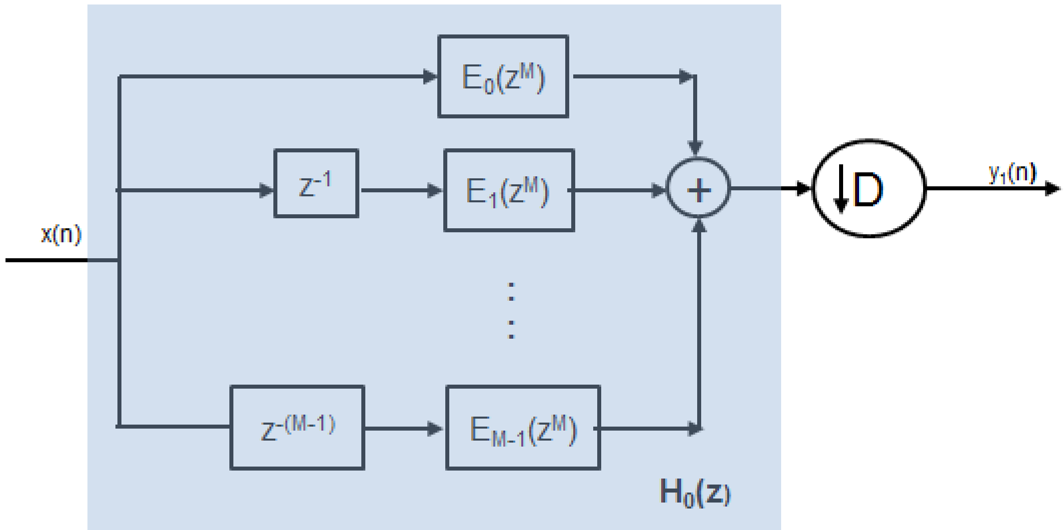
Here is the multirate noble identity for decimation.



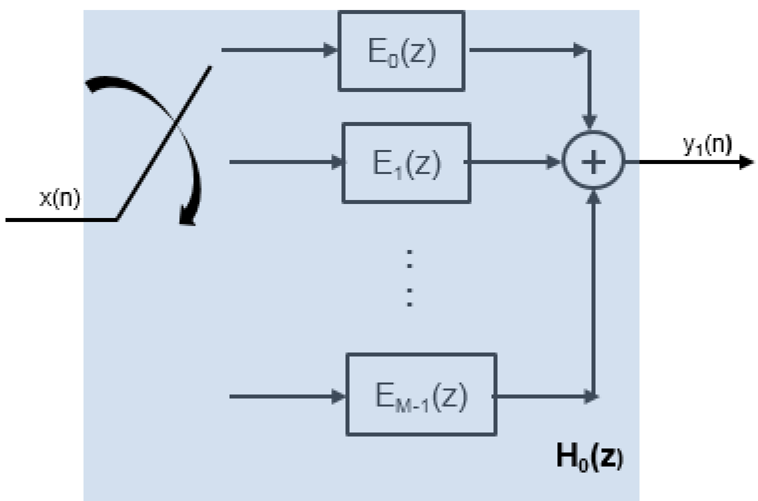
For illustration, consider the first branch of the filter bank that contains the lowpass filter.



Replace H_0(z) with its polyphase representation.



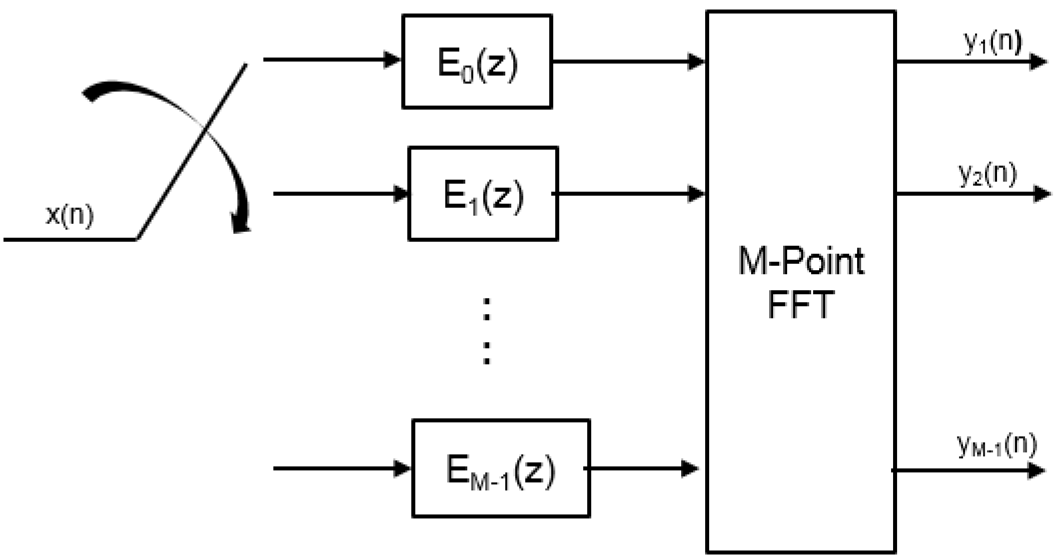
After applying the noble identity for decimation, you can replace the delays and the decimation factor with a commutator switch.



For all *M* channels in the filter bank, the MIMO transfer function, *H*(z), is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{-jw_1} & e^{-j2w_1} & \dots & e^{-j(M-1)w_1} \\ & & \vdots & & \\ 1 & e^{-jw_{M-1}} & e^{-j2w_{M-1}} & \dots & e^{-j(M-1)w_{M-1}} \end{bmatrix} \begin{bmatrix} E_0(z) \\ E_1(z) \\ \vdots \\ E_{M-1}(z) \end{bmatrix}$$

The matrix on the left is a DFT matrix. With the DFT matrix, the efficient implementation of the lowpass prototype based filter bank looks like the following.



References

[1] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.

[2] Harris, F.J., Chris Dick, Michael Rice. "Digital Receivers and Transmitters Using Polyphase Filter Banks for Wireless Communications." *IEEE® Transactions on microwave theory and techniques*. Vol. 51, Number 4, April 2003.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Blocks

`Channel Synthesizer` | `Dyadic Analysis Filter Bank` | `Two-Channel Analysis Subband Filter`

System Objects

dsp.Channelizer | dsp.ChannelSynthesizer | dsp.SubbandAnalysisFilter | dsp.DyadicAnalysisFilterBank

Introduced in R2017a

Channel Synthesizer

Polyphase FFT synthesis filter bank

Library: DSP System Toolbox / Filtering / Multirate Filters

Description

The Channel Syntheiszer block merges multiple narrowband signals into a broadband signal by using an FFT-based synthesis filter bank. The filter bank uses a prototype lowpass filter and is implemented using a polyphase structure. You can specify the filter coefficients directly or through design parameters. When you specify the design parameters, the filter is designed using the `designMultirateFIR` function.

This block also accepts variable-size inputs. That is, during the simulation, you can change the size of each input channel. The number of channels cannot change.

Ports

Input

`Port_1` — Input narrowband signals

L-by-M matrix | L-by-M-by-N array

Input narrowband signals, which the channel synthesizer merges to form the broadband signal. Each narrowband signal forms a column in the input signal. The number of columns in the input correspond to the number of frequency bands of the filter bank. If the input is three-dimensional, each matrix corresponds to a separate channel.

Data Types: `single` | `double`

Complex Number Support: Yes

Output

`Port_1` — Broadband signal

$L \times M$-by-1 vector | $L \times M$-by-N matrix

Broadband signal the channel synthesizer forms from the multiple input narrow subbands.

If the input is one of the following:

- L-by-M matrix — The output is a $L \times M$-by-1 vector. M is the number of frequency bands.
- L-by-M-by-N matrix — The output is a $L \times M$-by-N matrix.

Data Types: `single` | `double`
Complex Number Support: Yes

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

`Polyphase filter specification` — Filter design parameters or coefficients

`Number of taps per band and stopband attenuation` (default) | `Coefficients`

- `Number of taps per band and stopband attenuation` — Specify the filter design parameters through the **Number of filter taps per frequency band** and **Stopband attenuation (dB)** parameters. When you specify the design parameters, the filter is designed using the `designMultirateFIR` function.
- `Coefficients` — Specify the filter coefficients directly using the **Prototype lowpass filter coefficients** parameter.

`Number of filter taps per frequency band` — Number of filter coefficients per frequency band

`12` (default) | positive integer

Number of filter coefficients that each polyphase branch uses. The number of polyphase branches matches the number of frequency bands. The total number of filter coefficients for the prototype lowpass filter is given by the product of the number of frequency bands and the number of filter taps per frequency band. The number of frequency bands equals the number of columns in the input. For a given stopband attenuation, increasing the number of taps per band narrows the transition width of the filter. As a result, there is more usable bandwidth for each frequency band, at the expense of increased computation.

This parameter appears when you set **Polyphase filter specification** to `Number of taps per band and stopband attenuation`.

Dependencies

To enable this parameter, set **Polyphase filter specification** to `Number of taps per band and stopband attenuation`.

`Stopband attenuation (dB)` — Stopband attenuation

`80` (default) | positive real scalar

Stopband attenuation of the lowpass filter, in dB. This value controls the maximum amount of aliasing from one frequency band to the next. Larger is the stopband attenuation, smaller is the passband ripple.

Dependencies

To enable this parameter, set **Polyphase filter specification** to `Number of taps per band and stopband attenuation`.

`Prototype lowpass filter coefficients` — Coefficients of the prototype lowpass filter

`rcosdesign(0.25,6,8,'sqrt')` (default) | row vector

This parameter specifies the coefficients of the prototype lowpass filter. The default value is the coefficients vector `rcosdesign(0.25,6,8,'sqrt')` returns. There must be at least one coefficient per frequency band. If the length of the lowpass filter is less than the number of frequency bands, the object zero-pads the coefficients.

Tunable: Yes

Dependencies

To enable this parameter, set **Polyphase filter specification** to `Coefficients`.

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

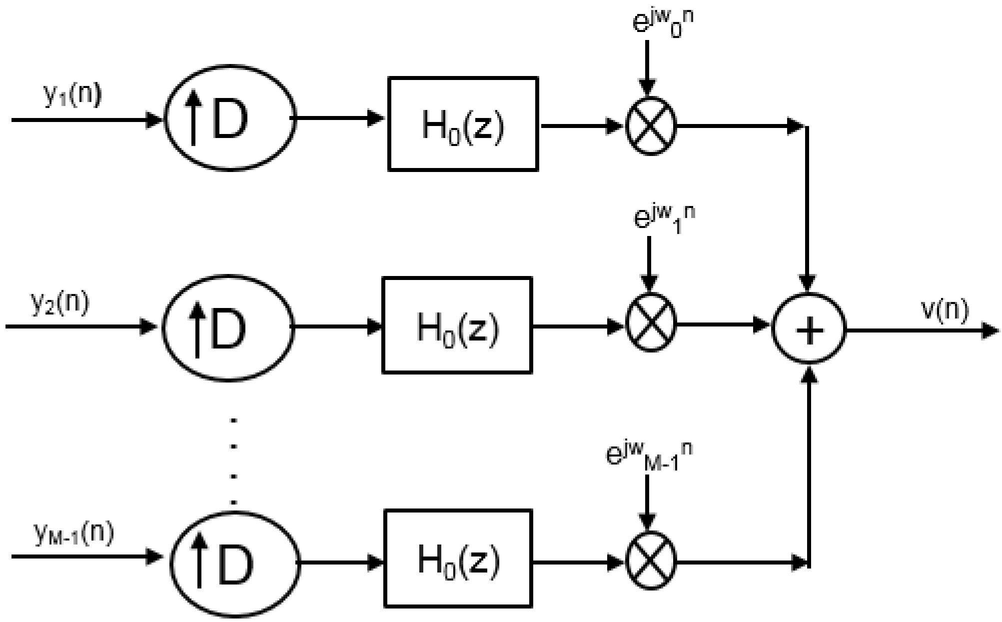
  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Model Examples

Definitions

Synthesis Filter Bank

The synthesis filter bank consists of a set of parallel bandpass filters that merge multiple input narrowband signals, y_1(n), y_2(n),..., y_{M-1}(n) into a single broadband signal, *v(n)*. The input narrowband signals are in the baseband. Each narrowband signal is interpolated to a higher sampling rate by using the upsampler, and then filtered by the lowpass filter. A complex exponential that follows the lowpass filter centers the baseband signal around w_k.



Prototype Lowpass Filter

To implement the synthesis filter bank efficiently, the synthesizer uses a prototype lowpass filter. This filter has an impulse response of h[n], a normalized two-sided bandwidth of $2\pi/M$, and a cutoff frequency of π/M. M is the number of frequency bands, that is, the branches of the synthesis filter bank. This value corresponds to the FFT length that the filter bank uses. M can be high, in the order of 2048 or more. The stopband attenuation determines the minimum level of interference (aliasing) from one frequency band to another. The passband ripple must be small so that the input signal is not distorted in the passband.

The prototype lowpass filter models the first branch of the filter bank. The other $M-1$ branches are modeled by filters that are modulated versions of the prototype filter. The modulation factor is given by $e^{jw_k n}, \quad w_k = 2\pi k / M, \quad k = 0,1,\ldots,M-1$.

The output of each bandpass filter forms a specific portion of the broadband signal. The output of all the branches are added to form the broadband signal, $v(n)$.

Algorithm

Polyphase Implementation

The synthesis filter bank can be implemented efficiently using the polyphase structure.

To derive the polyphase structure, start with the transfer function of the prototype lowpass filter.

$$H_0(z) = b_0 + b_1 z^{-1} + \ldots + b_N z^{-N}$$

N+1 is the length of the prototype filter.

You can rearrange this equation as follows:

$$\begin{aligned} H_0(z) = \; & \left(b_0 + b_M z^{-M} + b_{2M} z^{-2M} + .. + b_{N-M+1} z^{-(N-M+1)}\right) + \\ & z^{-1}\left(b_1 + b_{M+1} z^{-M} + b_{2M+1} z^{-2M} + .. + b_{N-M+2} z^{-(N-M+1)}\right) + \\ & \vdots \\ & z^{-(M-1)}\left(b_{M-1} + b_{2M-1} z^{-M} + b_{3M-1} z^{-2M} + .. + b_N z^{-(N-M+1)}\right) \end{aligned}$$

M is the number of polyphase components.

You can write this equation as:

$$H_0(z) = E_0(z^M) + z^{-1} E_1(z^M) + \ldots + z^{-(M-1)} E_{M-1}(z^M)$$

$E_0(z^M)$, $E_1(z^M)$,..., $E_{M-1}(z^M)$ are polyphase components of the prototype lowpass filter, H_0(z).

The other filters in the filter bank, H_k(z), where k = 1, ..., M-1, are modulated versions of this prototype filter.

You can write the transfer function of the kth modulated bandpass filter as $H_k(z) = H_0(ze^{jw_k})$. Replacing z with ze^{jw_k},

$$H_k(z) = h_0 + h_1 e^{jwk} z^{-1} + h_2 e^{j2wk} z^{-2} \ldots + h_N e^{jNwk} z^{-N}$$

N+1 is the length of the kth filter.

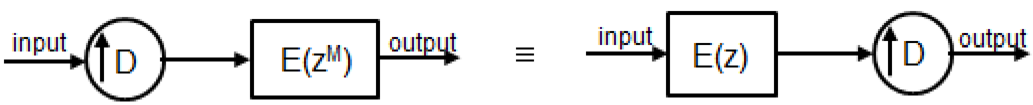
In polyphase form, the equation is as follows:

$$H_k(z) = \begin{bmatrix} 1 & e^{jw_k} & e^{j2w_k} & \ldots & e^{j(M-1)w_k} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1}E_1(z^M) \\ \vdots \\ z^{-(M-1)}E_{M-1}(z^M) \end{bmatrix}$$

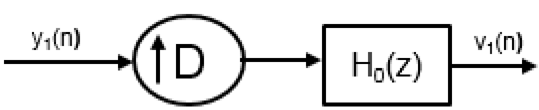
For all M channels in the filter bank, the MIMO transfer function, H(z), is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & e^{jw_1} & e^{j2w_1} & \ldots & e^{j(M-1)w_1} \\ & & \vdots & & \\ 1 & e^{jw_{M-1}} & e^{j2w_{M-1}} & \ldots & e^{j(M-1)w_{M-1}} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1}E_1(z^M) \\ \vdots \\ z^{-(M-1)}E_{M-1}(z^M) \end{bmatrix}$$

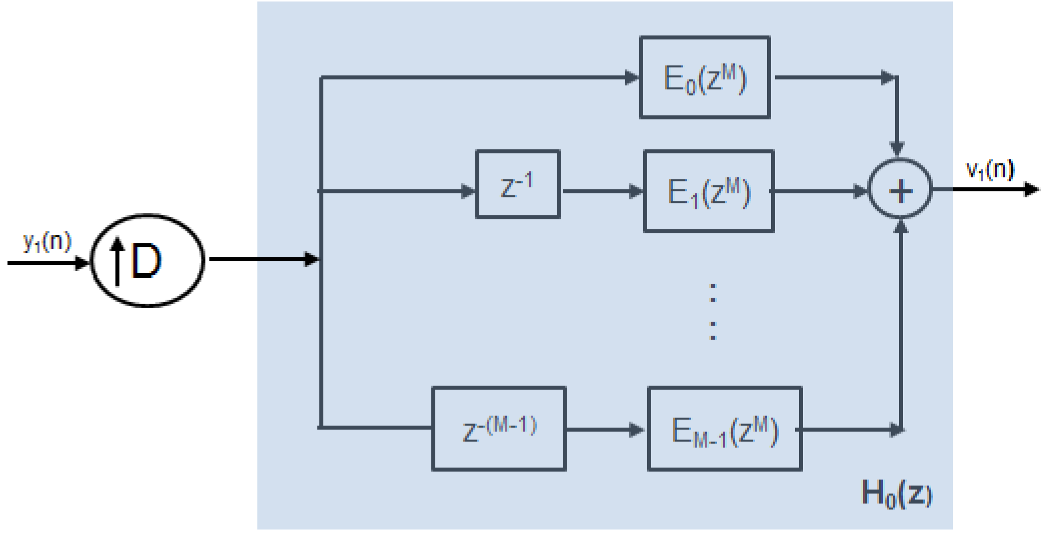
Here is the multirate noble identity for interpolation:



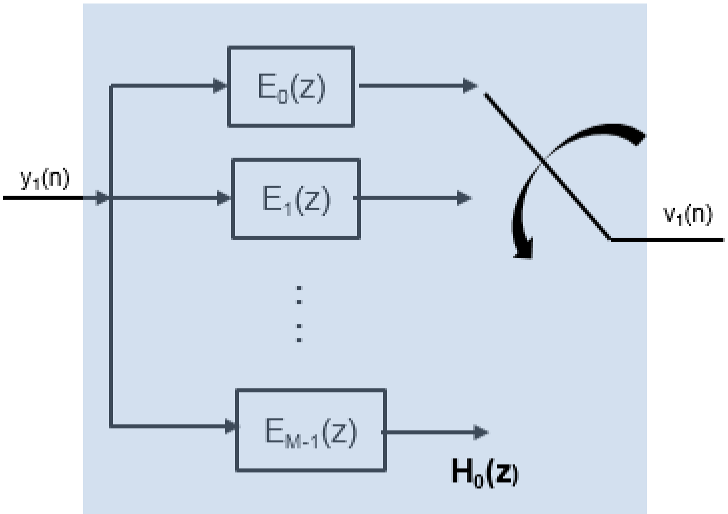
For illustration, consider the first branch of the filter bank that contains the lowpass filter.



Replace H_0(z) with its polyphase representation.



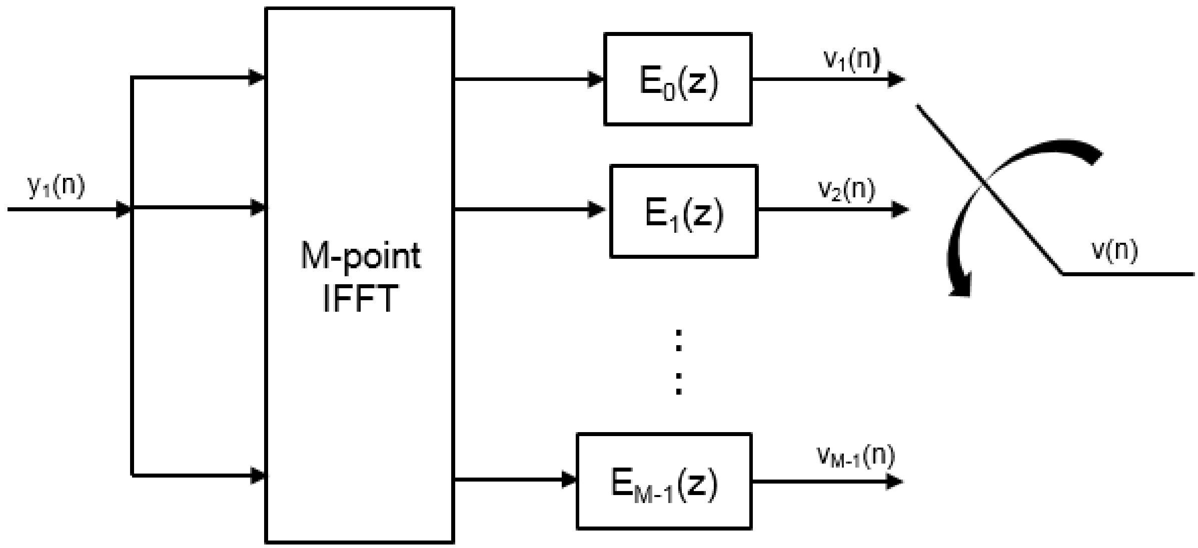
After applying the noble identity for interpolation, you can replace the delays, interpolation factor, and the adder with a commutator switch.



For all the M channels in the filter bank, the MIMO transfer function, $H(z)$, is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{jw_1} & e^{j2w_1} & \dots & e^{j(M-1)w_1} \\ & & \vdots & & \\ 1 & e^{jw_{M-1}} & e^{j2w_{M-1}} & \dots & e^{j(M-1)w_{M-1}} \end{bmatrix} \begin{bmatrix} E_0(z) \\ E_1(z) \\ \vdots \\ E_{M-1}(z) \end{bmatrix}$$

The matrix on the left is an IDFT matrix. With the IDFT matrix, the efficient implementation of the lowpass prototype based filter bank looks like the following.



References

[1] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.

[2] Harris, F.J., Chris Dick, Michael Rice. "Digital Receivers and Transmitters Using Polyphase Filter Banks for Wireless Communications." *IEEE Transactions on microwave theory and techniques*. Vol. 51, Number 4, April 2003.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Blocks

`Channelizer` | `Dyadic Synthesis Filter Bank` | `Two-Channel Synthesis Subband Filter`

System Objects

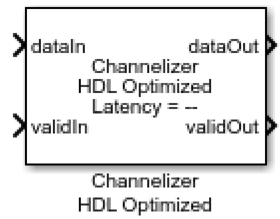
dsp.ChannelSynthesizer | dsp.Channelizer | dsp.SubbandSynthesisFilter | dsp.DyadicSynthesisFilterBank

Introduced in R2017a

Channelizer HDL Optimized

Polyphase filter bank and fast Fourier transform—optimized for HDL code generation

Library: DSP System Toolbox / Filtering / Multirate Filters
DSP System Toolbox HDL Support / Filtering



Description

The Channelizer HDL Optimized block separates a broadband input signal into multiple narrowband output signals. It provides hardware speed and area optimization for streaming data applications. The block accepts scalar or vector input of real or complex data, provides hardware-friendly control signals, and has optional output frame control signals. You can achieve giga-sample-per-second (GSPS) throughput using vector input. The block implements a polyphase filter, with one subfilter per input vector element. The hardware implementation interleaves the subfilters, which results in sharing each filter multiplier (*FFT Length* / *Input Size*) times. The FFT implementation uses the same pipelined Radix 2^2 FFT algorithm as the `FFT HDL Optimized` block.

Ports

Input

`dataIn` — Input data

scalar or column vector of real or complex values

The vector size must be a power of 2 that is from 1 to 64, and is not greater than the number of channels (FFT length). `double` and `single` input data are allowed for simulation but not for HDL code generation.

The block does not accept `uint64` data.

Data Types: `fixed_point` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `single` | `double`
Complex Number Support: Yes

`validIn` — Indicates valid input data
scalar

When **validIn** is `true`, the block captures the value on **dataIn**.

Data Types: `Boolean`

`reset` — Reset control signal (optional)
scalar

When **reset** is `true`, the block stops the current calculation and clears internal state.

Dependencies

To enable this port, select **Enable reset input port**.

Data Types: `Boolean`

Output

`dataOut` — Frequency channel output data
vector

- If you set **Output vector size** to `Same as number of frequency bands` (default), the output data is a 1-by-*M* vector where *M* is the FFT length. The output order is bit natural.
- If you set **Output vector size** to `Same as input size`, the output data is an *M*-by-1 vector where *M* is the input vector size. The output order is bit reversed.

The output data type is a result of the **Filter output data type** and the bit growth in the FFT necessary to avoid overflow.

`validOut` — Indicates valid output data
scalar

The block sets **validOut** to `true` with each valid sample on **dataOut**.

Data Types: `Boolean`

`startOut` — Indicates the first valid cycle of output data (optional)
scalar

The block sets **startOut** to `true` during the first valid sample on **dataOut**.

Dependencies

To enable this port, select **Enable start output port**.

Data Types: `Boolean`

`endOut` — Indicates the last valid cycle of output data (optional)
scalar

The block sets **endOut** to `true` during the last valid sample on **dataOut**.

Dependencies

To enable this port, select **Enable end output port**.

Data Types: `Boolean`

Parameters

Main

`Number of frequency bands (FFT length)` — FFT length
`8` (default) | integer power of two

For HDL code generation, the FFT length must be a power of 2 from 2^3 to 2^{16}.

`Filter coefficients` — Polyphase filter coefficients
`[ -0.032, 0.121, 0.318, 0.482, 0.546, 0.482, 0.318, 0.121, -0.032 ]` (default) | vector of numeric values

If the number of coefficients is not a multiple of **Number of frequency bands (FFT length)**, the block pads this vector with zeros. The default filter specification is a raised-cosine FIR filter, `rcosdesign(0.25,2,4,'sqrt')`. You can specify a vector of coefficients or a call to a filter design function that returns the coefficient values.

Complex coefficients are not supported. By default, the block casts the coefficients to the same data type as the input.

`Complex multiplication` — HDL implementation of complex multipliers

`Use 3 multipliers and 5 adders` (default) | `Use 4 multipliers and 2 adders`

Each multiplication is implemented with either 4 multipliers and 2 adders, or 3 multipliers and 5 adders. Which option is faster or smaller depends on your synthesis tool and target device.

`Output vector size` — Size and order of output data

`Same as number of frequency bands` (default) | `Same as input size`

The output data is a row vector of M-by-1 channels.

- `Same as number of frequency bands` — Output data is a 1-by-M vector, where M is the FFT length. The output order is bit natural.
- `Same as input size` — Output data is an M-by-1 vector, where M is the input vector size. The output order is bit reversed.

`Divide butterfly outputs by two` — FFT scaling

on (default) | off

When you select this parameter, the FFT implements an overall $1/N$ scale factor by scaling the result of each pipeline stage by 2. This adjustment keeps the output of the FFT in the same amplitude range as its input. If scaling is disabled, the FFT avoids overflow by increasing the word length by 1 bit at each stage.

Data Types

`Rounding mode` — Rounding method used for internal fixed-point calculations

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

See “Rounding Modes”. The block uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input is `single` or `double`. Each FFT stage rounds after the twiddle factor multiplication but before the butterflies. Rounding can also occur when casting the coefficients and the output of the polyphase filter to the data types you specify.

`Overflow mode` — Overflow handling for internal fixed-point calculations

`Wrap` (default) | `Saturate`

See “Overflow Handling”. The block uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input is `single` or `double`. This option applies to casting the coefficients and the output of the polyphase filter to the data types you specify.

The FFT algorithm avoids overflow by either scaling the output of each stage (`Normalize` enabled), or by increasing the word length by 1 bit at each stage (`Normalize` disabled).

`Coefficient data type` — Data type of the filter coefficients

`Inherit: Same word length as input` (default) | data type expression

The block casts the polyphase filter coefficients to this data type, using the rounding and overflow settings you specify. When you select `Inherit: Same word length as input` (default), the block selects the binary point using `fi()` best-precision rules.

`Filter output data type` — Data type of the output of the polyphase filter

`Inherit: Same word length as input` (default) | data type expression

The block casts the output of the polyphase filter (the input to the FFT) to this data type, using the rounding and overflow settings you specify. When you select `Inherit: Same word length as input` (default), the block selects a best-precision binary point by considering the values of your filter coefficients and the range of your input data type.

By default, the FFT logic does not modify the data type. When you disable **Divide butterfly outputs by two**, the FFT increases the word length by 1 bit at each stage to avoid overflow.

Control Ports

`Enable reset input port` — Optional reset signal

off (default) | on

When you select this parameter, the **reset** port shows on the block icon. When the **reset** input is `true`, the block stops calculation and clears all internal state.

`Enable start output port` — Optional control signal indicating start of data

off (default) | on

When you select this parameter, the **startOut** port shows on the block icon. The **startOut** signal is true for the first cycle of output data in a frame.

`Enable end output port` — Optional control signal indicating end of data
off (default) | on

When you select this parameter, the **endOut** port shows on the block icon. The **endOut** signal is true for the last cycle of output data in a frame.

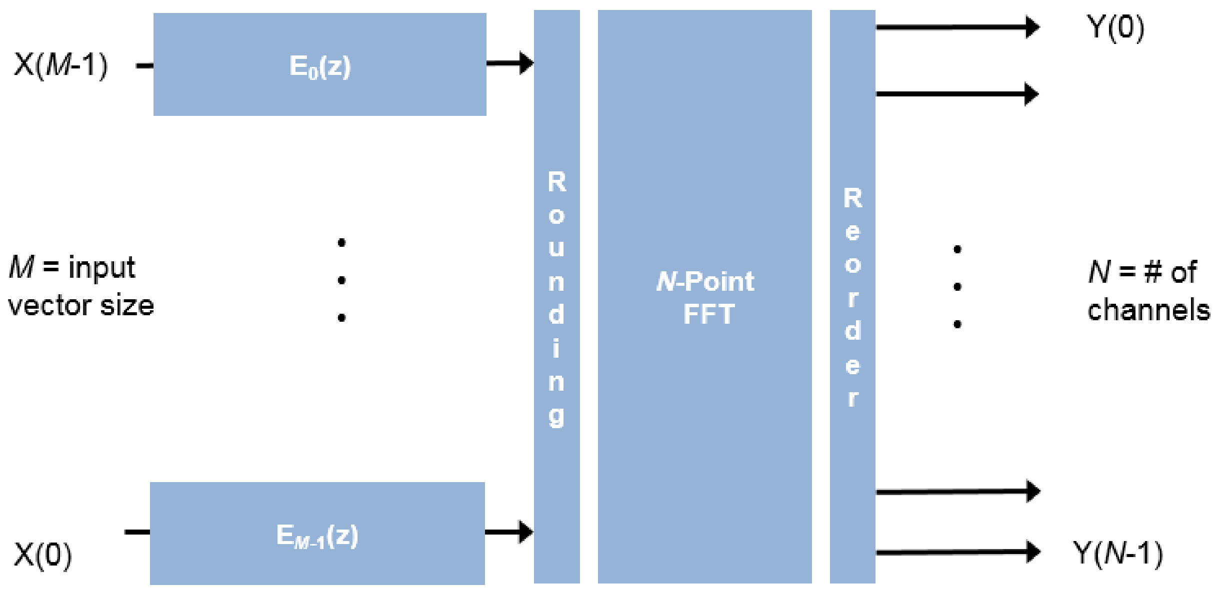
Model Examples

Algorithm

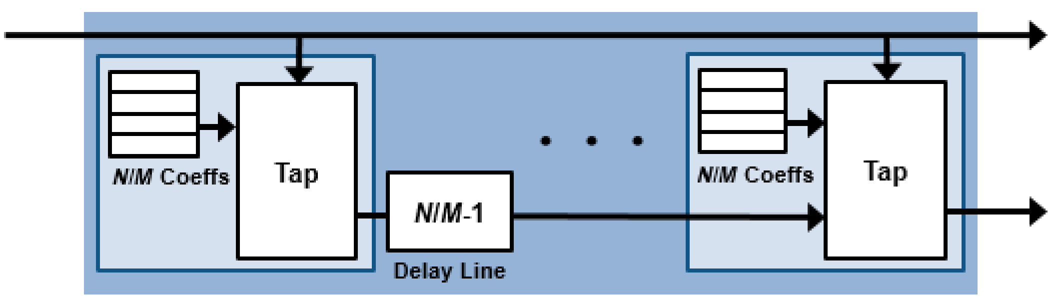
The polyphase filter algorithm requires a subfilter for each FFT channel. For more detail on the polyphase filter architecture, refer to [1], and to the `Channelizer` block reference page.

Note: The output of the Channelizer HDL Optimized block does not match the output from the Channelizer block sample-for-sample. This mismatch is because the blocks apply the input samples to the subfilters in different orders. The Channelizer HDL Optimized block applies input X(0) to subfilter $E_{M-1}(z)$, X(1) to subfilter $E_{M-2}(z)$, ..., X(M-1) to subfilter $E_0(z)$. The channels detected by both blocks match, when analyzed over multiple frames.

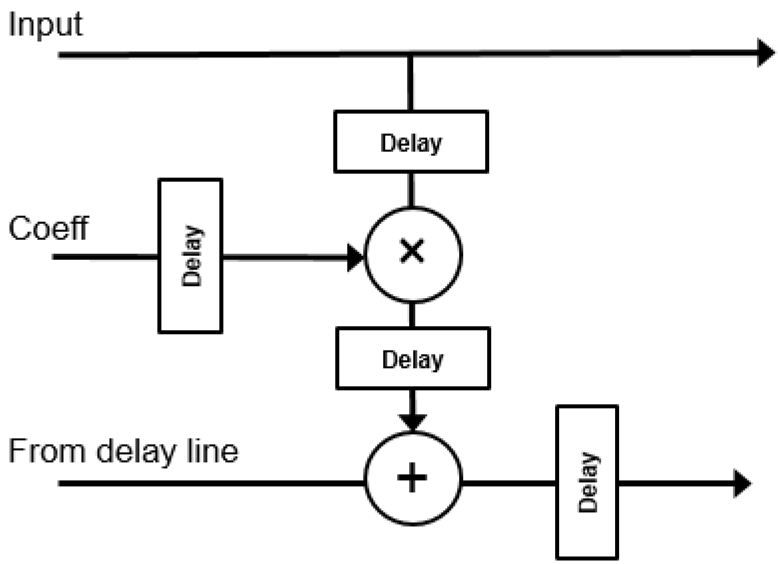
If the input vector size, M, is the same as the FFT length, N, then the block implements N subfilters in the hardware. Each subfilter is a direct-form transposed FIR filter with $NumCoeffs/N$ taps.



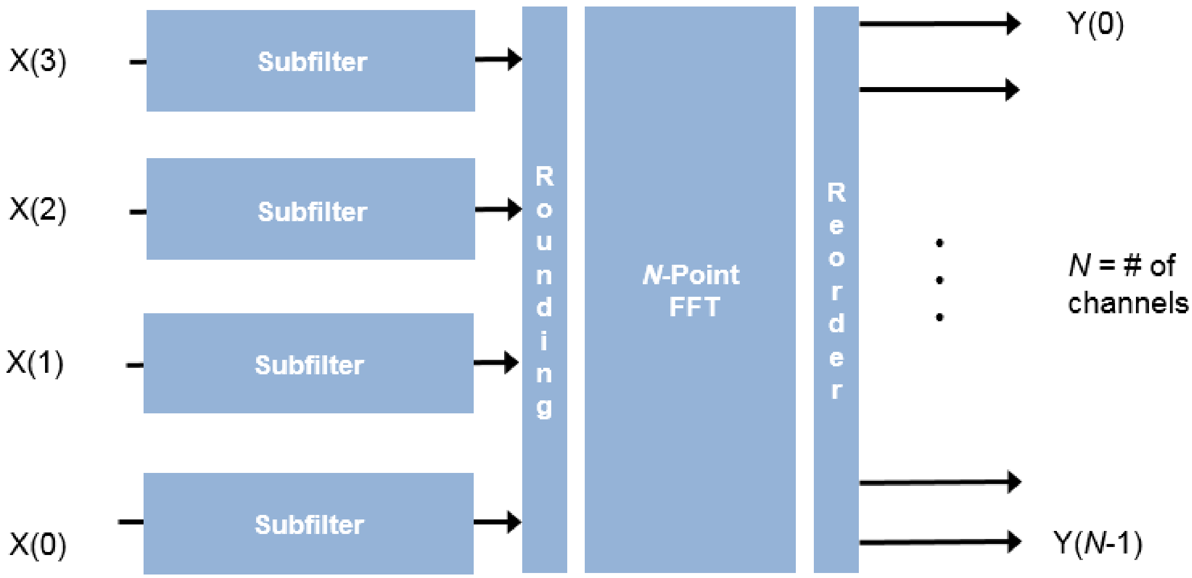
If the vector size is less than *N*, the block implements one subfilter for each input vector element. The subfilter multipliers are shared as necessary to implement *N* channel filters. The shared multiplier taps have a lookup table for *N*/*M* filter coefficients. Each tap is followed by a delay line of *N*/*M*–1 cycles.



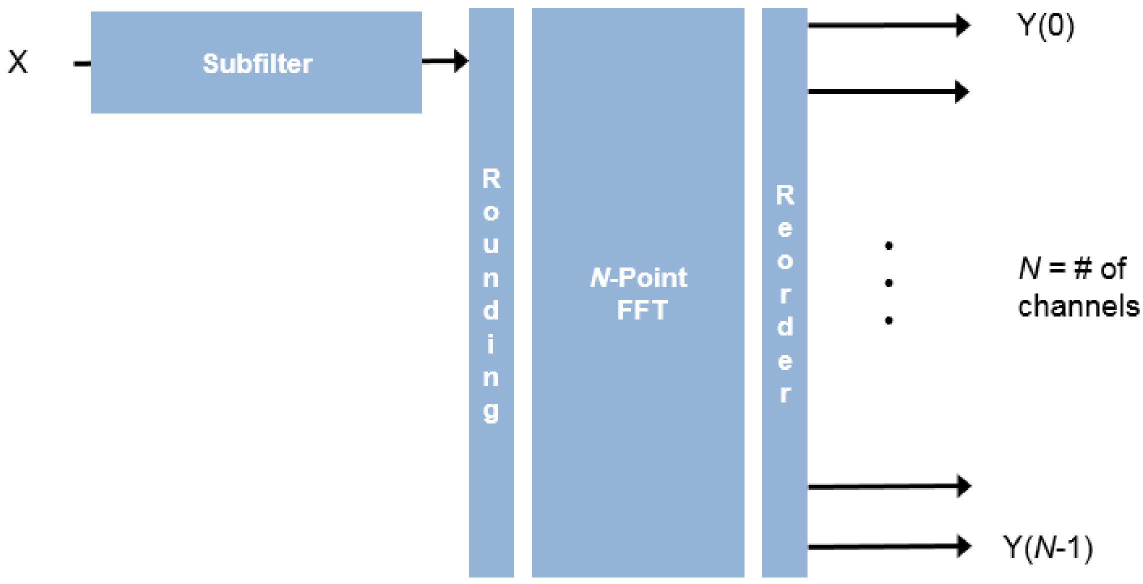
The output of the subfilters is cast to the specified **Filter output data type**, using the rounding and overflow settings you chose. Each filter tap in the subfilter is pipelined to target the DSP sections of an FPGA.



For instance, for an FFT length of 8, and an input vector size of 4, the block implements four filters. Each multiplier is shared *N*/*M* times, or twice. Each tap applies two coefficients, and the delay line is *N*/*M*–1 cycles.



For scalar input, the block implements one filter. Each multiplier is shared N times. Each tap applies N coefficients, and the delay line is N–1 cycles.



Latency

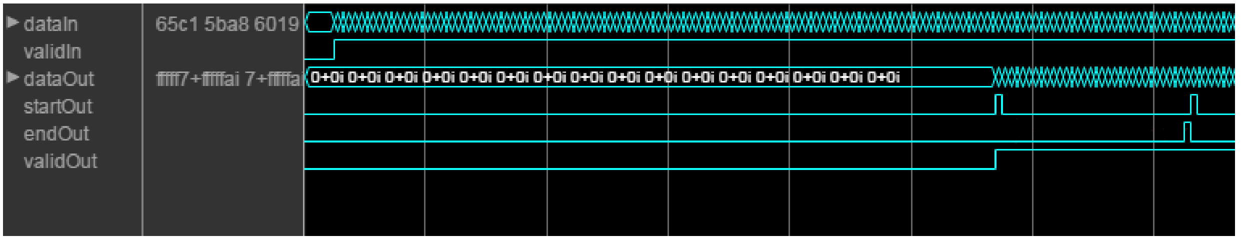
The latency varies with FFT length and vector size. After you update the model, the latency is displayed on the block icon. The displayed latency is the number of cycles between the first valid input and the first valid output, assuming that the input is contiguous. The filter coefficients do not affect the latency. Setting the output size equal to the input size reduces the latency, because the samples are not saved and reordered.

Control Signals

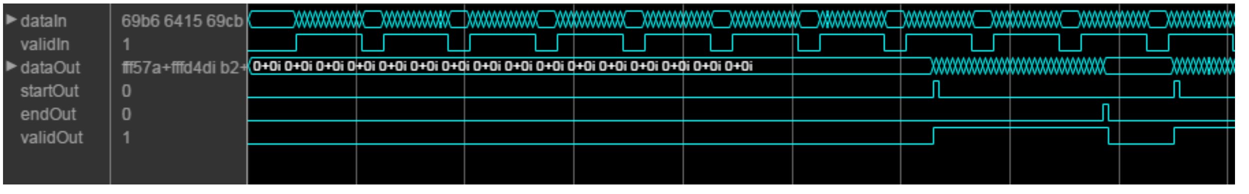
This diagram shows `validIn` and `validOut` signals for contiguous input data with a vector size of 16 and an FFT length of 512.

The diagram also shows the optional `startOut` and `endOut` signals that indicate frame boundaries. If you enable `startOut`, it pulses for one cycle with the first `validOut` of the frame. If you enable `endOut`, it pulses for one cycle with the last `validOut` of the frame.

If you apply continuous input frames, the output will also be continuous, after the initial latency.



The `validIn` signal can be noncontiguous. Data accompanied by a `validIn` signal is stored until a frame is filled, and output in a contiguous frame of *N* (FFT length) cycles. This diagram shows noncontiguous input and contiguous output for an FFT length of 512 and a vector size of 16 samples.



Performance

These resource and performance data are the place-and-route results from the generated HDL targeted to a Xilinx® Virtex® 6 (XC6VLX240-1ff784) FPGA. The three examples in the tables use this configuration:

- FFT length — 8
- Filter length — 96 coefficients
- 16-bit complex input data
- Coefficient and filter output data types same word length as input data (default)
- Complex multiplication (default) — 4 multipliers, 2 adders
- Output scaling — Enabled
- Minimize clock enables (HDL Coder parameter)

Performance of the synthesized HDL code varies with your target and synthesis options.

For scalar input, the design achieves a clock frequency of 346 MHz. The latency is 53 cycles. The subfilters share each multiplier eight (*N*) times. The design uses these resources.

Resource	Uses
LUT	1591
FFS	2681
Xilinx LogiCORE® DSP48	16

For four-sample vector input, the design achieves a clock frequency of 333 MHz. The latency is 31 cycles. The subfilters share each multiplier twice (*N*/*M*). The design uses these resources.

Resource	Uses
LUT	1912
FFS	3986
Xilinx LogiCORE DSP48	56

For eight-sample vector input, the design achieves a clock frequency of 292 MHz. The latency is 20 cycles. When the input size is the same as the FFT length, the subfilters do not share any multipliers. The design uses these resources.

Resource	Uses
LUT	1388
FFS	2302
Xilinx LogiCORE DSP48	110

References

[1] Harris, F. J., C. Dick, and M. Rice. “Digital Receivers and Transmitters Using Polyphase Filter Banks for Wireless Communications.” *IEEE Transactions on Microwave Theory and Techniques*. Vol. 51, No. 4, April 2003.

See Also

See Also

Blocks
`Channelizer` | `FFT HDL Optimized`

System Objects
dsp.HDLChannelizer

Topics
"High Throughput HDL Algorithms"

Introduced in R2017a

Check Signal Attributes

Error when input signal does or does not match selected attributes exactly

Library

Signal Management / Signal Attributes

`dspsigattribs`

Description

The Check Signal Attributes block terminates the simulation with an error when the input characteristics differ from the characteristics you specify in the block parameters.

When you set **Error when input** to `Does not match attributes exactly`, the block generates an error when the input fails to match *any* of the specified attributes. Only signals that possess *all* of the specified attributes propagate to the output unaltered and do not cause the block to generate an error.

When you set **Error when input** to `Matches attributes exactly`, the block generates an error only when the input possesses *all* specified attributes. Signals that do not possess *all* of the specified attributes propagate to the output unaltered, and do not cause the block to generate an error.

Signal Attributes

The Check Signal Attributes block can test for up to five different signal attributes, as specified by the following parameters. When you select `Ignore` for any parameter, the block does not check the signal for the corresponding attribute. For example, when you set **Complexity** to `Ignore`, neither real nor complex inputs cause the block to generate an error. The attributes are:

- **Complexity**

Check whether the input is real or complex. You can display this information in a model by attaching a Probe block with **Probe complex signal** selected. Alternatively, you can select **Port Data Types** from the **Display** > **Signals & Ports** menu.

- **Dimensionality**

  Check the dimensionality of the input for compliance or noncompliance with the attributes in the subordinate **Dimension** menu. See the following table. *M* and *N* are positive integers unless otherwise indicated.

Dimensions	Is...	Is not...
1-D	1-D vector, 1-D scalar	*M*-by-*N* matrix, 1-by-*N* matrix (row vector), *M*-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar)
2-D	*M*-by-*N* matrix, 1-by-*N* matrix (row vector), *M*-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar)	1-D vector, 1-D scalar
Scalar (1-D or 2-D)	1-D scalar, 1-by-1 matrix (2-D scalar)	1-D vector with length>1, *M*-by-*N* matrix with *M*>1 and/or *N*>1
Vector (1-D or 2-D)	1-D vector, 1-D scalar, 1-by-*N* matrix (row vector), *M*-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar) Vector (1-D or 2-D) or scalar	*M*-by-*N* matrix with *M*>1 and *N*>1
Row Vector (2-D)	1-by-*N* matrix (row vector), 1-by-1 matrix (2-D scalar) Row vector (2-D) or scalar	1-D vector, 1-D scalar, *M*-by-*N* matrix with *M*>1
Column Vector (2-D)	*M*-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar) Column vector (2-D) or scalar	1-D vector, 1-D scalar, *M*-by-*N* matrix with *N*>1
Full matrix	*M*-by-*N* matrix with *M*>1 and *N*>1	1-D vector, 1-D scalar, 1-by-*N* matrix (row vector),

Dimensions	Is...	Is not...
		M-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar)
Square matrix	*M*-by-*N* matrix with *M*=*N*, 1-D scalar, 1-by-1 matrix (2-D scalar	*M*-by-*N* matrix with *M*≠*N*, 1-D vector, 1-by-*N* matrix (row vector), *M*-by-1 matrix (column vector)

When you select **Signal Dimensions** from the **Display** > **Signals & Ports** menu, Simulink displays the size of a 1-D vector signal as an unbracketed integer, and displays the dimension of a 2-D signal as a pair of bracketed integers, `[MxN]`. Simulink *does not display* any size information for a 1-D or 2-D scalar signal. You can also display dimension information for a signal in a model by attaching a Probe block with **Probe signal dimensions** selected.

- **Data type**

  Check the signal data type for compliance `(Is...)` or noncompliance `(Is not...)` with the attributes in the subordinate **General data type** menu. See the following table. You can individually select any of the specific data types listed in the `(Is...)` column from the subordinate **Specific data type** menu.

General Data Type	Is...	Is not...
Boolean	`boolean`	`single, double, uint8, int8, uint16, int16, uint32, int32`, fixed point, enumerated
Enumerated	A user-defined enumerated data type. See “Data Types” (Simulink) in the Simulink documentation.	`boolean, single, double, uint8, int8, uint16, int16, uint32, int32`, fixed point
Floating point	`single, double`	`boolean, uint8, int8, uint16, int16, uint32, int32`, fixed point, enumerated
Floating point or Boolean	`single, double, boolean`	`uint8, int8, uint16, int16, uint32, int32`, fixed point, enumerated

General Data Type	Is...	Is not...
Fixed point	fixed point, `uint8, int8, uint16, int16, uint32, int32`	`boolean, single, double`, enumerated
Integer	Signed integer `int8`, `int16`, `int32` Unsigned integer `uint8`, `uint16`, `uint32`	`boolean, single, double`, fixed point, enumerated

To display data type information, in your model window, from the **Display** menu, point to **Signals & Ports** and select **Port Data Types**.

- **Sample time**

  Check whether the signal is discrete time or continuous time. When you select **Colors** from the **Display** > **Sample Time** menu, Simulink displays continuous-time signal lines in black or grey and discrete-time signal lines in colors corresponding to the relative rate.

  When you attach a Probe block with **Probe sample time** enabled to a continuous-time signal, the block icon displays `Ts:[0 To]`, where `To` is the sample time offset. Valid values of `To` for continuous-time signals are `0` and `1`. When `To` is `0`, updates occur at every major and minor time step. When `To` is `1`, updates occur only at major time steps and the sample time is *fixed in minor time step*.

  When you attach a Probe block with **Probe sample time** enabled to a discrete-time signal, the block icon displays `Ts:[Ts To]` for sample-based signals, and `Tf:[Tf To]` for frame-based signals. `Ts` and `Tf` are the positive sample period and frame period, respectively. `To` is the offset, such that 0 ≤ *`offset`* < *`period`*. Frame-based signals are almost always discrete time.

Parameters

Error when input

Specify whether the block generates an error when the input *does* or *does not* possess *all* of the required attributes.

Complexity

Specify the complexity for which you want to check the input, `Real` or `Complex`. When you select `Ignore` from the list, the block does not check the complexity of the input.

Dimensionality

Specify whether you want to check the input for compliance or noncompliance with the attributes in the subordinate **Dimensions** menu. When you select `Ignore` from the list, the block does not check the dimensionality of the input.

Dimensions

Specify the dimensions for which you want to check the input. This parameter is only visible when you set the **Dimensionality** parameter to `Is...` or `Is not...`.

Data type

Specifies whether you want to check the input for compliance or noncompliance with the attributes in the subordinate **General data type** menu. When you select `Ignore` from the list, the block does not check the input data type.

General data type

Specify the general data type for which you want to check the input. This parameter is only visible when you set the **Data type** to `Is...` or `Is not...`.

Specific floating-point

Specify the floating-point data type for which you want to check the input. This parameter is only visible when you set the **General data type** to `Floating-point` or `Floating-point or boolean`.

Specific fixed-point

Specify the fixed-point data type for which you want to check the input. This parameter is only visible when you set the **General data type** to `Fixed-point`.

Specific integer

Specify the integer data type for which you want to check the input. This parameter is only visible when you set the **General data type** to `Integer`.

Sample time

Specify the sample time for which you want to check the input, `Discrete` or `Continuous`. When you select `Ignore` from the list, the block does not check the sample time of the input.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8, 16, and 32-bit signed integers • 8, 16, and 32-bit unsigned integers • Enumerated
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8, 16, and 32-bit signed integers • 8, 16, and 32-bit unsigned integers • Enumerated

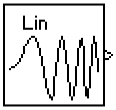
See Also

`Buffer`	DSP System Toolbox
`Convert 1-D to 2-D`	DSP System Toolbox
`Convert 2-D to 1-D`	DSP System Toolbox
Data Type Conversion	Simulink
`Inherit Complexity`	DSP System Toolbox
Probe	Simulink
Reshape	Simulink
`Submatrix`	DSP System Toolbox

Introduced before R2006a

Chirp

Generate swept-frequency cosine (chirp) signal



Library

Sources

`dspsrcs4`

Description

The Chirp block outputs a swept-frequency cosine (chirp) signal with unity amplitude and continuous phase. To specify the desired output chirp signal, you must define its instantaneous frequency function, also known as the output frequency sweep. The frequency sweep can be linear, quadratic, or logarithmic, and repeats once every **Sweep time** by default. See other sections of this reference page for more details about the block.

Sections of This Reference Page

- Variables Used in This Reference Page
- “Shaping the Frequency Sweep by Setting Frequency Sweep and Sweep Mode” on page 2-252
- “Unidirectional and Bidirectional Sweep Modes” on page 2-253
- “Setting Instantaneous Frequency Sweep Values” on page 2-254
- “Block Computation Methods” on page 2-255
- “Cautions Regarding the Swept Cosine Sweep” on page 2-257
- “Parameters” on page 2-258
- “Examples” on page 2-259
- “Supported Data Types” on page 2-269

- "See Also" on page 2-269

Variables Used in This Reference Page

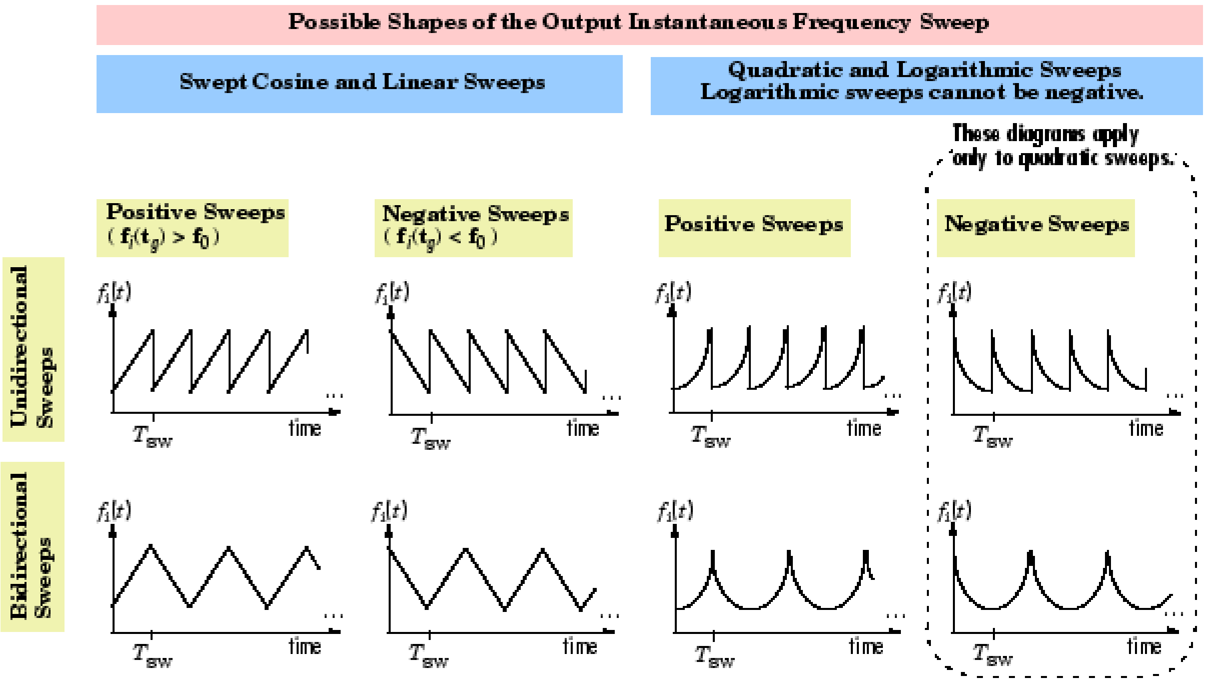
f_0	**Initial frequency** parameter (Hz)
$f_i(t_g)$	**Target frequency** parameter (Hz)
t_g	**Target time** parameter (seconds)
T_{sw}	**Sweep time** parameter (seconds)
ϕ_0	**Initial phase** parameter (radians)
$\psi(t)$	Phase of the chirp signal (radians)
$f_i(t)$	User-specified output instantaneous frequency function (Hz); user-specified sweep
$f_{i(actual)}(t)$	Actual output instantaneous frequency function (Hz); actual output sweep
$y_{chirp}(t)$	Output chirp function

Shaping the Frequency Sweep by Setting Frequency Sweep and Sweep Mode

The basic shape of the output instantaneous frequency sweep, $f_i(t)$, is set by the **Frequency sweep** and **Sweep mode** parameters, described in the following table.

Parameters for Setting Sweep Shape	Possible Setting	Parameter Description
Frequency sweep	Linear Quadratic Logarithmic Swept cosine	Determines whether the sweep frequencies vary linearly, quadratically, or logarithmically. Linear and swept cosine sweeps both vary linearly.
Sweep mode	Unidirectional Bidirectional	Determines whether the sweep is unidirectional or bidirectional. For details, see "Unidirectional and Bidirectional Sweep Modes" on page 2-253

The following diagram illustrates the possible shapes of the frequency sweep that you can obtain by setting the **Frequency sweep** and **Sweep mode** parameters.



For information on how to set the frequency values in your sweep, see "Setting Instantaneous Frequency Sweep Values" on page 2-254.

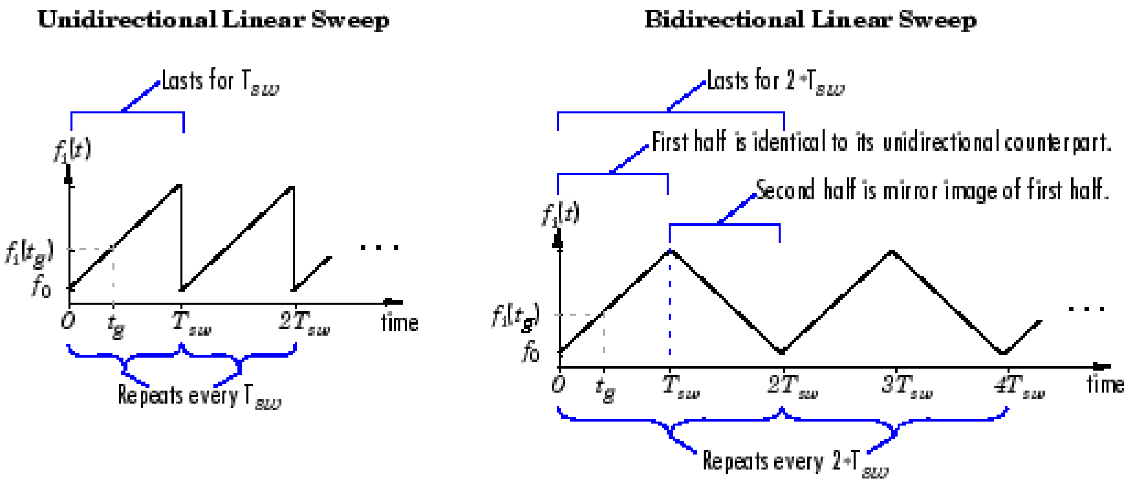
Unidirectional and Bidirectional Sweep Modes

The **Sweep mode** parameter determines whether your sweep is unidirectional or bidirectional, which affects the shape of your output frequency sweep (see "Shaping the Frequency Sweep by Setting Frequency Sweep and Sweep Mode" on page 2-252). The following table describes the characteristics of unidirectional and bidirectional sweeps.

Sweep Mode Parameter Settings	Sweep Characteristics
`Unidirectional`	• Lasts for one **Sweep time**, T_{sw} • Repeats once every T_{sw}
`Bidirectional`	• Lasts for twice the **Sweep time**, $2*T_{sw}$ • Repeats once every $2*T_{sw}$

Sweep Mode Parameter Settings	Sweep Characteristics
	• First half is identical to its unidirectional counterpart. • Second half is a mirror image of the first half.

The following diagram illustrates a linear sweep in both sweep modes. For information on setting the frequency values in your sweep, see “Setting Instantaneous Frequency Sweep Values” on page 2-254.



Setting Instantaneous Frequency Sweep Values

Set the following parameters to tune the frequency values of your output frequency sweep.

- **Initial frequency** (Hz), f_0
- **Target frequency** (Hz), $f_i(t_g)$
- **Target time** (seconds), t_g

The following table summarizes the sweep values at specific times for all **Frequency sweep** settings. For information on the formulas used to compute sweep values at other times, see “Block Computation Methods” on page 2-255.

Instantaneous Frequency Sweep Values

Frequency Sweep	Sweep Value at $t = 0$	Sweep Value at $t = t_g$	Time when Sweep Value Is Target Frequency, $f_i(t_g)$
Linear	f_0	$f_i(t_g)$	t_g
Quadratic	f_0	$f_i(t_g)$	t_g
Logarithmic	f_0	$f_i(t_g)$	t_g
Swept cosine	f_0	$2f_i(t_g)$ - f_0	$t_g/2$

Block Computation Methods

The Chirp block uses one of two formulas to compute the block output, depending on the **Frequency Sweep** parameter setting. For details, see the following sections:

- "Equations for Output Computation" on page 2-255
- "Output Computation Method for Linear, Quadratic, and Logarithmic Frequency Sweeps" on page 2-256
- "Output Computation Method for Swept Cosine Frequency Sweep" on page 2-257

Equations for Output Computation

The following table shows the equations used by the block to compute the user-specified output frequency sweep, $f_i(t)$, the block output, $y_{chirp}(t)$, and the actual output frequency sweep, $f_{i(actual)}(t)$. The only time the user-specified sweep is not the actual output sweep is when the **Frequency sweep** parameter is set to `Swept cosine`.

Note The following equations apply only to unidirectional sweeps in which $f_i(0) < f_i(t_g)$. To derive equations for other cases, you might find it helpful to examine the following table and the diagram in "Shaping the Frequency Sweep by Setting Frequency Sweep and Sweep Mode" on page 2-252.

The table below contains the following variables:

- $f_i(t)$ — the user-specified frequency sweep
- $f_{i(actual)}(t)$ — the actual output frequency sweep, usually equal to $f_i(t)$

- $y(t)$ — the Chirp block output
- $\psi(t)$ — the phase of the chirp signal, where $\psi(0) = 0$, and $2\pi f_i(t)$ is the derivative of the phase

  $$f_i(t) = \frac{1}{2\pi} \cdot \frac{d\psi(t)}{dt}$$

- ϕ_0 — the **Initial phase** parameter value, where $y_{chirp}(0) = \cos(\phi_0)$

Equations Used by the Chirp Block for Unidirectional Positive Sweeps

Frequency Sweep	Block Output Chirp Signal	User-Specified Frequency Sweep, $f_i(t)$	β	Actual Frequency Sweep, $f_{i(actual)}(t)$
`Linear`	$y(t) = \cos(\psi(t) + \phi_0)$	$f_i(t) = f_0 + \beta t$	$\beta = \frac{f_i(t_g) - f_0}{t_g}$	$f_{i(actual)}{}^{(t)} = f_i^{(t)}$
`Quadratic`	Same as `Linear`	$f_i(t) = f_0 + \beta t^2$	$\beta = \frac{f_i(t_g) - f_0}{t_g^2}$	$f_{i(actual)}{}^{(t)} = f_i^{(t)}$
`Logarithmic`	Same as `Linear`	$F_i(t) = f_0 \left(\frac{f_i(t_g)}{f_0} \right)^{\frac{t}{t_g}}$ Where $f_i(t_g) > f_0 > 0$	N/A	$f_{i(actual)}(t) = f_i(t)$
`Swept cosine`	$y(t) = \cos(2\pi f_i(t)t + \phi_0)$	Same as `Linear`	Same as `Linear`	$f_{i(actual)}(t) = f_i(t) + \beta t$

Output Computation Method for Linear, Quadratic, and Logarithmic Frequency Sweeps

The derivative of the phase of a chirp function gives the instantaneous frequency of the chirp function. The Chirp block uses this principle to calculate the chirp output when the **Frequency Sweep** parameter is set to `Linear`, `Quadratic`, or `Logarithmic`.

$y_{chirp}(t) = \cos(\psi(t) + \phi_0)$	Linear, quadratic, or logarithmic chirp signal with phase $\psi(t)$

$$f_i(t) = \frac{1}{2\pi} \cdot \frac{d\psi(t)}{dt}$$

Phase derivative is instantaneous frequency

For instance, if you want a chirp signal with a linear instantaneous frequency sweep, you should set the **Frequency Sweep** parameter to `Linear`, and tune the linear sweep values by setting other parameters appropriately. The block outputs a chirp signal, the phase derivative of which is the specified linear sweep. This ensures that the instantaneous frequency of the output is the linear sweep you desired. For equations describing the linear, quadratic, and logarithmic sweeps, see “Equations for Output Computation” on page 2-255.

Output Computation Method for Swept Cosine Frequency Sweep

To generate the swept cosine chirp signal, the block sets the swept cosine chirp output as follows.

$$y_{chirp}(t) = \cos(\psi(t) + \phi_0) = \cos(2\pi f_i(t)t + \phi_0)$$

Swept cosine chirp output (Instantaneous frequency equation, shown above, does not hold.)

Note that the instantaneous frequency equation, shown above, does not hold for the swept cosine chirp, so the user-defined frequency sweep, $f_i(t)$, is not the actual output frequency sweep, $f_{i(actual)}(t)$, of the swept cosine chirp. Thus, the swept cosine output might not behave as you expect. To learn more about swept cosine chirp behavior, see “Cautions Regarding the Swept Cosine Sweep” on page 2-257 and “Equations for Output Computation” on page 2-255.

Cautions Regarding the Swept Cosine Sweep

When you want a linearly swept chirp signal, we recommend you use a linear frequency sweep. Though a swept cosine frequency sweep also yields a linearly swept chirp signal, the output might have unexpected frequency content. For details, see the following two sections.

Swept Cosine Instantaneous Output Frequency at the Target Time is not the Target Frequency

The swept cosine sweep value at the **Target time** is not necessarily the **Target frequency**. This is because the user-specified sweep is not the actual frequency sweep of the swept cosine output, as noted in "Output Computation Method for Swept Cosine Frequency Sweep" on page 2-257. See the table Instantaneous Frequency Sweep Values for the actual value of the swept cosine sweep at the **Target time**.

Swept Cosine Output Frequency Content May Greatly Exceed Frequencies in the Sweep

In **Swept cosine** mode, you should not set the parameters so that $1/T\mathrm{sw}$ is very large compared to the values of the **Initial frequency** and **Target frequency** parameters. In such cases, the actual frequency content of the swept cosine sweep might be closer to $1/T\mathrm{sw}$, far exceeding the **Initial frequency** and **Target frequency** parameter values.

Parameters

Frequency sweep

The type of output instantaneous frequency sweep, $f_i(t)$: `Linear`, `Logarithmic`, `Quadratic`, or `Swept cosine`.

Sweep mode

The directionality of the chirp signal: `Unidirectional` or `Bidirectional`.

Initial frequency (Hz)

For `Linear`, `Quadratic`, and `Swept cosine` sweeps, the initial frequency, f_0, of the output chirp signal. For `Logarithmic` sweeps, **Initial frequency** is one less than the actual initial frequency of the sweep. Also, when the sweep is `Logarithmic`, you must set the **Initial frequency** to be less than the **Target frequency**. Tunable (Simulink).

Target frequency (Hz)

For `Linear`, `Quadratic`, and `Logarithmic` sweeps, the instantaneous frequency, $f_i(t_g)$, of the output at the **Target time**, t_g. For a `Swept cosine` sweep, **Target frequency** is the instantaneous frequency of the output at half the **Target time**,

$t_g/2$. When **Frequency sweep** is `Logarithmic`, you must set the **Target frequency** to be greater than the **Initial frequency**. Tunable (Simulink).

Target time (s)

For `Linear`, `Quadratic`, and `Logarithmic` sweeps, the time, t_g, at which the **Target frequency**, $f_i(t_g)$, is reached by the sweep. For a `Swept cosine` sweep, **Target time** is the time at which the sweep reaches $2f_i(t_g) - f_0$. You must set **Target time** to be *no greater than* **Sweep time** , $T_{sw} \geq t_g$. Tunable (Simulink).

Sweep time (s)

In `Unidirectional` **Sweep mode**, the **Sweep time**, T_{sw}, is the period of the output frequency sweep. In `Bidirectional` **Sweep mode**, the **Sweep time** is half the period of the output frequency sweep. You must set **Sweep time** to be no less than **Target time**, $T_{sw} \geq t_g$. Tunable (Simulink).

Initial phase (rad)

The phase, ϕ_0 , of the cosine output at t=0; $y_{chirp}(t) = \cos(\phi_0)$. Tunable (Simulink).

Sample time

The sample period, T_s, of the output. The output frame period is $M_o{*}T_s$.

Samples per frame

The number of samples, M_o, to buffer into each output frame.

Output data type

The data type of the output, single-precision or double-precision.

Examples

The first few examples demonstrate how to use the Chirp block's main parameters, how to view the output in the time domain, and how to view the output spectrogram:

- “Example 1: Setting a Final Frequency Value for Unidirectional Sweeps” on page 2-260
- “Example 2: Bidirectional Sweeps” on page 2-263
- “Example 3: When Sweep Time is Greater Than Target Time” on page 2-264

Examples 4 and 5 illustrate Chirp block settings that might produce unexpected outputs:

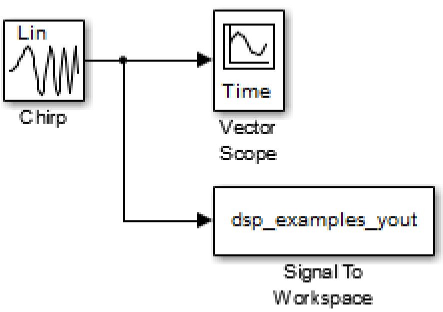
- "Example 4: Output Sweep with Negative Frequencies" on page 2-266
- "Example 5: Output Sweep with Frequencies Greater Than Half the Sampling Frequency" on page 2-267

Example 1: Setting a Final Frequency Value for Unidirectional Sweeps

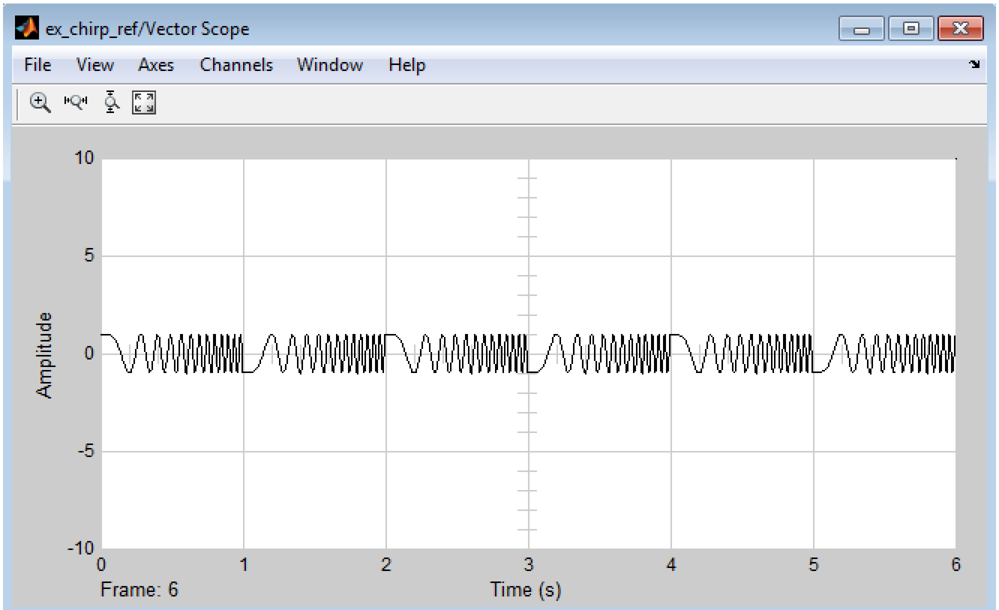
Often times, you might want a unidirectional sweep for which you know the initial and final frequency values. You can specify the final frequency of a unidirectional sweep by setting **Target time** equal to **Sweep time**, in which case the **Target frequency** becomes the final frequency in the sweep. The following model demonstrates this method.

This technique might not work for swept cosine sweeps. For details, see "Cautions Regarding the Swept Cosine Sweep" on page 2-257.

Open the Example 1 model by typing ex_chirp_ref at the MATLAB command line. You can also rebuild the model yourself; see the following list for model parameter settings (leave unlisted parameters in their default states).

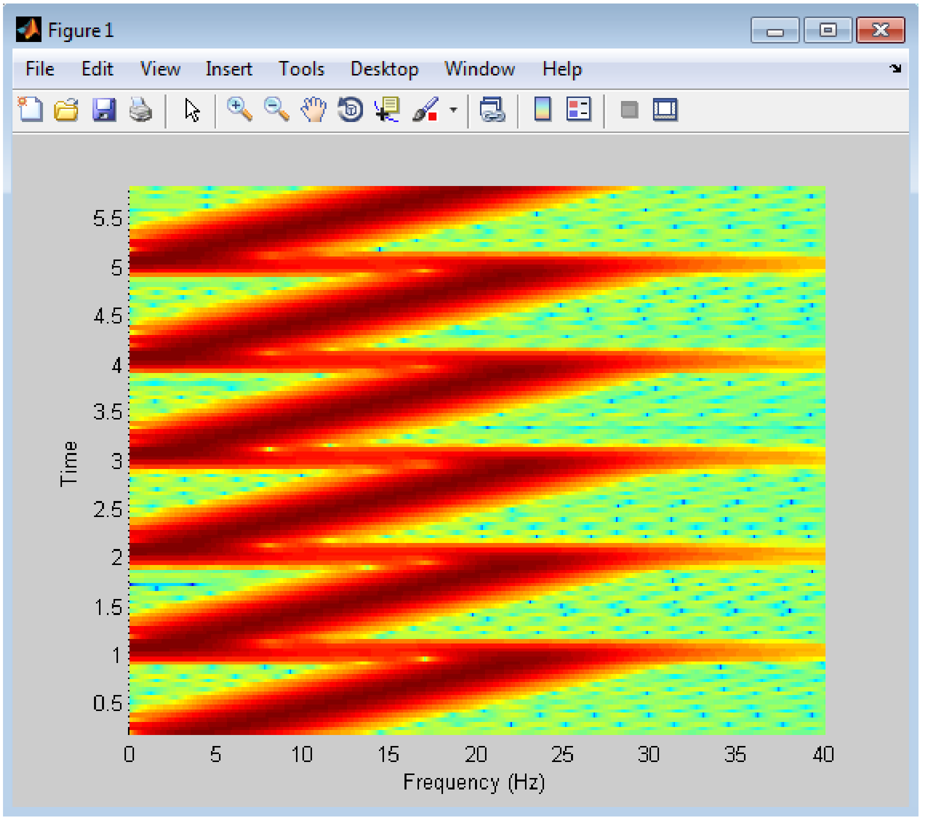


Since **Target time** is set to equal **Sweep time** (1 second), the **Target frequency** (25 Hz) is the final frequency of the unidirectional sweep. Run your model to see the time domain output:



Type the following command to view the chirp output spectrogram:

```
spectrogram(dsp_examples_yout,hamming(128),...
      110,[0:.01:40],400)
```



Chirp Block Parameters for Example 1	
Frequency sweep	`Linear`
Sweep mode	`Unidirectional`
Initial frequency	0
Target frequency	25
Target time	1
Sweep time	1
Initial phase	0
Sample time	1/400
Samples per frame	400

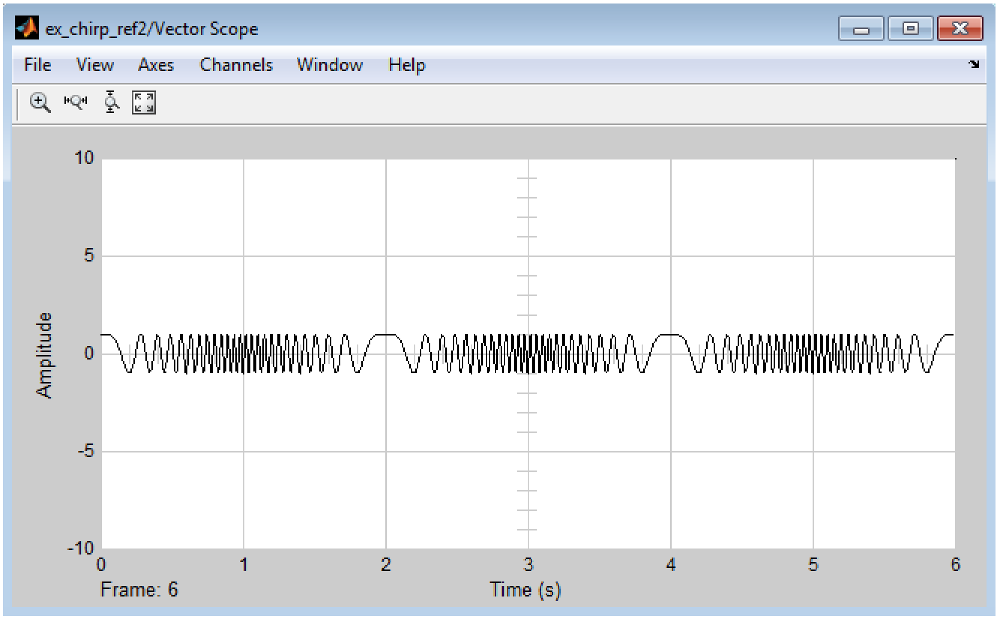
Vector Scope Block Parameters for Example 1	
Input domain	`Time`
Time display span	6
Signal To Workspace Block Parameters for Example 1	
Variable name	`dsp_examples_yout`
Configuration Dialog Parameters for Example 1	
Stop time	5

Example 2: Bidirectional Sweeps

Change the **Sweep mode** parameter in the Example 1 model to `Bidirectional`, and leave all other parameters the same to view the following bidirectional chirp. Note that in the bidirectional sweep, the period of the sweep is twice the **Sweep time** (2 seconds), whereas it was one **Sweep time** (1 second) for the unidirectional sweep in Example 1.

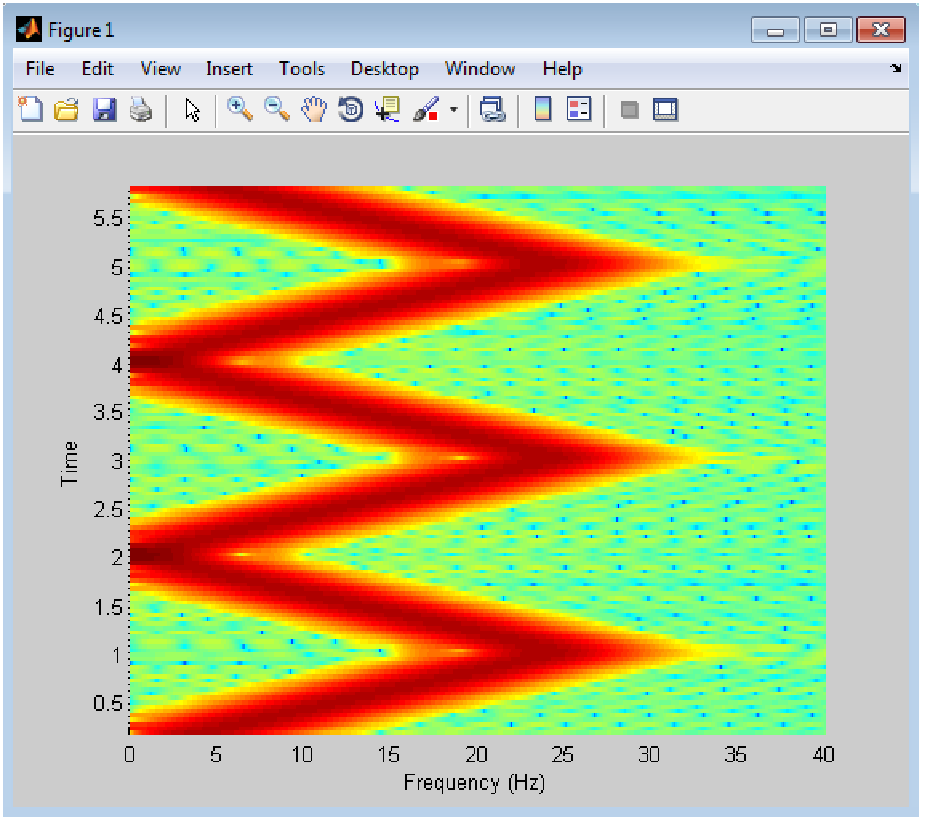
Open the Example 2 model by typing `ex_chirp_ref2` at the MATLAB command line.

Run your model to see the time domain output:



Type the following command to view the chirp output spectrogram:

```
spectrogram(dsp_examples_yout,hamming(128),...
     110,[0:.01:40],400)
```



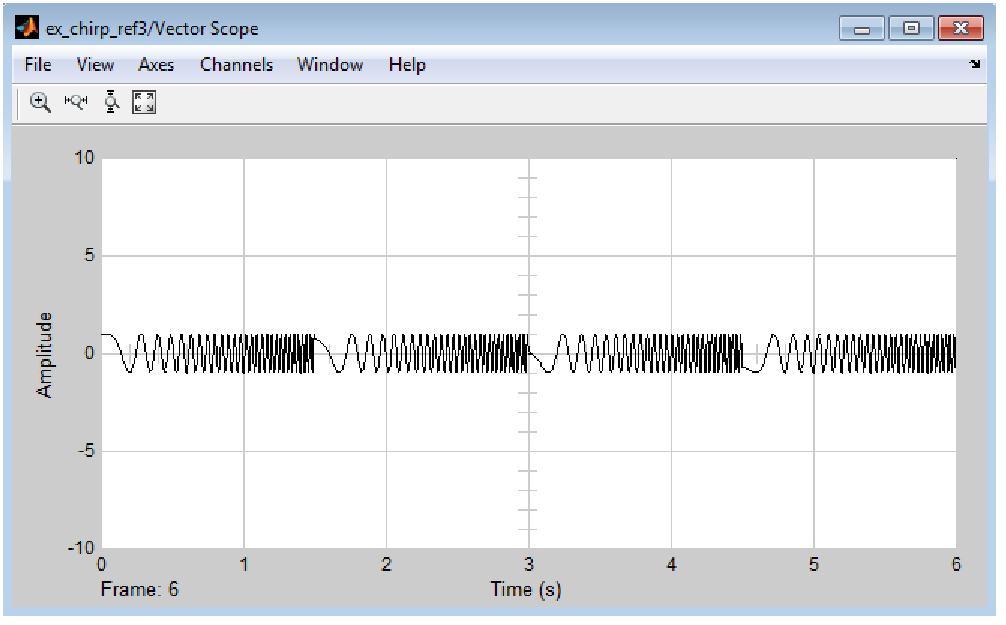
Example 3: When Sweep Time is Greater Than Target Time

Setting **Sweep time** to `1.5` and leaving the rest of the parameters as in the Example 1 model gives the following output. The sweep still reaches the **Target frequency** (25 Hz) at the **Target time** (1 second), but since **Sweep time** is greater than **Target time**, the sweep continues on its linear path until one **Sweep time** (1.5 seconds) is traversed.

Unexpected behavior might arise when you set **Sweep time** greater than **Target time**; see "Example 4: Output Sweep with Negative Frequencies" on page 2-266 for details.

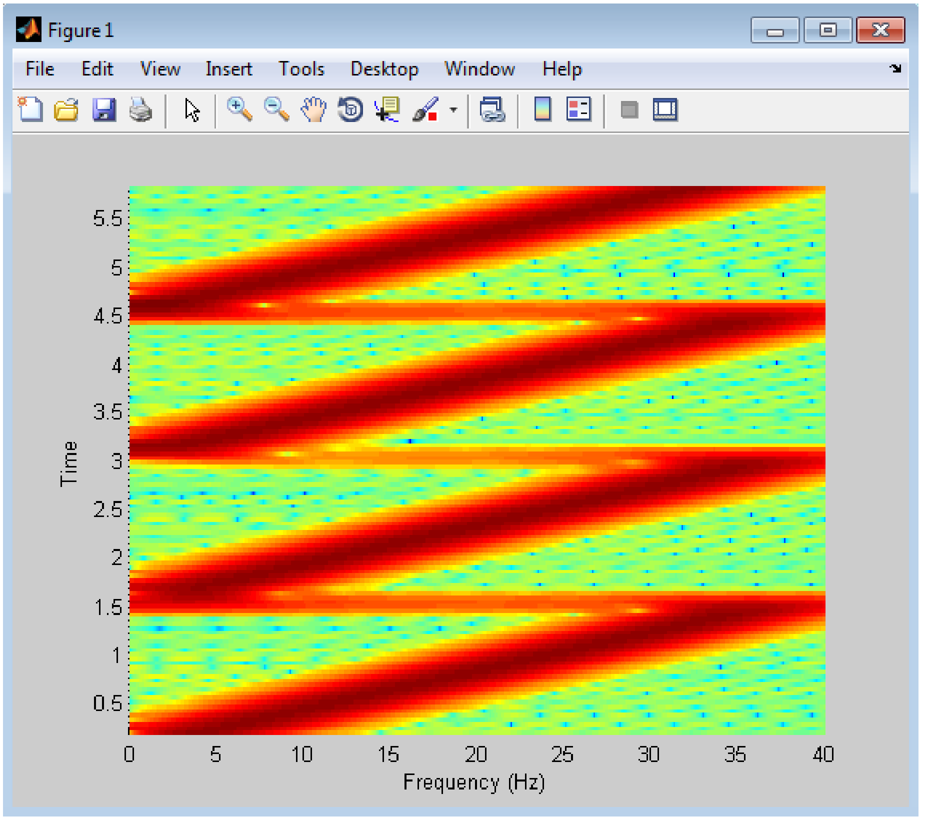
Open the Example 3 model by typing `ex_chirp_ref3` at the MATLAB command line.

Run your model to see the time domain output:



Type the following command to view the chirp output spectrogram:

```
spectrogram(dsp_examples_yout,hamming(128),...
      110,[0:.01:40],400)
```

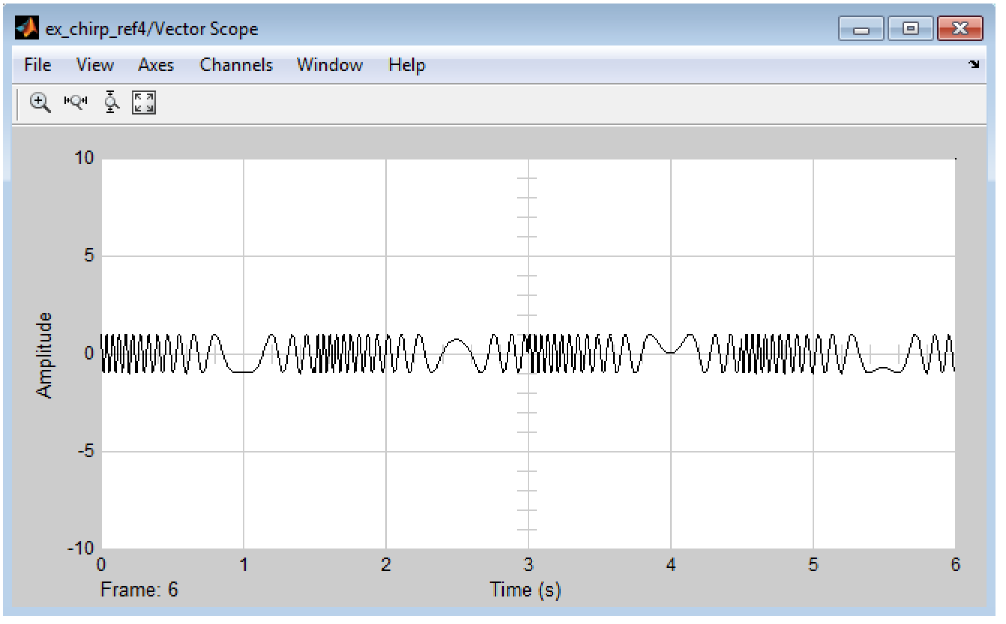


Example 4: Output Sweep with Negative Frequencies

Modify the Example 1 model by changing **Sweep time** to `1.5`, **Initial frequency** to `25`, and **Target frequency** to `0`. *The output chirp of this example might not behave as you expect* because the sweep contains negative frequencies between 1 and 1.5 seconds. The sweep reaches the **Target frequency** of 0 Hz at one second, then continues on its negative slope, taking on negative frequency values until it traverses one **Sweep time** (1.5 seconds).

Open the Example 4 model by typing `ex_chirp_ref4` at the MATLAB command line.

Run your model to see the time domain output:

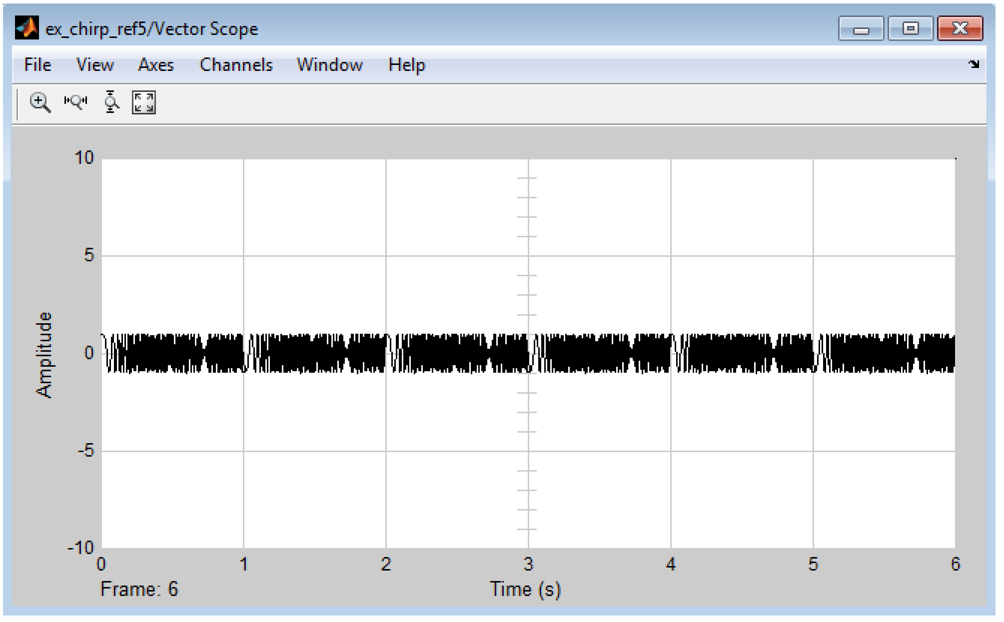


Example 5: Output Sweep with Frequencies Greater Than Half the Sampling Frequency

Modify the Example 1 model by changing the **Target frequency** parameter to `275`. *The output chirp of this model might not behave as you expect* because the sweep contains frequencies greater than half the sampling frequency (`200` Hz), which causes aliasing. If you unexpectedly get a chirp output with a spectrogram resembling the one following, your chirp's sweep might contain frequencies greater than half the sampling frequency.

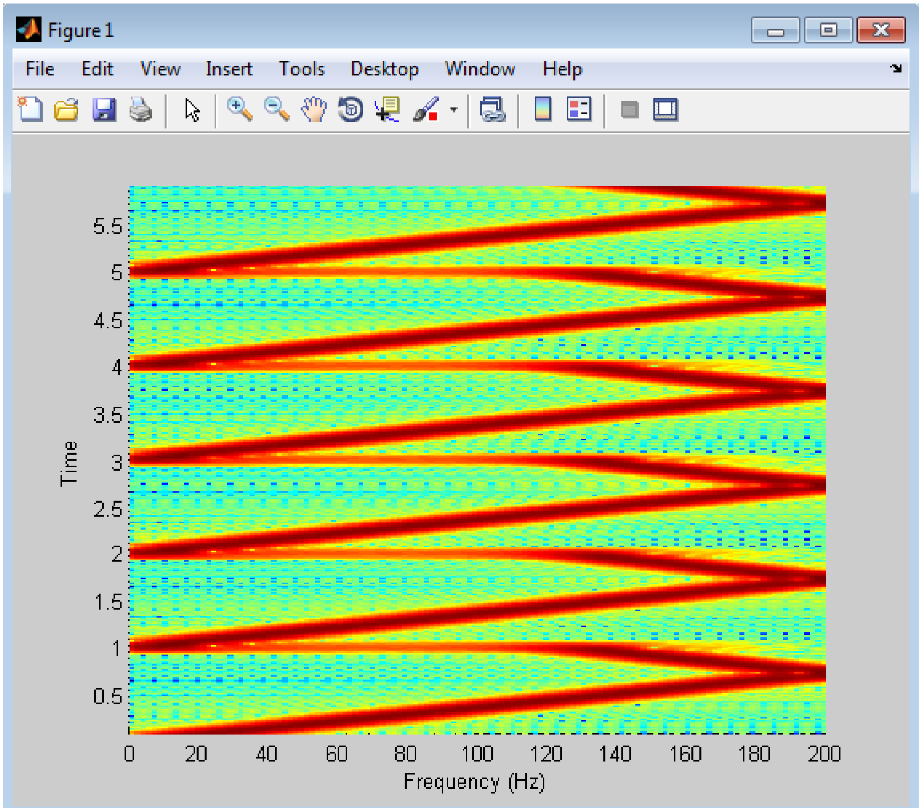
Open the Example 5 model by typing `ex_chirp_ref5` at the MATLAB command line.

Run your model to see the time domain output:



Type the following command to view the chirp output spectrogram:

```
spectrogram(dsp_examples_yout,hamming(64),...
       60,256,400)
```



Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

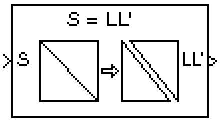
`Signal From Workspace`	DSP System Toolbox
Signal Generator	Simulink
`Sine Wave`	DSP System Toolbox

`chirp`	Signal Processing Toolbox
`spectrogram`	Signal Processing Toolbox

Introduced before R2006a

Cholesky Factorization

Factor square Hermitian positive definite matrix into triangular components



Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

`dspfactors`

Description

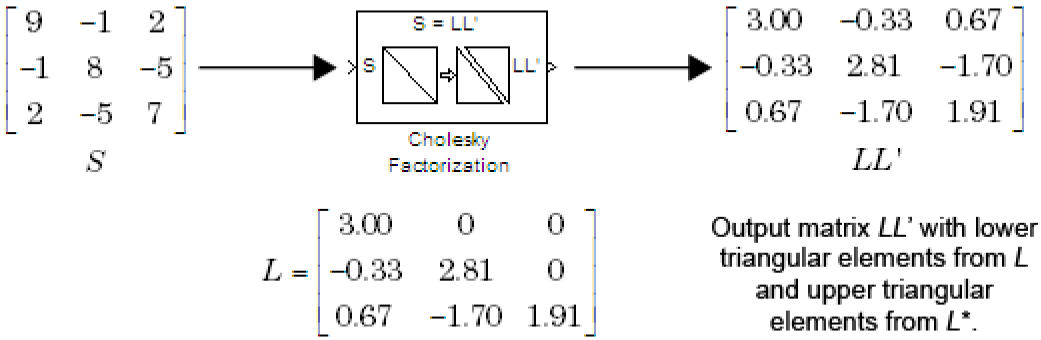
The Cholesky Factorization block uniquely factors the square Hermitian positive definite input matrix S as

$$S = LL^*$$

where L is a lower triangular square matrix with positive diagonal elements and L^* is the Hermitian (complex conjugate) transpose of L. The block outputs a matrix with lower triangle elements from L and upper triangle elements from L^*. The output is not in the same form as the output of the MATLAB `chol` function. In order to convert the output of the Cholesky Factorization block to the MATLAB form, use the following equation:

```
R = triu(LL');
```

Here, `LL'` is the output of the Cholesky Factorization block. Due to roundoff error, these equations do not produce a result that is exactly the same as the MATLAB result.



Block Output Composed of L and L[*]

Input Requirements for Valid Output

The block output is valid only when its input has the following characteristics:

- Hermitian — The block does *not* check whether the input is Hermitian; it uses only the diagonal and upper triangle of the input to compute the output.
- Real-valued diagonal entries — The block disregards any imaginary component of the input's diagonal entries.
- Positive definite — Set the block to notify you when the input is not positive definite as described in “Response to Nonpositive Definite Input” on page 2-272.

Response to Nonpositive Definite Input

To generate a valid output, the block algorithm requires a positive definite input (see “Input Requirements for Valid Output” on page 2-272). Set the **Non-positive definite input** parameter to determine how the block responds to a nonpositive definite input:

- `Ignore` — Proceed with the computation and *do not* issue an alert. The output is *not* a valid factorization. A partial factorization will be present in the upper left corner of the output.
- `Warning` — Display a warning message in the MATLAB Command Window, and continue the simulation. The output is *not* a valid factorization. A partial factorization will be present in the upper left corner of the output.
- `Error` — Display an error dialog and terminate the simulation.

Note The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog box, it is set to `Ignore` in the code generated for this block by Simulink Coder™ code generation software.

Performance Comparisons with Other Blocks

Note that L and L^* share the same diagonal in the output matrix. Cholesky factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable.

Parameters

Non-positive definite input

Response to nonpositive definite matrix inputs: `Ignore`, `Warning`, or `Error`. See “Response to Nonpositive Definite Input” on page 2-272.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

Port	Supported Data Types
S	• Double-precision floating point • Single-precision floating point
LL'	• Double-precision floating point • Single-precision floating point

See Also

`Autocorrelation LPC`	DSP System Toolbox

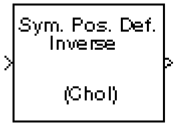
`Cholesky Inverse`	DSP System Toolbox
`Cholesky Solver`	DSP System Toolbox
`LDL Factorization`	DSP System Toolbox
`LU Factorization`	DSP System Toolbox
`QR Factorization`	DSP System Toolbox
`chol`	MATLAB

See “Matrix Factorizations” for related information.

Introduced before R2006a

Cholesky Inverse

Compute inverse of Hermitian positive definite matrix using Cholesky factorization



Library

Math Functions / Matrices and Linear Algebra / Matrix Inverses

`dspinverses`

Description

The Cholesky Inverse block computes the inverse of the Hermitian positive definite input matrix S by performing Cholesky factorization.

$$S^{-1} = (LL^{*})^{-1}$$

L is a lower triangular square matrix with positive diagonal elements and L^{*} is the Hermitian (complex conjugate) transpose of L. Only the diagonal and upper triangle of the input matrix are used, and any imaginary component of the diagonal entries is disregarded. Cholesky factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable.

Response to Nonpositive Definite Input

The algorithm requires that the input be Hermitian positive definite. When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- `Ignore` — Proceed with the computation and *do not* issue an alert. The output is *not* a valid inverse.

- `Warning` — Display a warning message in the MATLAB Command Window, and continue the simulation. The output is *not* a valid inverse.
- `Error` — Display an error dialog box and terminate the simulation.

Note The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog box, it is set to `Ignore` in the code generated for this block by Simulink Coder code generation software.

Parameters

Non-positive definite input

Response to nonpositive definite matrix inputs: `Ignore`, `Warning`, or `Error`. See "Response to Nonpositive Definite Input" on page 2-275.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

`Cholesky Factorization`	DSP System Toolbox
`Cholesky Solver`	DSP System Toolbox
`LDL Inverse`	DSP System Toolbox
`LU Inverse`	DSP System Toolbox
`Pseudoinverse`	DSP System Toolbox

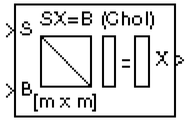
`inv`	MATLAB

See “Matrix Inverses” for related information.

Introduced before R2006a

Cholesky Solver

Solve *S*X=*B* for X when *S* is square Hermitian positive definite matrix



Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

`dspsolvers`

Description

The Cholesky Solver block solves the linear system *S*X=*B* by applying Cholesky factorization to input matrix at the `S` port, which must be square (*M*-by-*M*) and Hermitian positive definite. Only the diagonal and upper triangle of the matrix are used, and any imaginary component of the diagonal entries is disregarded. The input to the `B` port is the right side *M*-by-*N* matrix, *B*. The *M*-by-*N* output matrix X is the unique solution of the equations.

A length-*M* vector input for right side *B* is treated as an *M*-by-1 matrix.

Response to Nonpositive Definite Input

When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- `Ignore` — Proceed with the computation and *do not* issue an alert. The output is *not* a valid solution.
- `Warning` — Proceed with the computation and display a warning message in the MATLAB Command Window. The output is *not* a valid solution.
- `Error` — Display an error dialog box and terminate the simulation.

Note The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog box, it is set to `Ignore` in the code generated for this block by Simulink Coder code generation software.

Algorithm

Cholesky factorization uniquely factors the Hermitian positive definite input matrix S as

$$S = LL^*$$

where L is a lower triangular square matrix with positive diagonal elements.

The equation $SX=B$ then becomes

$$LL^*X = B$$

which is solved for X by making the substitution $Y = L^*X$, and solving the following two triangular systems by forward and backward substitution, respectively.

$$LY = B$$

$$L^*X = Y$$

Parameters

Non-positive definite input

Response to nonpositive definite matrix inputs: `Ignore`, `Warning`, or `Error`. See "Response to Nonpositive Definite Input" on page 2-278.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

`Autocorrelation LPC`	DSP System Toolbox
`Cholesky Factorization`	DSP System Toolbox
`Cholesky Inverse`	DSP System Toolbox
`LDL Solver`	DSP System Toolbox
`LU Solver`	DSP System Toolbox
`QR Solver`	DSP System Toolbox
`chol`	MATLAB

See “Linear System Solvers” for related information.

Introduced before R2006a

CIC Compensator

Design CIC compensator

Note: The CIC Compensator block has been removed from the DSP System Toolbox block library. Existing instances of the CIC Compensator block will continue to operate. For new models, use the `CIC Compensation Decimator` and `CIC Compensation Interpolator` blocks. These blocks replace the functionality of the CIC Compensator block, when **FilterType** is set to `Decimator` and `Interpolator`, respectively.

Library

Filtering / Filter Designs

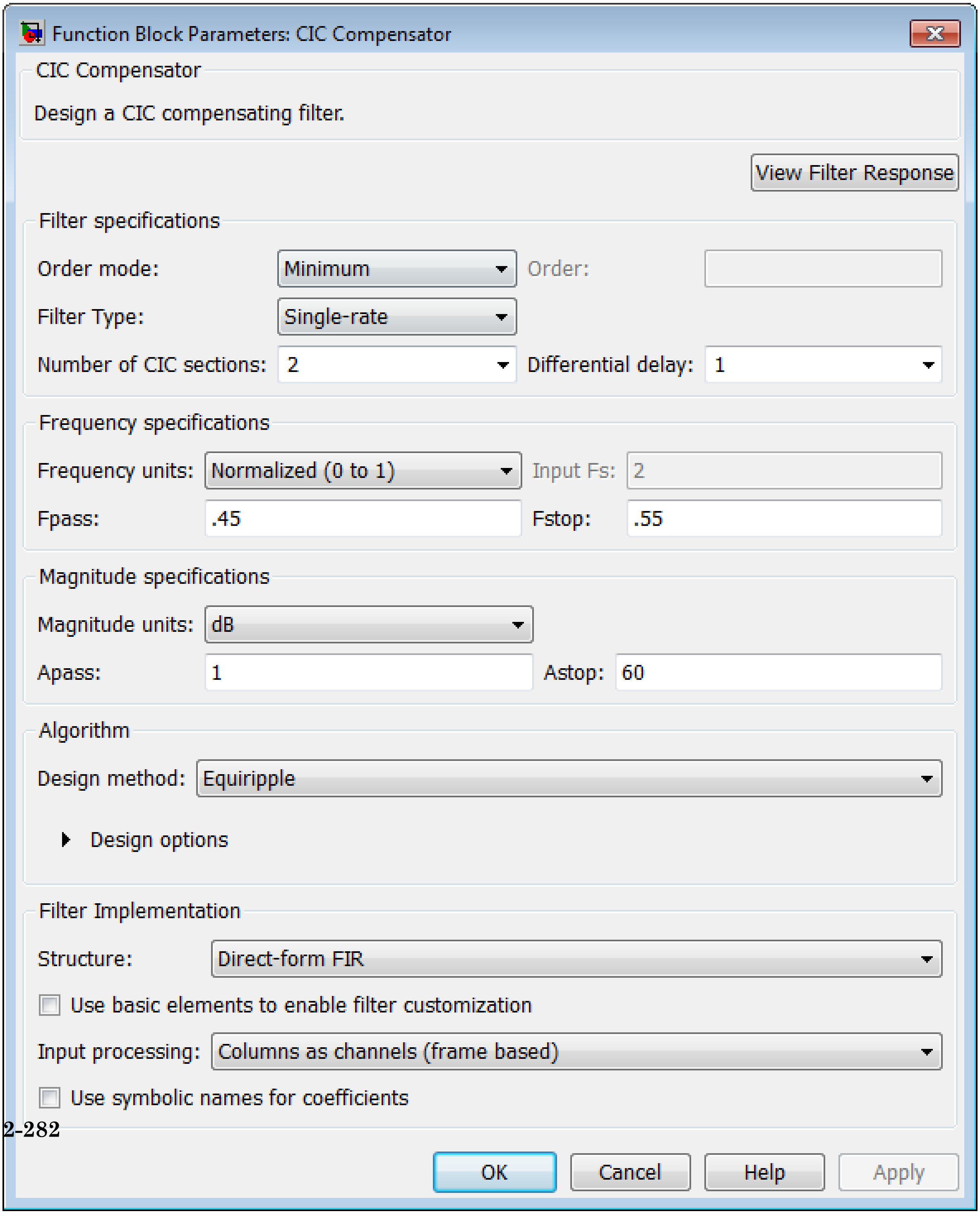
`dspfdesign`

Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See "CIC Compensator Design — Main Pane" on page 5-733 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.


Function Block Parameters: CIC Compensator
CIC Compensator
Design a CIC compensating filter.
View Filter Response
Filter specifications
Order mode:
Minimum
Order:
Filter Type:
Single-rate
Number of CIC sections:
2
Differential delay:
1
Frequency specifications
Frequency units:
Normalized (0 to 1)
Input Fs:
2
Fpass:
.45
Fstop:
.55
Magnitude specifications
Magnitude units:
dB
Apass:
1
Astop:
60
Algorithm
Design method:
Equiripple
Design options
Filter Implementation
Structure:
Direct-form FIR
Use basic elements to enable filter customization
Input processing:
Columns as channels (frame based)
Use symbolic names for coefficients
OK
Cancel
Help
Apply

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

In its normal mode of operation, the CIC Compensator block allows the adder's numbers to wrap around. The Fixed-Point infrastructure then causes warnings to appear on the command line.

Filter Specifications

In this group, you specify your filter format, such as the filter order mode and the filter type.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Number of CIC sections

Specify the number of sections in the CIC filter for which you are designing this compensator. Select the number of sections from the drop-down list or enter the number.

Differential Delay

Specify the differential delay of your target CIC filter. The default value is `1`. Most CIC filters use 1 or 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

Frequency Specifications

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default method is `Equiripple`.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Specify the phase constraint of the filter as `Linear`, `Maximum`, or `Minimum`.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent *n* in the relation $(1/f)^n$ to define the stopband decay. The block applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, the filter uses direct-form structure.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note: The `Inherited (this choice will be removed — see release notes)` option will be removed in a future release. See “Frame-Based Processing” in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2006b

CIC Compensation Interpolator

Compensate for CIC filter using FIR interpolator

Library

Filtering/Filter Designs

`dspfdesign`

Description

The CIC Compensation Interpolator block uses an FIR polyphase interpolator as the compensation filter. CIC compensation interpolators are multirate FIR filters that can be cascaded with CIC interpolators to mitigate the drawbacks of the CIC filters.

CIC interpolation filters are used in areas that require high interpolation. These filters are popular in ASICs and FPGAs, since they do not have any multipliers. CIC filters have two drawbacks:

- CIC filters have a magnitude response that causes a droop in the passband region. This magnitude response is:

$$abs\left(\frac{\sin\left(M\frac{\omega}{2}\right)}{\sin\left(\frac{\omega}{2}\right)}\right)^{n}$$

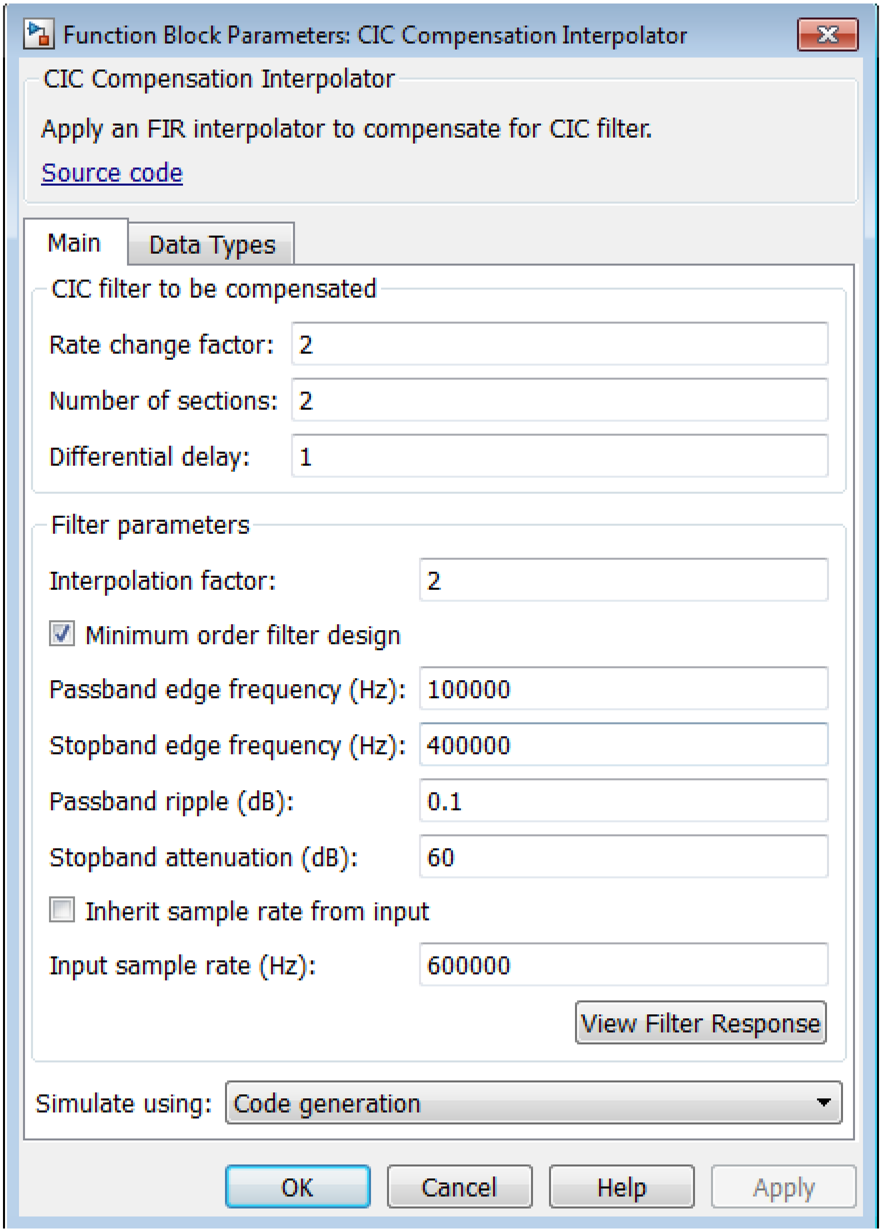
  - M — Differential delay
  - n — Number of stages
  - ω — Normalized angular frequency

- CIC filters have a wide transition region.

The compensation interpolator filters have an inverse sinc passband response to correct for the CIC droop, and they have a narrow transition width.

This block brings the capabilities of the dsp.CICCompensationInterpolator System object to the Simulink environment.

Dialog Box


Function Block Parameters: CIC Compensation Interpolator
CIC Compensation Interpolator
Apply an FIR interpolator to compensate for CIC filter.
Source code
Main
Data Types
CIC filter to be compensated
Rate change factor:
2
Number of sections:
2
Differential delay:
1
Filter parameters
Interpolation factor:
2
Minimum order filter design
Passband edge frequency (Hz):
100000
Stopband edge frequency (Hz):
400000
Passband ripple (dB):
0.1
Stopband attenuation (dB):
60
Inherit sample rate from input
Input sample rate (Hz):
600000
View Filter Response
Simulate using:
Code generation
OK
Cancel
Help
Apply

Rate change factor

Rate change factor for the CIC filter to be compensated, specified as a positive scalar integer. The default is 2.

Number of sections

Number of integrator and comb sections of the CIC filter to be compensated, specified as a positive scalar integer. The default is 2.

Differential delay

Delay value used in each of the comb sections of the CIC filter to be compensated, specified as a positive scalar integer. The default is 1.

Filter parameters

Interpolation factor

Interpolation factor of the compensator, specified as a positive scalar integer. The default value is 2.

Minimum order filter design

When you select this check box, the block designs filters with the minimum order that meets the specifications for passband frequency, stopband frequency, passband ripple, and stopband attenuation. When you clear this check box, the block designs filters with the order that you specify in **Filter order**.

By default, this check box is selected.

Filter order

Order of the compensation filter, specified as a positive scalar integer. The default is 12.

Passband edge frequency (Hz)

Passband edge frequency of the compensation filter, specified as a real positive scalar in Hz. **Passband edge frequency (Hz)** must be less than $F_s/2$, where F_s is the output sample rate. The default is 100000.

Stopband edge frequency (Hz)

Stopband edge frequency of the compensation filter, specified as a real positive scalar in hertz. **Stopband edge frequency (Hz)** must be less than $F_s/2$, where F_s is the output sample rate. The default is 400000.

Passband ripple (dB)

Passband ripple of compensation filter, specified as a real positive scalar in dB. The default is 0.1.

Stopband attenuation (dB)

Stopband attenuation of compensation filter, specified as a real positive scalar in dB. The default is 60.

Inherit sample rate from input

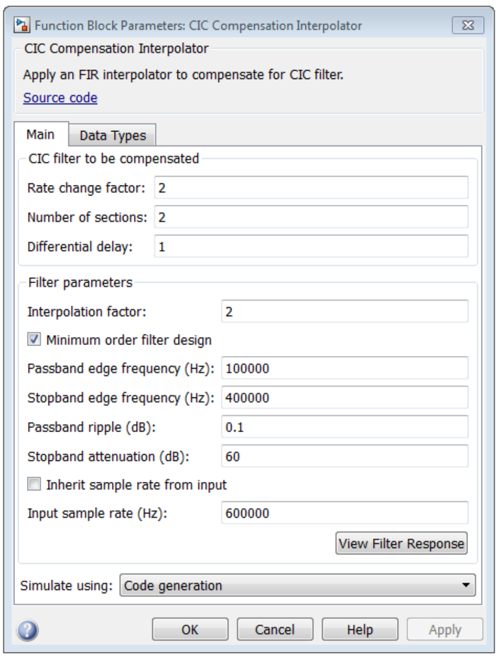
When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you must specify the sample rate in **Input sample rate (Hz)**.

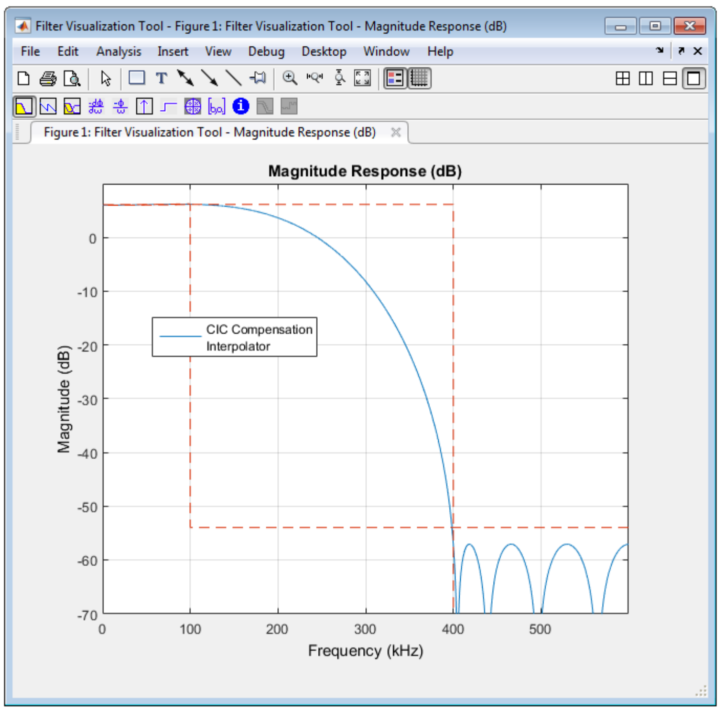
Input sample rate (Hz)

Input sample rate, specified as a scalar in Hz. The default is 600000.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the CIC Compensation Interpolator. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.





To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

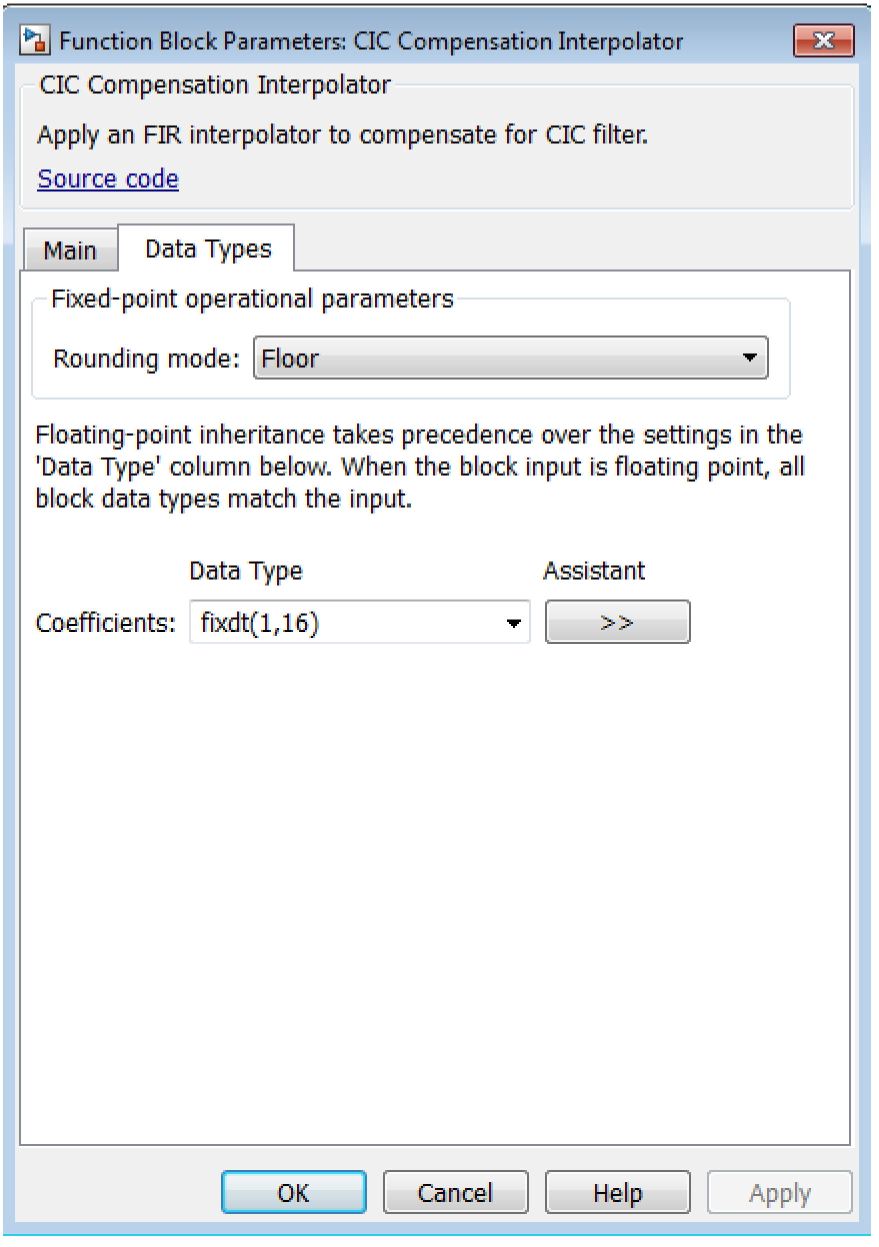
- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Data Types Tab



Rounding mode

Rounding method for the output fixed-point operations. The rounding methods are `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, and `Zero`. The default is `Floor`.

Coefficients data type

Fixed-point data type of the coefficients, specified as one of the following:

- `fixdt(1,16)` (default) — Signed fixed-point data type of word length `16`, with binary point scaling. The block determines the fraction length automatically from the coefficient values in such a way that the coefficients occupy maximum representable range without overflowing.
- `fixdt(1,16,0)` — Signed fixed-point data type of word length `16`, fraction length `0`. You can change the fraction length to any other integer value.
- <data type expression> — Specify the coefficients data type by using the expression that evaluates to a data type object, for example, numeric type (`fixdt([ ],16, 15)`, to specify the coefficients data type. Specify the sign mode of this data type as `[ ]` or true.
- `Refresh Data Type` — Refreshes to the default data type.

Click the **Show data type assistant** button >> to display the data type assistant, which helps you set the stage input parameter.

See "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink) for more information.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed or unsigned) • 8-, 16-, 32-, and 64-bit signed integers
Output	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
	• Fixed point (signed only) • 8-, 16-, 32-, and 64-bit signed integers

See Also

dsp.CICCompensationInterpolator	DSP System Toolbox
`CIC Compensation Decimator`	DSP System Toolbox

Introduced in R2015b

CIC Compensation Decimator

Compensate for CIC filter using FIR decimator

Library

Filtering/Filter Designs

`dspfdesign`

Description

The CIC Compensation Decimator block uses an FIR polyphase decimator as the compensation filter. CIC compensation decimators are multirate FIR filters that can be cascaded with CIC decimators to mitigate the drawbacks of the CIC filters.

CIC decimation filters are used in areas that require high decimation. These filters are popular in ASICs and FPGAs, since they do not have any multipliers. CIC filters have two drawbacks:

- CIC filters have a magnitude response that causes a droop in the passband region. This magnitude response is:

$$abs\left(\frac{\sin\left(M\frac{\omega}{2}\right)}{\sin\left(\frac{\omega}{2}\right)}\right)^n$$

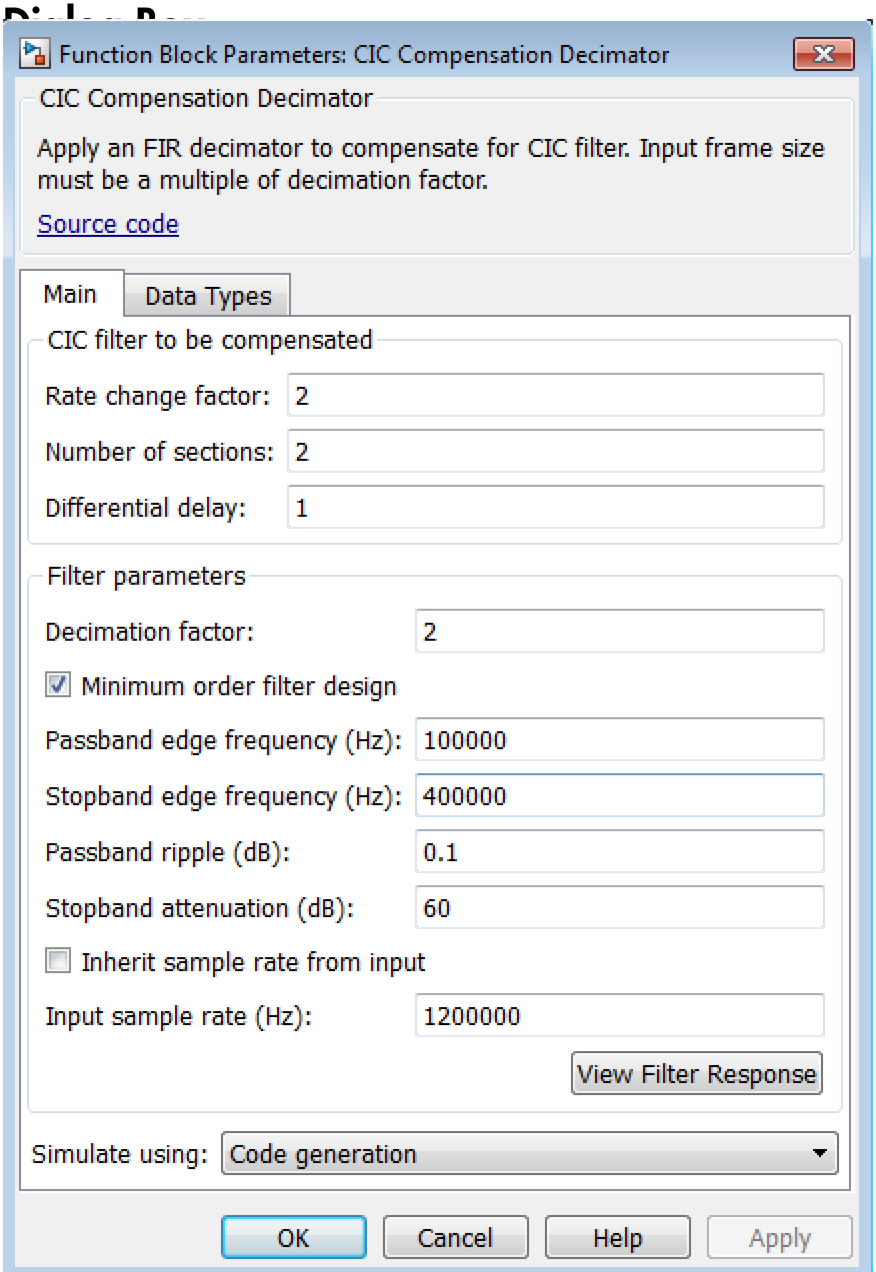
  - M — Differential delay
  - n — Number of stages
  - ω — Normalized angular frequency

- CIC filters have a wide transition region.

The compensation decimator filters have an inverse passband response to correct for the CIC droop, and they have a narrow transition width.

This block brings the capabilities of the dsp.CICCompensationDecimator System object to the Simulink environment.

Dialog Box


Function Block Parameters: CIC Compensation Decimator
CIC Compensation Decimator
Apply an FIR decimator to compensate for CIC filter. Input frame size must be a multiple of decimation factor.
Source code
Main
Data Types
CIC filter to be compensated
Rate change factor:
2
Number of sections:
2
Differential delay:
1
Filter parameters
Decimation factor:
2
Minimum order filter design
Passband edge frequency (Hz):
100000
Stopband edge frequency (Hz):
400000
Passband ripple (dB):
0.1
Stopband attenuation (dB):
60
Inherit sample rate from input
Input sample rate (Hz):
1200000
View Filter Response
Simulate using:
Code generation
OK
Cancel
Help
Apply

Rate change factor

Rate change factor for the CIC filter to be compensated, specified as a positive scalar integer. The default is `2`.

Number of sections

Number of decimator and comb sections of the CIC filter to be compensated, specified as a positive scalar integer. The default is `2`.

Differential delay

Delay value used in each of the comb sections of the CIC filter to be compensated, specified as a positive scalar integer. The default is `1`.

Decimation factor

Decimation factor of the compensator, specified as a positive scalar integer. The default is `2`.

Minimum order filter design

When you select this check box, the block designs filters with the minimum order that meets the specifications passband frequency, stopband frequency, passband ripple, and stopband attenuation. When you clear this check box, the block designs filters with the order that you specify in **Filter order**.

By default, this check box is selected.

Filter order

Order of compensation filter, specified as a positive scalar integer. The default is `12`.

Passband edge frequency (Hz)

Passband edge frequency of the compensation filter, specified as a real positive scalar in Hz. **Passband edge frequency (Hz)** must be less than $F_s/2$, where F_s is the input sample rate. The default value is `100000`.

Stopband edge frequency (Hz)

Stopband edge frequency of the compensation filter, specified as a real positive scalar in Hz. **Stopband edge frequency (Hz)** must be less than $F_s/2$, where F_s is the input sample rate. This parameter applies when you select the **Minimum order filter design** check box. The default is `400000`.

Passband ripple (dB)

Passband ripple of the compensation filter, specified as a real positive scalar in dB. The default is `0.1`.

Stopband attenuation (dB)

Stopband attenuation of the compensation filter, specified as a real positive scalar in dB. The default is 60.

Inherit sample rate from input

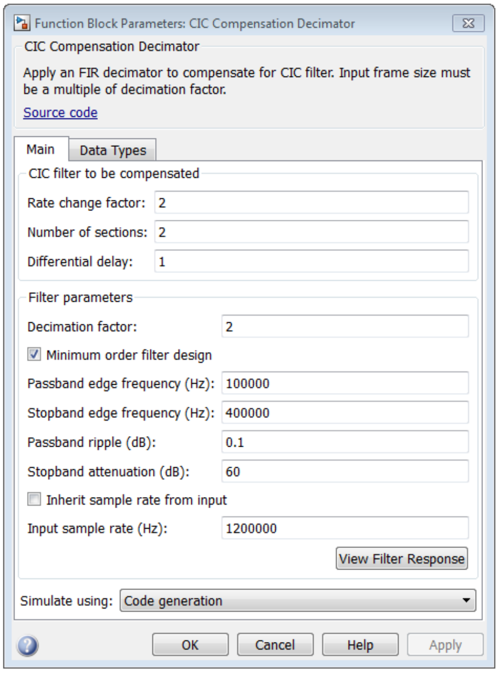
When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you must specify the sample rate in **Input sample rate (Hz)**.

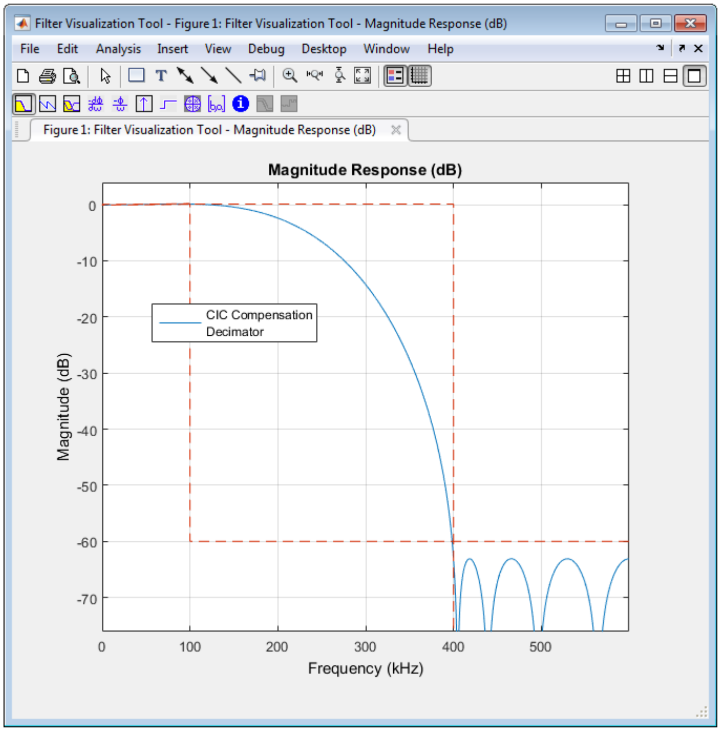
Input sample rate (Hz)

Input sample rate, specified as a scalar in Hz. The default is 1200000.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the CIC Compensation Decimator. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.





To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

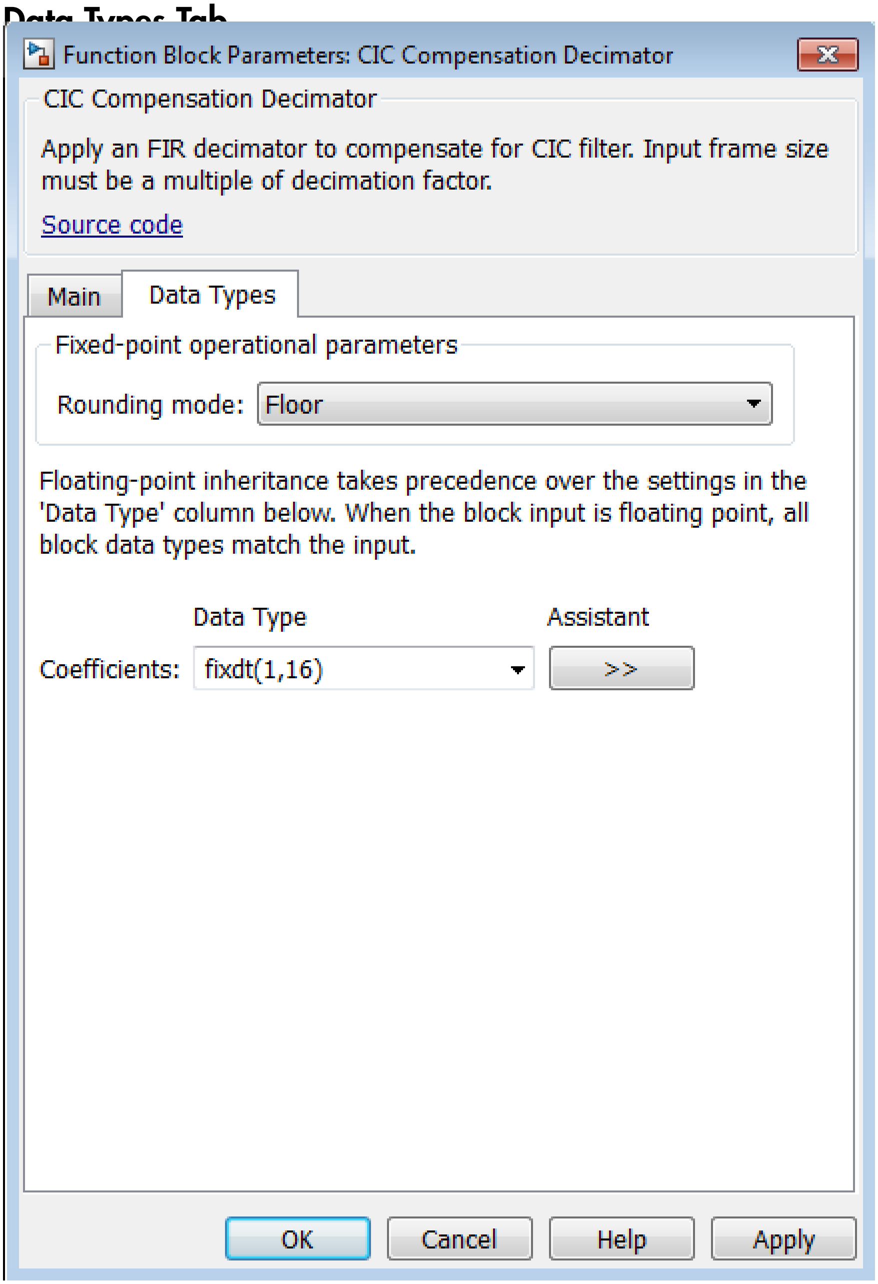
- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Data Types Tab



Rounding mode

Rounding method for the output fixed-point operations. The rounding methods are `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, and `Zero`. The default is `Floor`.

Coefficients data type

Fixed-point data type of the coefficients, specified as one of the following:

- `fixdt(1,16)` (default) — Signed fixed-point data type of word length `16`, with binary point scaling. The block determines the fraction length automatically from the coefficient values in such a way that the coefficients occupy maximum representable range without overflowing.
- `fixdt(1,16,0)` — Signed fixed-point data type of word length `16`, fraction length `0`. You can change the fraction length to any other integer value.
- `<data type expression>` — Specify the coefficients data type by using an expression that evaluates to a data type object, for example, numeric type (fixdt(`[ ]`,`16`, `15`)), to specify the coefficients data type. Specify the sign mode of this data type as `[ ]` or true.
- `Refresh Data Type` — Refresh to the default data type.

Click the **Show data type assistant** button >> to display the data type assistant, which helps you set the stage input parameter.

See "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink) for more information.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed or unsigned) • 8-, 16-, 32-, and 64-bit signed integers
Output	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
	• Fixed point (signed only) • 8-, 16-, 32-, and 64-bit signed integers

See Also

See Also

System Objects
dsp.CICCompensationDecimator

Blocks
`CIC Compensation Interpolator`

Introduced in R2015b

CIC Decimation

Decimate signal using Cascaded Integrator-Comb filter

Library

Filtering / Multirate Filters

`dspmlti4`

Description

The CIC Decimation block performs a sample rate decrease (decimation) on an input signal by an integer factor. Cascaded Integrator-Comb (CIC) filters are a class of linear phase FIR filters comprised of a comb part and an integrator part.

The transfer function of a CIC decimation filter is

$$H(z) = H_I{}^N(z)H_c{}^N(z) = \frac{\left(1 - z^{-RM}\right)^N}{\left(1 - z^{-1}\right)^N} = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N$$

where

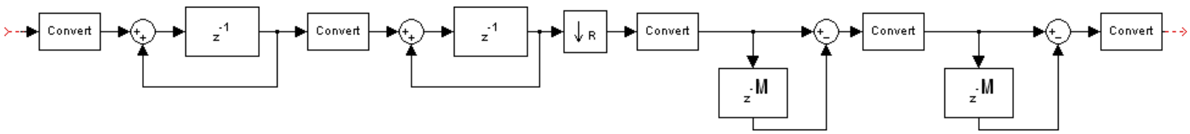
- H_I is the transfer function of the integrator part of the filter.
- H_C is the transfer function of the comb part of the filter.
- N is the number of sections. The number of sections in a CIC filter is defined as the number of sections in either the comb part *or* the integrator part of the filter. This value does not represent the total number of sections throughout the entire filter.
- R is the decimation factor.
- M is the differential delay.

The CIC Decimation block requires a Fixed-Point Designer™ license.

The block supports real and complex fixed-point inputs. In its normal mode of operation, the CIC Decimation block allows the adder's numeric values to overflow and wrap around [1] [3]. The Fixed-Point infrastructure then causes overflow warnings to appear on the command line. This overflow is of no consequence.

CIC Filter Structure

The CIC Decimation block has the following CIC filter structure. The structure consists of N sections of cascaded integrators, followed by a rate change by a factor R, followed by N sections of cascaded comb filters [1] .



The unit delay in the integrator portion of the CIC Filter can be located in either the feed-forward or the feedback path. These two configurations yield identical filter frequency response. However, the numerical outputs from these two configurations are different due to the latency. This block puts the unit delay in the feed forward path of the integrator since it is a preferred configuration for HDL implementation.

Examples

See GSM Digital Down Converter for an example using the CIC Decimation block.

Dialog Box

Coefficient Source

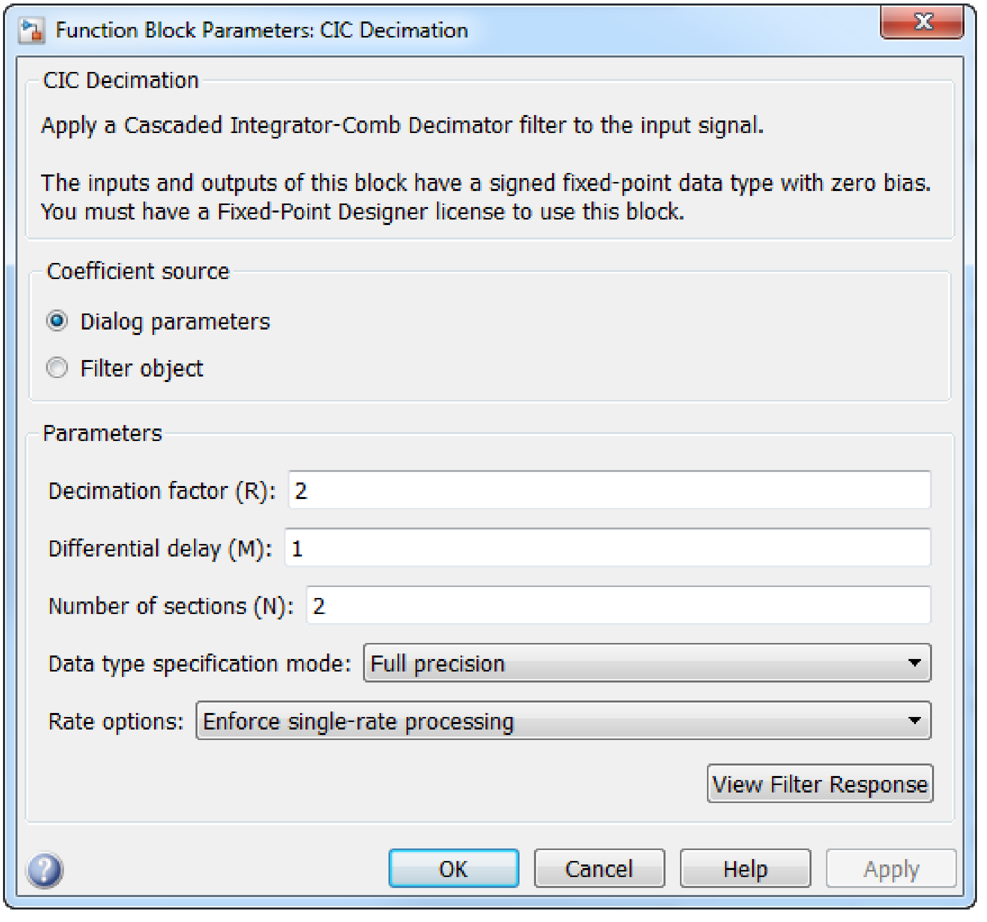
The CIC Decimation block can operate in two different modes. Select the mode in the **Coefficient source** group box.

- **Dialog parameters** — Enter information about the filter, such as **Decimation factor (R)**, **Differential delay (M)** and **Number of sections (N)** in the block mask.
- **Filter object** — Specify the filter using a dsp.CICDecimator System object.

Different items appear on the CIC Decimation block dialog depending on whether you select **Dialog parameters** or **Filter object** in the **Coefficient source** group box. See the following sections for details:

Specify Filter Characteristics in Dialog

The **Main** pane of the CIC Decimation block dialog appears as follows when you select **Dialog parameters** in the **Coefficient source** group box.



Decimation factor (R)

Specify the decimation factor of the filter.

Differential delay (M)

Specify the differential delay of the comb part of the filter, `M`, as shown in the diagram in "CIC Filter Structure" on page 2-309.

Number of sections (N)

Specify the number of filter sections. The number you specify determines the number of sections in either the comb part of the filter or the integrator part of the filter. This value does not represent the total number of sections in the comb and integrator parts combined.

Data type specification mode

Choose how you specify the fixed-point word length and fraction length of the filter sections and/or output.

- `Full precision` — In this mode, the word and fraction lengths of the filter sections and outputs are automatically selected for you. All word lengths are set to

$$\text{word length} = \text{ceil}(N * \log_2(M * R)) + I$$

  where

  - I = input word length
  - M = differential delay
  - N = number of sections
  - R = decimation factor

  All fraction lengths are set to the input fraction length.

- `Minimum section word lengths` — In this mode, you specify the word length of the filter output in the **Output word length** parameter. The block automatically selects the word lengths of the filter sections and all fraction lengths such that each of the section word lengths is as small as possible. The precision of each filter section is less than in `Full precision` mode, but the range of each section is preserved.
- `Specify word lengths` — In this mode you specify the word lengths of the filter sections and output in the **Section word lengths** and **Output word length** parameters. The block automatically selects fraction lengths for the filter sections and output such that the range of each section is preserved when the least significant bits are discarded.

- `Binary point scaling` — In this mode you fully specify the word and fraction lengths of the filter sections and output in the **Section word lengths**, **Section fraction lengths**, **Output word length**, and **Output fraction length** parameters.

Section word lengths

Specify the word length, in bits, of the filter sections.

This parameter is only visible if `Specify word lengths` or `Binary point scaling` is selected for the **Data type specification mode** parameter.

Section fraction lengths

Specify the fraction length of the filter sections.

This parameter is only visible if `Binary point scaling` is selected for the **Data type specification mode** parameter.

Output word length

Specify the word length, in bits, of the filter output.

This parameter is only visible if you select `Minimum section word lengths`, `Specify word lengths`, or `Binary point scaling` for the **Data type specification mode** parameter.

Output fraction length

Specify the fraction length of the filter output.

This parameter is only visible if `Binary point scaling` is selected for the **Data type specification mode** parameter.

Rate options

Specify the rate processing rule for the block. You can select one of the following options:

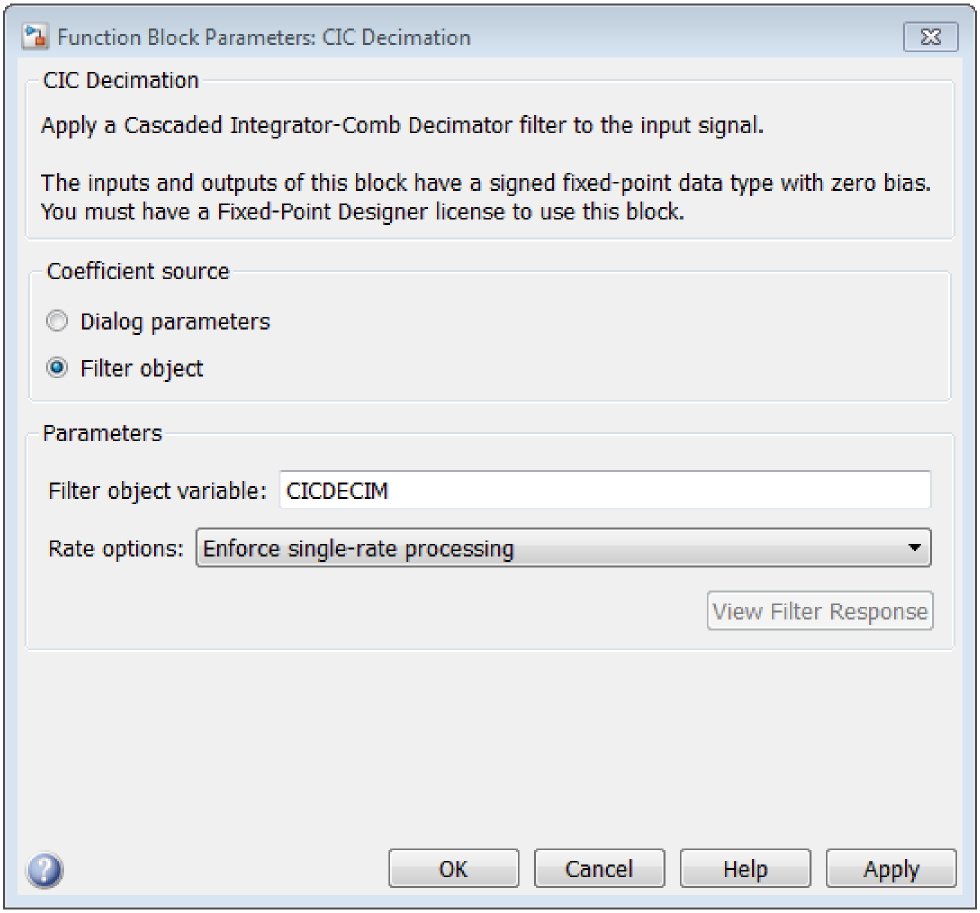
- `Enforce single-rate processing` — When you select this option, the block performs frame-based processing and produces an output that has the same sample rate as the input. To decimate the signal while maintaining the input sample rate, the block decreases the output frame size. In this mode, the input column size must be a multiple of the decimation factor, `R`.
- `Allow multirate processing` — When you select this option, the block performs sample-based processing. In this mode, the block produces an output with a sample rate that is `R` times slower than the input sample rate.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the filter defined in the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

Specify Multirate Filter Object

The **Main** pane of the CIC Decimation block dialog appears as follows when you select **Filter object** in the **Coefficient source** group box.



Multirate filter variable

Specify the name of the multirate filter object that you want the block to implement. You must specify the filter as a dsp.CICDecimator System object.

You can define the System object in the block mask or in a MATLAB workspace variable.

For information on creating System objects, see “Define Basic System Objects” (MATLAB).

Rate options

Specify the rate processing rule for the block. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block performs frame-based processing and produces an output that has the same sample rate as the input. To decimate the signal while maintaining the input sample rate, the block decreases the output frame size. In this mode, the input column size must be a multiple of the decimation factor, `R`.
- `Allow multirate processing` — When you select this option, the block performs sample-based processing and produces an output with a sample rate that is `R` times slower than the input sample rate.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the System object specified in the **Multirate filter variable** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note: If you specify a filter in the **Multirate filter variable** parameter, you must apply the filter by clicking the **Apply** button before using the **View filter response** button.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see CIC Decimation.

References

[1] Hogenauer, E.B., "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-29(2): pp. 155–162, 1981.

[2] Meyer-Baese, U., *Digital Signal Processing with Field Programmable Gate Arrays*, Springer Verlag, 2001.

[3] Harris, Fredric J., *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.

Supported Data Types

- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

See Also

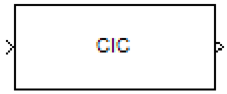
`CIC Interpolation`	DSP System Toolbox
`FIR Decimation`	DSP System Toolbox
`FIR Interpolation`	DSP System Toolbox
`Digital Down-Converter`	DSP System Toolbox
`Digital Up-Converter`	DSP System Toolbox
dsp.CICDecimator	DSP System Toolbox
dsp.CICInterpolator	DSP System Toolbox

dsp.FIRDecimator	DSP System Toolbox
dsp.FIRInterpolator	DSP System Toolbox
dsp.CICCompensationDecimator	DSP System Toolbox
dsp.CICCompensationInterpolator	DSP System Toolbox
dsp.FIRHalfbandDecimator	DSP System Toolbox
dsp.FIRHalfbandInterpolator	DSP System Toolbox
`filter`	DSP System Toolbox

Introduced before R2006a

CIC Filter

Design Cascaded Integrator-Comb (CIC) Filter



Library

Filtering / Filter Designs

`dspfdesign`

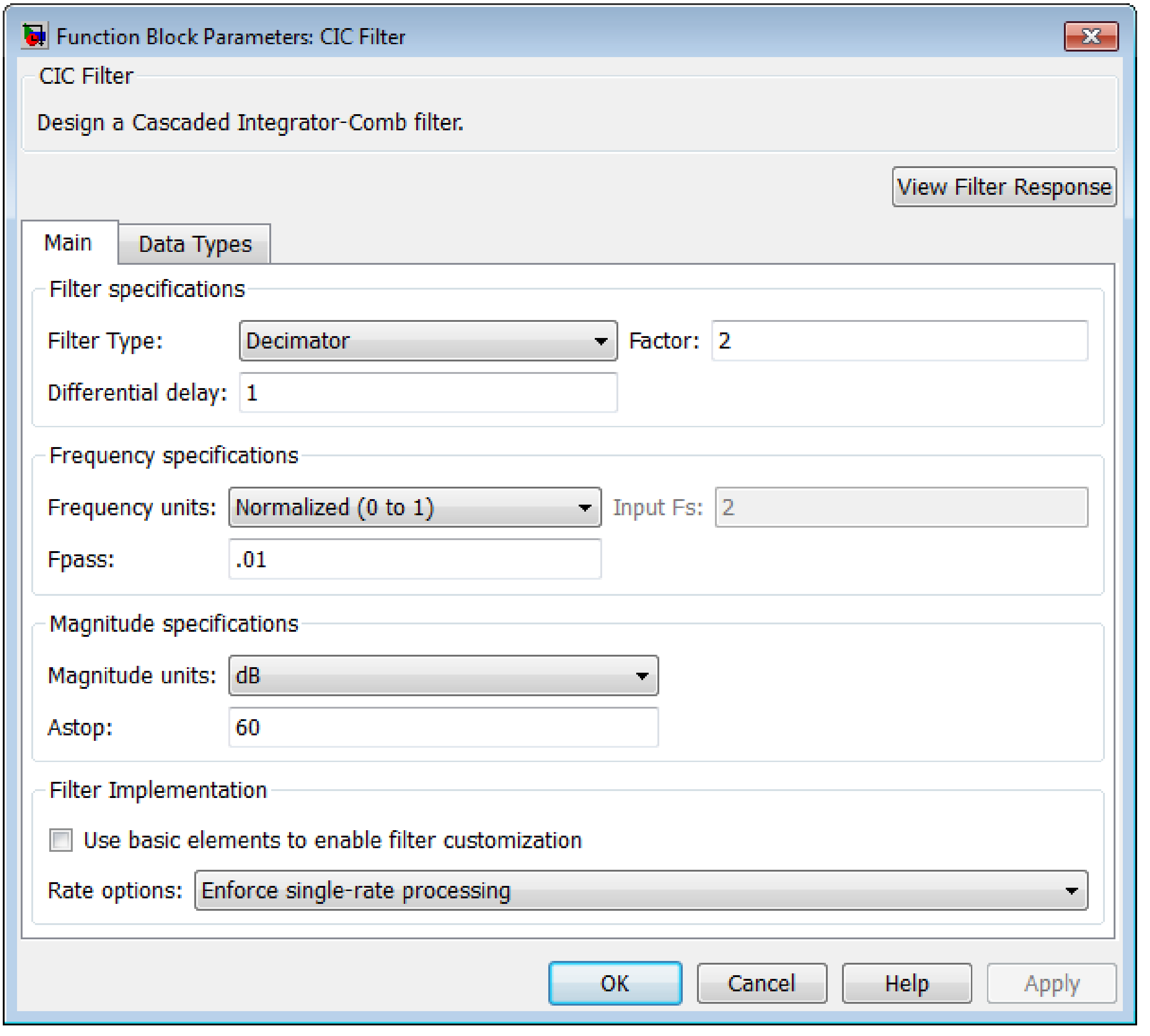
Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

Main Pane

See "CIC Filter Design — Main Pane" on page 5-730 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.



View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

In its normal mode of operation, the CIC Filter block allows the adder's numbers to wrap around. The Fixed-Point infrastructure then causes warnings to appear on the command line.

Filter Specifications

In this group, you specify your CIC filter format, such as the filter type and the differential delay.

Filter type

Select whether your filter will be a `decimator` or an `interpolator`. Your choice determines the type of filter and the design methods and structures that are available to implement your filter. Selecting `decimator` or `interpolator` activates the **Factor** option. When you design an interpolator, you enable the **Output Fs** parameter.

When you design either a decimator or interpolator, the resulting filter is a CIC filter that decimates or interpolates your input signal.

Differential delay

Specify the differential delay of your CIC filter as an integer value greater than or equal to 1. The default value is 1. The differential delay changes the shape, number, and location of nulls in the filter response. Increasing the differential delay increases the sharpness of the nulls and the response between the nulls. In practice, differential delay values of 1 or 2 are the most common.

Factor

Specify the decimation or interpolation factor for your filter as an integer value greater than or equal to 1. The default value is 2.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Filter Implementation

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

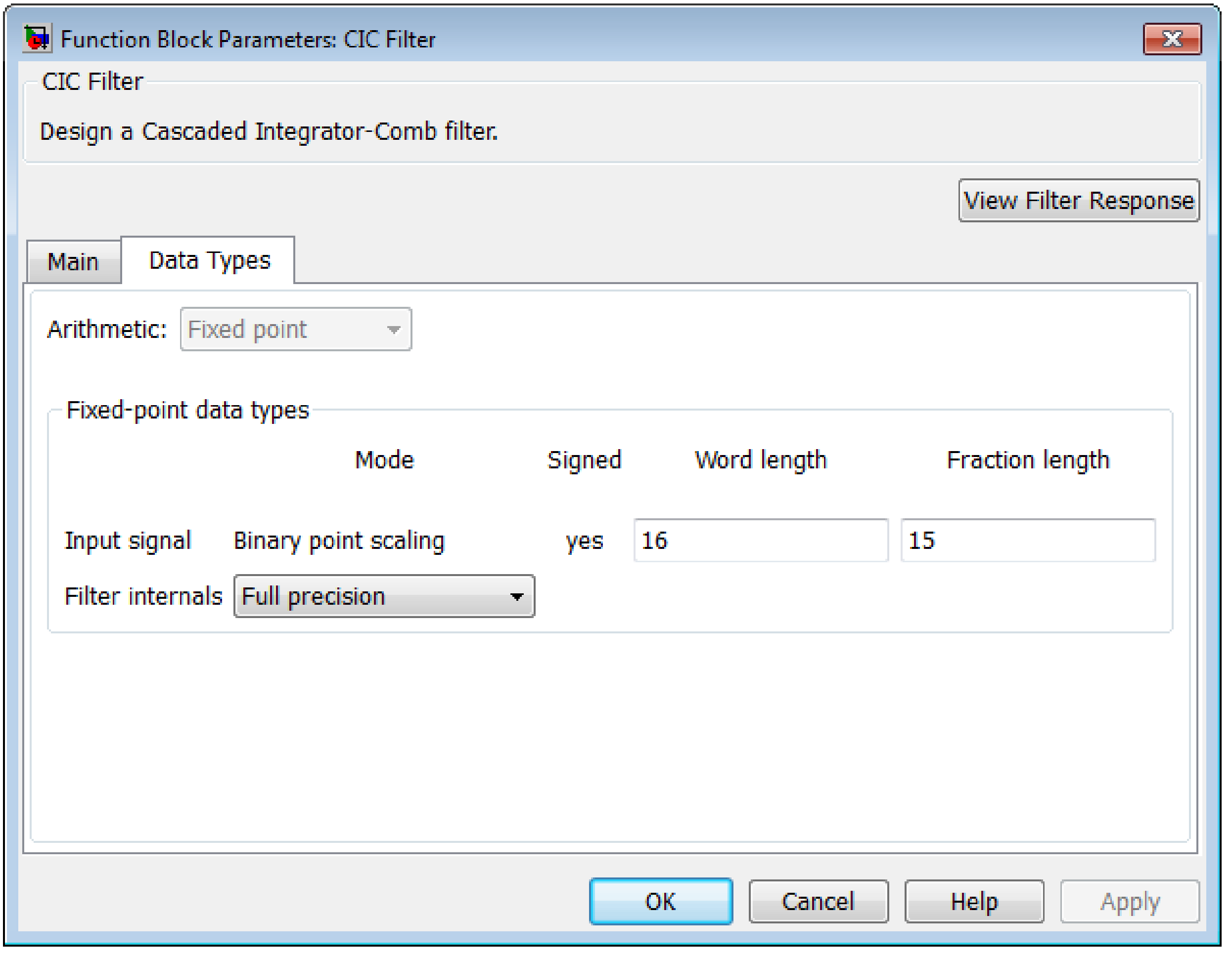
The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Data Types Pane



See the Data Types Pane subsection of the `filterBuilder` function reference page for more information about specifying data type parameters.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Fixed point

Introduced in R2007b

CIC Interpolation

Interpolate signal using Cascaded Integrator-Comb filter

Library

Filtering / Multirate Filters

`dspmlti4`

Description

The CIC Interpolation block performs a sample rate increase (interpolation) on an input signal by an integer factor. Cascaded Integrator-Comb (CIC) filters are a class of linear phase FIR filters that consist of a comb part and an integrator part.

The transfer function of a CIC interpolator filter is

$$H(z) = {H_I}^N(z){H_C}^N(z) = \frac{(1-z^{-RM})^N}{(1-z^{-1})^N} = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N$$

where

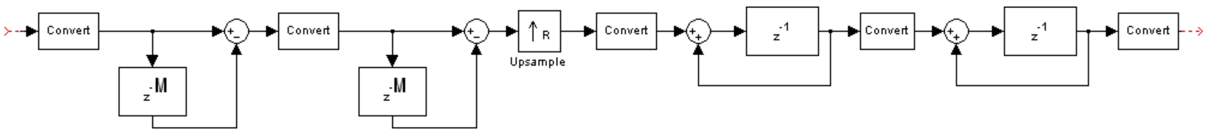
- H_I is the transfer function of the integrator part of the filter.
- H_C is the transfer function of the comb part of the filter.
- N is the number of sections. The number of sections in a CIC filter is defined as the number of sections in either the comb part *or* the integrator part of the filter. This value does not represent the total number of sections throughout the entire filter.
- R is the interpolation factor.
- M is the differential delay.

The CIC Interpolation block requires a Fixed-Point Designer license.

The block supports real and complex fixed-point inputs. In its normal mode of operation, the CIC Interpolation block allows the adder's numeric values to overflow and wrap around [1] [3]. The Fixed-Point infrastructure then causes overflow warnings to appear on the command line. This overflow is of no consequence.

CIC Filter Structure

The CIC Interpolation block has the following CIC filter structure. The structure consists of *N* sections of cascaded comb filters, followed by a rate change by a factor *R*, followed by *N* sections of cascaded integrators [1] .



The unit delay in the integrator portion of the CIC Filter can be located in either the feed-forward or the feedback path. These two configurations yield identical filter frequency response. However, the numerical outputs from these two configurations are different due to the latency. This block puts the unit delay in the feed forward path of the integrator since it is a preferred configuration for HDL implementation.

Dialog Box

Coefficient Source

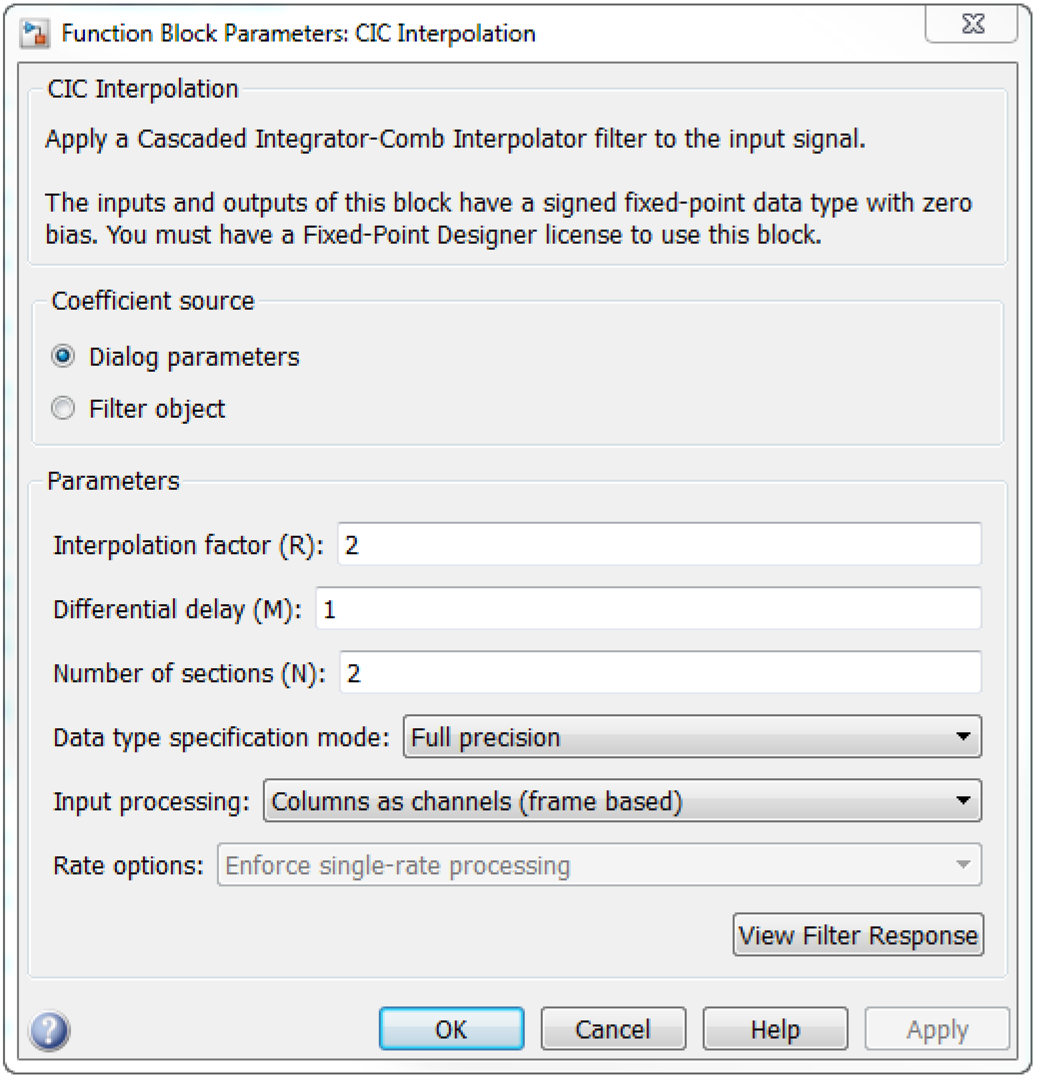
The CIC Interpolation block can operate in two different modes. Select the mode in the **Coefficient source** group box. If you select:

- **Dialog parameters** — Enter information about the filter, such as **Interpolation factor (R)**, **Differential delay (M)** and **Number of sections (N)** in the block mask.
- **Filter object** — Specify the filter using a dsp.CICInterpolator System object.

Different items appear on the CIC Interpolation block dialog depending on whether you select **Dialog parameters** or **Filter object** in the **Coefficient source** group box. See the following sections for details:

Specify Filter Characteristics in Dialog

The **Main** pane of the CIC Interpolation block dialog appears as follows when you select **Dialog parameters** in the **Coefficient source** group box.



Interpolation factor (R)

Specify the interpolation factor of the filter.

Differential delay (M)

Specify the differential delay of the comb portion of the filter, `M`, as shown in the diagram in "CIC Filter Structure" on page 2-325.

Number of sections (N)

Specify the number of filter sections. The number you specify determines the number of sections in either the comb part of the filter or in the integrator part of the filter. This value does not represent the total number of sections in the comb and integrator parts combined.

Data type specification mode

Choose how you specify the fixed-point word length and fraction length of the filter sections and/or output.

- `Full precision` — In this mode, the word and fraction lengths of the filter sections and outputs are automatically selected for you. The output and last section word lengths are set to

$$\text{word length} = \text{ceil}(\log_2(\frac{(R*M)^N}{R})) + I$$

  where

  - I = input word length
  - M = differential delay
  - N = number of sections
  - R = interpolation factor

  The other section word lengths are set is such a way as to accommodate the bit growth, as described in Hogenauer's paper [1]. All fraction lengths are set to the input fraction length.

- `Minimum section word lengths` — In this mode, you specify the word length of the filter output in the **Output word length** parameter. The word lengths of the filter sections are set in the same way as in `Full precision` mode.

  The section fraction lengths are set to the input fraction length. The output fraction length is set to the input fraction length minus the difference between the last section and output word lengths.

- `Specify word lengths` — In this mode you specify the word lengths of the filter sections and output in the **Section word lengths** and **Output word length** parameters. The fraction lengths of the filter sections are set such that the spread between word length and fraction length is the same as in full-precision mode. The output fraction length is set to the input fraction length minus the difference between the last section and output word lengths.
- `Binary point scaling` — In this mode you fully specify the word and fraction lengths of the filter sections and output in the **Section word lengths**, **Section fraction lengths**, **Output word length**, and **Output fraction length** parameters.

Section word lengths

Specify the word length, in bits, of the filter sections.

This parameter is only visible if you select `Specify word lengths` or `Binary point scaling` for the **Data type specification mode** parameter.

Section fraction lengths

Specify the fraction length of the filter sections.

This parameter is only visible if you select `Binary point scaling` for the **Data type specification mode** parameter.

Output word length

Specify the word length, in bits, of the filter output.

This parameter is only visible if you select `Minimum section word lengths`, `Specify word lengths` or `Binary point scaling` for the **Data type specification mode** parameter.

Output fraction length

Specify the fraction length of the filter output.

This parameter is only visible if you select `Binary point scaling` for the **Data type specification mode** parameter.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel. In this mode, the block always performs single-rate processing.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel. In this mode, the input to the block must be a scalar or a vector. You can use the **Rate options** parameter to specify whether the block performs single-rate or multirate processing.

Rate options

Specify the rate processing rule for the block. You can select one of the following options:

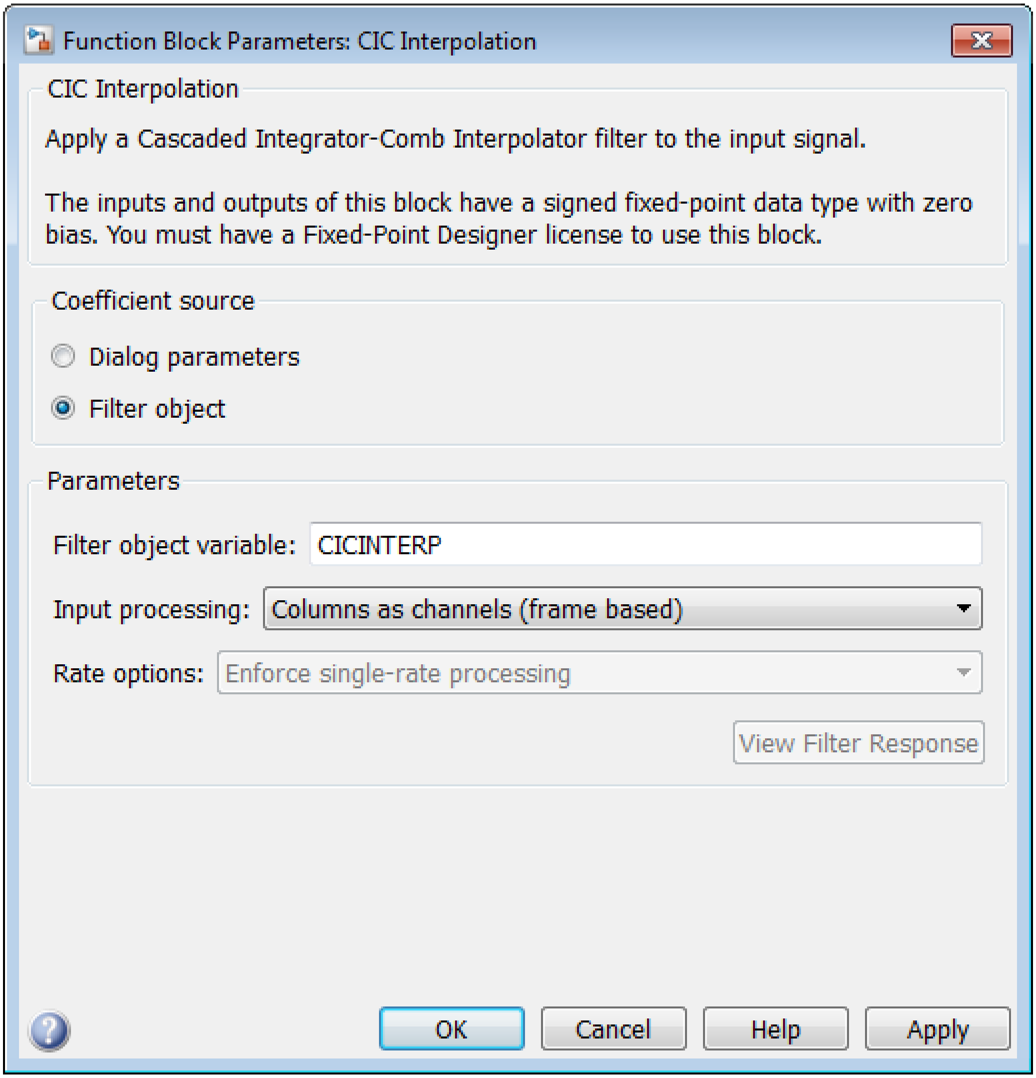
- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block produces an output with a sample rate that is `R` times faster than the input sample rate. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the filter defined in the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

Specify Multirate Filter Object

The **Main** pane of the CIC Interpolation block dialog appears as follows when you specify **Filter object** in the **Coefficient source** group box.



Multirate filter variable

Specify the name of the multirate filter object that you want the block to implement. You must specify the filter as a dsp.CICInterpolator System object.

You can define the System object in the block mask or in a MATLAB workspace variable.

For information on creating System objects, see “Define Basic System Objects” (MATLAB).

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel. In this mode, the block always performs single-rate processing.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel. In this mode, the input to the block must be a scalar or a vector. You can use the **Rate options** parameter to specify whether the block performs single-rate or multirate processing.

Rate options

Specify the rate processing rule for the block. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block produces an output with a sample rate that is `R` times faster than the input sample rate. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the System object specified in the **Multirate filter variable** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note: If you specify a filter in the **Multirate filter variable** parameter, you must apply the filter by clicking the **Apply** button before using the **View filter response** button.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see CIC Interpolation in the HDL Coder documentation.

References

[1] Hogenauer, E.B., “An Economical Class of Digital Filters for Decimation and Interpolation,” *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-29(2): pp. 155–162, 1981.

[2] Meyer-Baese, U., *Digital Signal Processing with Field Programmable Gate Arrays*, Springer Verlag, 2001.

[3] Harris, Fredric J., *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.

Supported Data Types

- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

See Also

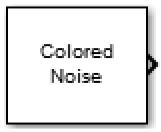
`CIC Decimation`	DSP System Toolbox
`FIR Decimation`	DSP System Toolbox
`FIR Interpolation`	DSP System Toolbox
`Digital Down-Converter`	DSP System Toolbox
`Digital Up-Converter`	DSP System Toolbox
dsp.CICDecimator	DSP System Toolbox
dsp.CICInterpolator	DSP System Toolbox

dsp.FIRDecimator	DSP System Toolbox
dsp.FIRInterpolator	DSP System Toolbox
dsp.CICCompensationDecimator	DSP System Toolbox
dsp.CICCompensationInterpolator	DSP System Toolbox
dsp.FIRHalfbandDecimator	DSP System Toolbox
dsp.FIRHalfbandInterpolator	DSP System Toolbox
`filter`	DSP System Toolbox

Introduced before R2006a

Colored Noise

Generate colored noise signal



Library

Sources

`dspsrcs4`

Description

The Colored Noise block generates a colored noise signal with a power spectral density of $1/|f|^{\alpha}$ over its entire frequency range. The inverse power spectral density component, α, can be any value in the interval `[-2 2]`. The type of colored noise the block generates depends on the **Noise color** option you choose in the block dialog box. When you set **Noise color** to `custom`, you can specify the power density of the noise through the **Power of inverse frequency** parameter.

The size and data type properties of the generated signal are determined by the parameters set in the block dialog box.

Algorithms

This block brings the capabilities of the dsp.ColoredNoise System object to the Simulink environment.

For information on the various colored noise processes, see the “Definitions” on page 4-424 section of `dsp.ColoredNoise`. For information on the algorithm used by this block, see the “Algorithms” on page 4-423 section of `dsp.ColoredNoise`.

Parameters

Noise color

Color of the noise the block generates. You can set this parameter to:

- pink (default) — Generates pink noise. This option is equivalent to setting **Power of inverse frequency** to 1.
- white — Generates white noise (flat power spectral density). This option is equivalent to setting **Power of inverse frequency** to 0.
- brown — Generates brown noise. This option is equivalent to setting **Power of inverse frequency** to 2.
- blue — Generates blue noise. This option is equivalent to setting **Power of inverse frequency** to -1.
- purple — Generates violet (purple) noise. This option is equivalent to setting **Power of inverse frequency** to -2.
- custom — Specify the power density of the noise through **Power of inverse frequency**.

Power of inverse frequency

Inverse power spectral density component, a, specified as a real scalar in the interval [-2 2]. The inverse exponent defines the power spectral density of the random process by $1/|f|^{\alpha}$. The default value of this property is 1. When **Power of inverse frequency** is greater than 0, the block generates lowpass noise, with a singularity (pole) at $f = 0$. These processes exhibit long memory. When **Power of inverse frequency** is less than 0, the block generates highpass noise with negatively correlated increments. These processes are referred to as antipersistent. In a log-log plot of power as a function of frequency, processes generated by the Colored Noise block exhibit an approximate linear relationship, with the slope equal to $–a$.

This parameter applies when you set **Noise color** to custom.

Number of output channels

Number of output channels, specified as a positive integer scalar. This parameter defines the number of columns in the generated signal. The default is 1. This parameter is nontunable.

Output data type

Data type of the output. You can set this parameter to:

- `double` (default)
- `single`

This parameter is nontunable.

Number of samples per output channel

Number of samples in each frame of the output signal, specified as a positive integer scalar. This parameter defines the number of rows in the generated signal. The default is `1024`. This parameter is nontunable.

Output sample time (s)

Sample time of the output signal, specified as a positive scalar in seconds. The default is `1`. This parameter is nontunable.

Initial seed

Initial seed of the random number generator algorithm, specified as a real positive integer scalar. The default is `67`. This parameter is nontunable.

Simulate using

Type of simulation to run. You can set this parameter to:

- `Interpreted execution` (default)

  Simulate model using the MATLAB interpreter. This option shortens startup time.
- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

References

[1] Beran, J., Feng, Y., Ghosh, S., and Kulik, R. *Long-Memory Processes: Probabilistic Properties and Statistical Methods*. Springer, 2013.

[2] Kasdin, N.J. "Discrete Simulation of Colored Noise and Stochastic Processes and $1/f^{\alpha}$ Power Law Noise Generation". *Proceedings of the IEEE*. Vol. 83, No. 5, 1995, pp. 802–827.

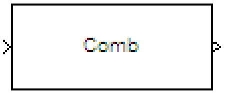
See Also

dsp.ColoredNoise	DSP System Toolbox
`randn`	DSP System Toolbox

Introduced in R2015a

Comb Filter

Design comb Filter



Library

Filtering / Filter Designs

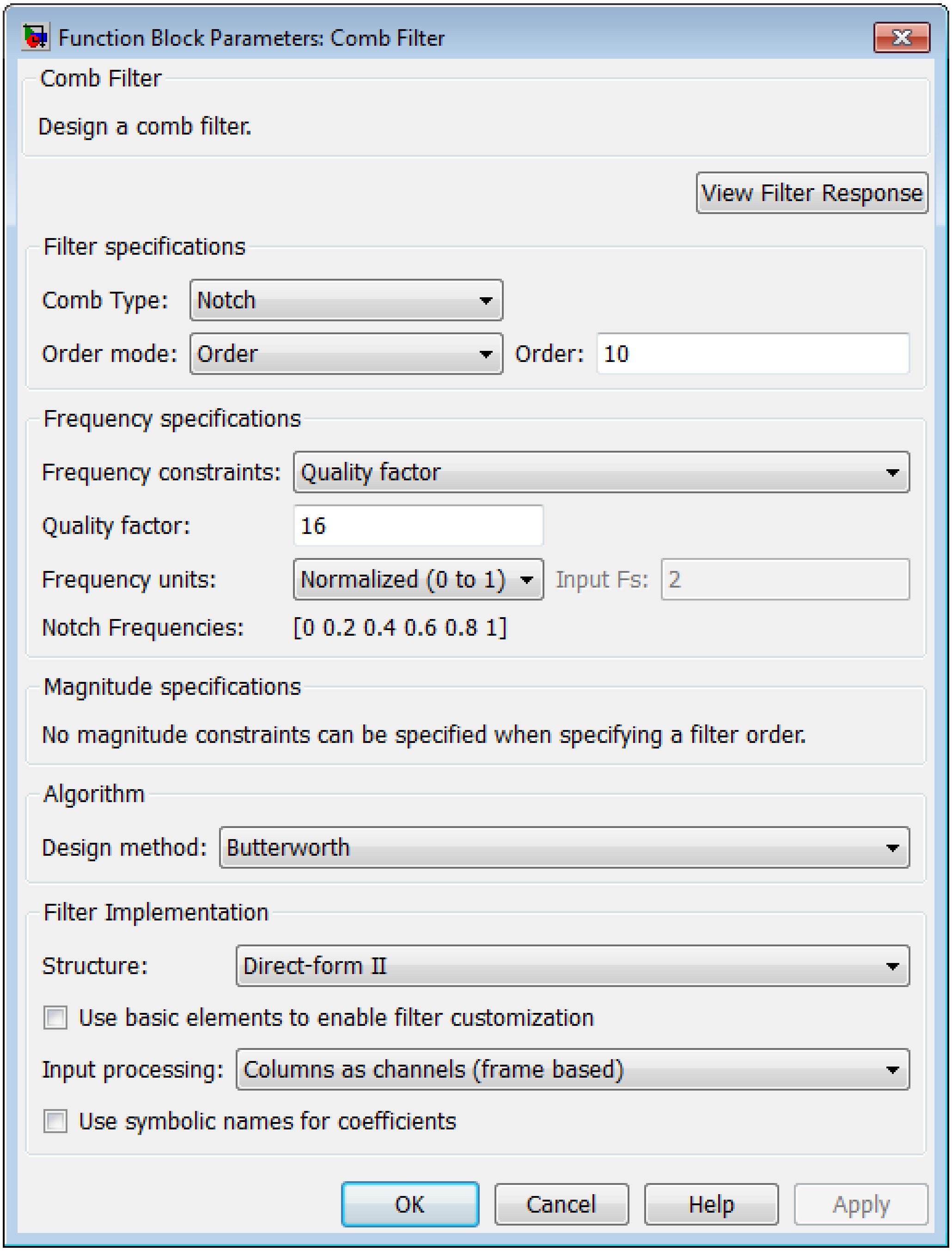
`dspfdesign`

Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See "Comb Filter Design —Main Pane" on page 5-739 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.


Function Block Parameters: Comb Filter
Comb Filter
Design a comb filter.
View Filter Response
Filter specifications
Comb Type: Notch
Order mode: Order
Order: 10
Frequency specifications
Frequency constraints: Quality factor
Quality factor: 16
Frequency units: Normalized (0 to 1)
Input Fs: 2
Notch Frequencies: [0 0.2 0.4 0.6 0.8 1]
Magnitude specifications
No magnitude constraints can be specified when specifying a filter order.
Algorithm
Design method: Butterworth
Filter Implementation
Structure: Direct-form II
Use basic elements to enable filter customization
Input processing: Columns as channels (frame based)
Use symbolic names for coefficients
OK
Cancel
Help
Apply

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify the type of comb filter and the number of peaks or notches.

Comb Type

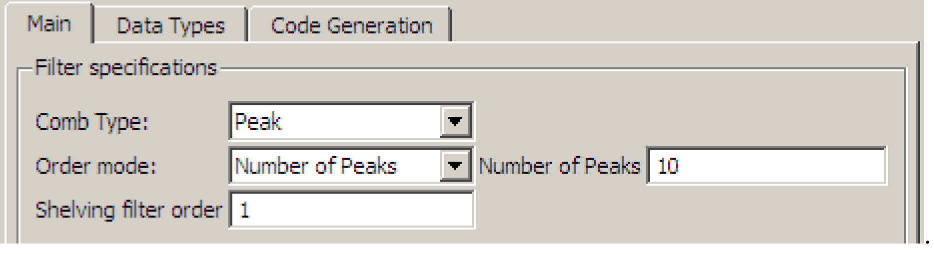
Select either `Notch` or `Peak` from the drop-down list. `Notch` creates a comb filter that attenuates a set of harmonically related frequencies. `Peak` creates a comb filter that amplifies a set of harmonically related frequencies.

Order mode

Select either `Order` or `Number of Peaks/Notches` from the drop-down menu.

Select `Order` to enter the desired filter order in the dialog box. The comb filter has notches or peaks at increments of 2/`Order` in normalized frequency units.

Select `Number of Peaks` or `Number of Notches` to specify the number of peaks or notches and the `Shelving filter order`



.

Shelving filter order

The `Shelving filter order` is a positive integer that determines the sharpness of the peaks or notches. Larger values result in sharper peaks or notches.

Frequency specifications

In this group, you specify the frequency constraints and frequency units.

Frequency specifications

Select either `Quality factor` or `Bandwidth`.

`Quality factor` is the ratio of the center frequency of the peak or notch to the bandwidth calculated at the –3 dB point.

`Bandwidth` specifies the bandwidth of the peak or notch. By default the bandwidth is measured at the –3 dB point. For example, setting the bandwidth equal to 0.1 results in 3 dB frequencies at normalized frequencies 0.05 above and below the center frequency of the peak or notch.

Frequency Units

Specify the frequency units. The default is normalized frequency. Choosing an option in Hz enables the **Input Fs** dialog box.

Magnitude specifications

Specify the units for the magnitude specification and the gain at which the bandwidth is measured. This menu is disabled if you specify a filter order. Select one of the following magnitude units from the drop down list:

- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Bandwidth gain — Specify the gain at which the bandwidth is measured. The default is –3 dB.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

The IIR Butterworth design is the only option for peaking or notching comb filters.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

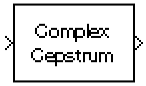
Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2010a

Complex Cepstrum

Compute complex cepstrum of input



Library

Transforms

`dspxfrm3`

Description

The Complex Cepstrum block computes the complex cepstrum of each column in the real-valued M-by-N input matrix, u. The block treats each column of the input as an independent channel containing M consecutive samples. The block *always* processes unoriented vector inputs as a single channel, and returns the result as a length-M column vector. The block does not accept complex-valued inputs.

The input is altered by the application of a linear phase term so that there is no phase discontinuity at ±π radians. That is, each input channel is independently zero padded and circularly shifted to have zero phase at π radians.

The output is a real M_o-by-N matrix, where M_o is specified by the **FFT length** parameter. Each output column contains the length-M_o complex cepstrum of the corresponding input column.

```
y = cceps(u,Mo)     % Equivalent MATLAB code
```

When you select the **Inherit FFT length from input port dimensions** check box, the output frame size matches the input frame size ($M_o = M$).

The output port rate is the same as the input port rate.

Parameters

Inherit FFT length from input port dimensions

When you select this check box, the output frame size matches the input frame size.

FFT length

The number of frequency points at which to compute the FFT, which is also the output frame size, M_o. This parameter is visible only when you clear the **Inherit FFT length from input port dimensions** check box.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

`DCT`	DSP System Toolbox
`FFT`	DSP System Toolbox
`Real Cepstrum`	DSP System Toolbox
`cceps`	Signal Processing Toolbox

Introduced before R2006a

Constant

Generate constant value

Library

Sources

`dspsrcs4`

Description

The Constant block is an implementation of the Simulink Constant block. See Constant for more information.

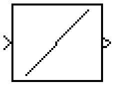
HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Constant.

Introduced in R2008b

Constant Ramp

Generate ramp signal with length based on input dimensions



Library

Signal Operations

`dspsigops`

Description

The Constant Ramp block generates the constant ramp signal

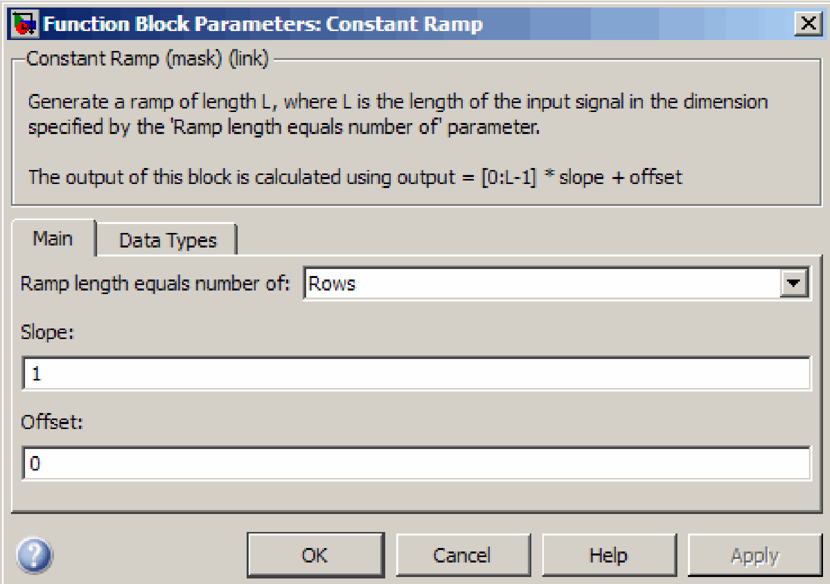
```
y = (0:L-1)*m + b
```

where m is the slope specified by the scalar **Slope** parameter, and b is the y-intercept specified by the scalar **Offset** parameter.

For an unoriented vector input, L is equal to the length of the input vector. For an N-D input array, the length L of the output ramp is equal to the length of the input in the dimension specified by the **Ramp length equals number of** or **Dimension** parameter. The output, y, is always an unoriented vector.

Dialog Box

The **Main** pane of the Constant Ramp block dialog appears as follows.



Ramp length equals number of

Specify whether the length of the output ramp is the number of rows, number of columns, or the length of the specified dimension of the input.

Dimension

Specify the one-based dimension of the input array that determines the length of the output ramp.

This parameter is only visible when you select `Elements in specified dimension` for the **Ramp length equals number of** parameter.

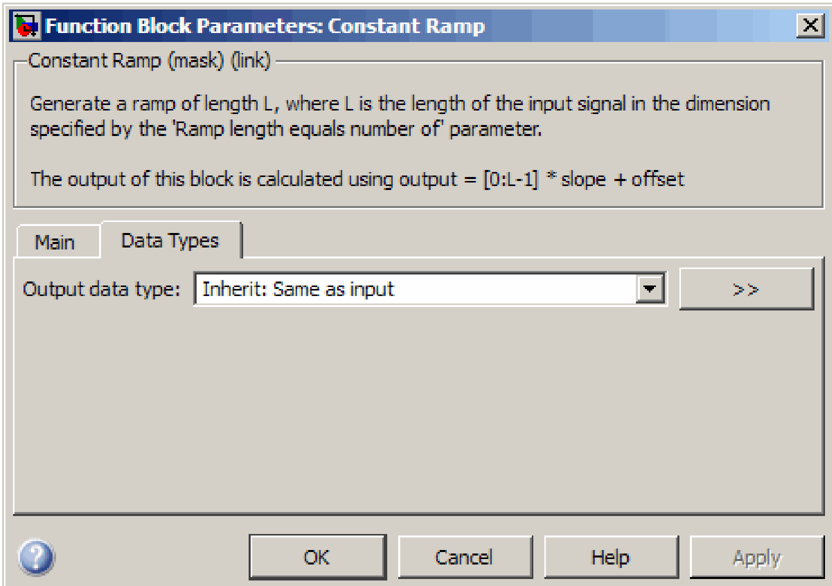
Slope

Specify the scalar slope of the ramp.

Offset

Specify the scalar *y*-intercept of the ramp.

The **Data Types** pane of the Constant Ramp block dialog appears as follows.



Output data type

Specify the output data type for this block. You can select one of the following:

- A rule that inherits a data type, for example, `Inherit: Same as input`.
- A built in data type, such as `double`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

- Fixed point
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

This block differs from other DSP System Toolbox blocks in that unless you choose `Same as input` for the **Output data type** parameter, the data types of the input and the output do not need to be the same.

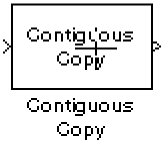
See Also

Create Diagonal Matrix	DSP System Toolbox
Constant	Simulink
Ramp	Simulink
Identity Matrix	DSP System Toolbox

Introduced before R2006a

Contiguous Copy (Obsolete)

Create discontiguous input in contiguous block of memory



Library

`dspobslib`

Description

Note The Contiguous Copy block is still supported but is likely to be obsoleted in a future release.

The Contiguous Copy block copies the input to a contiguous block of memory, and passes this new copy to the output. The output is identical to the input, but is guaranteed to reside in a contiguous section of memory.

Because Simulink software employs an efficient copy-by-reference method for propagating data in a model, some operations produce outputs with discontiguous memory locations.

Although this does not present a problem during simulation, blocks linked to versions of DSP Blockset prior to 4.0 may require contiguous inputs for code generation with the Simulink Coder product. When such blocks are used in a model intended for code generation, they should be preceded by the Contiguous Copy block to ensure that their inputs are contiguous.

Supported Data Types

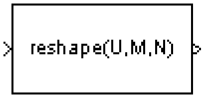
- Double-precision floating point

- Single-precision floating point
- Fixed point
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

Introduced in R2008b

Convert 1-D to 2-D

Reshape 1-D or 2-D input to 2-D matrix with specified dimensions



Library

Signal Management / Signal Attributes

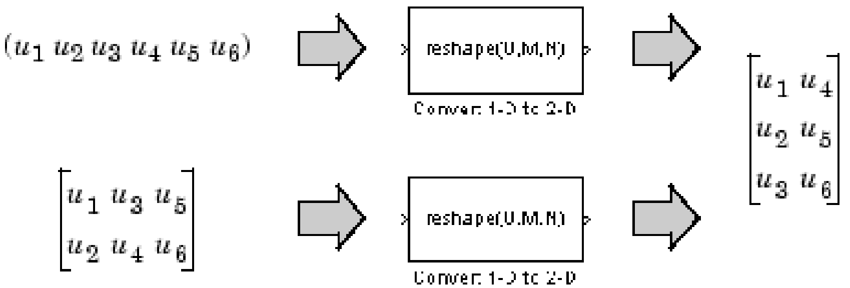
`dspsigattribs`

Description

The Convert 1-D to 2-D block reshapes a length-M_i 1-D vector or an M_i-by-N_i matrix to an M_o-by-N_o matrix, where M_o is specified by the **Number of output rows** parameter, and N_o is specified by the **Number of output columns** parameter.

```
y = reshape(u,Mo,No)        % Equivalent MATLAB code
```

The input is reshaped *columnwise*, as shown in the two cases below. The length-6 vector and the 2-by-3 matrix are both reshaped to the same 3-by-2 output matrix.



An error is generated when $(M_o*N_o) \neq (M_i*N_i)$. That is, the total number of input elements must be conserved in the output.

The output is frame based when you select the **Frame-based output** check box; otherwise, the output is sample based.

Parameters

Number of output rows

The number of rows, M_o, in the output matrix.

Number of output columns

The number of rows, N_o, in the output matrix.

Frame-based output

Creates a frame-based output when selected.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Convert 1-D to 2-D in the HDL Coder documentation.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

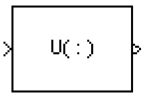
See Also

Buffer	DSP System Toolbox
Convert 2-D to 1-D	DSP System Toolbox
Frame Conversion	DSP System Toolbox
Reshape	Simulink
Submatrix	DSP System Toolbox

Introduced before R2006a

Convert 2-D to 1-D

Convert 2-D matrix input to 1-D vector



Library

Signal Management / Signal Attributes

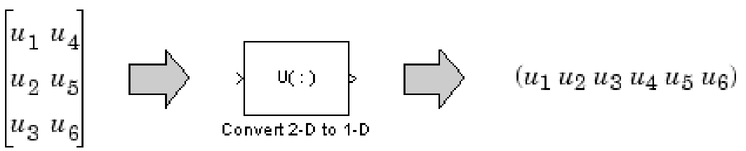
`dspsigattribs`

Description

The Convert 2-D to 1-D block reshapes an M-by-N matrix input to a 1-D vector with length $M*N$.

```
y = u(:)     % Equivalent MATLAB code
```

The input is reshaped *columnwise*, as shown below for a 3-by-2 matrix.



Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean

Port	Supported Data Types
	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated

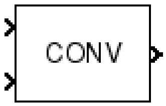
See Also

Buffer	DSP System Toolbox
Convert 1-D to 2-D	DSP System Toolbox
Frame Status Conversion (Obsolete)	DSP System Toolbox
Reshape	Simulink
Submatrix	DSP System Toolbox

Introduced before R2006a

Convolution

Convolution of two inputs



Library

Signal Operations

`dspsigops`

Description

The Convolution block convolves the first dimension of an N-D input array *u*, with the first dimension of an N-D input array *v*. The block can also independently convolve a column vector with the first-dimension of an N-D input array.

Convolution with DSP System Toolbox Blocks

The general equation for convolution is

$$y(k) = \sum_{n} u(n-k)h(k)$$

There are two DSP System Toolbox blocks that can be used for this purpose:

- Convolution
- `Discrete FIR Filter`

The Convolution block assumes that all of u and h are available at each Simulink time step, and computes the entire convolution at every one.

The Discrete FIR Filter block can be used for convolving signals in situations where all of h is available at each time step, but u is a sequence that comes in over the life of the simulation. When you use the Discrete FIR Filter block, the convolution is computed only once.

Use the following questions to help you determine which block best fits your needs:

Question	Answer	Recommended Block(s)
How many convolutions do you intend to perform?	Many convolutions, one at each time step	• Convolution block
	One convolution over the life of the simulation	• Convolution block • Discrete FIR Filter block
How long are your input sequences?	Both sequences have a finite length	• Convolution block • Discrete FIR Filter block
	One sequence has an infinite (not predetermined) length	• Discrete FIR Filter block
How many of the inputs are scalar streams?	None	• Convolution block • Discrete FIR Filter block
	One or both	• Buffer block followed by the Convolution block • Discrete FIR Filter block

Convolving Two N-D Arrays

The block always computes the convolution of two N-D input arrays along the first dimension. When both inputs are N-D arrays, the size of their first dimension can differ, but the size of all other dimensions must be equal. For example, when u is an M_u-by-N-by-P array, and v is an M_v-by-N-by-P array, the output is an (M_u+M_v-1)-by-N-by-P array.

When the input to the Convolution block is a M_u-by-N matrix u and an M_v-by-N matrix v, the output, y, is a (M_u+M_v-1)-by-N matrix whose jth column has the following elements

$$y_{i,j} = \sum_{k=0}^{\max(M_u,M_v)-1} u_{k,j} v_{(i-k),j} \quad 0 \le i \le (M_u + M_v - 2)$$

Inputs u and v are zero when indexed outside of their valid ranges. When both inputs are real, the output is real; when one or both inputs are complex, the output is complex.

Convolving a Column Vector with an N-D Array

When one input is a column vector and the other is an N-D array, the block independently convolves the vector with the first dimension of the N-D input array. For example, when u is a M_u-by-1 column vector and v is an M_v-by-N matrix, the output is an (M_u+M_v-1)-by-N matrix whose jth column has the following elements

$$y_{i,j} = \sum_{k=0}^{\max(M_u,M_v)-1} u_k v_{(i-k),j} \quad 0 \le i \le (M_u + M_v - 2)$$

Convolving Two Column Vectors

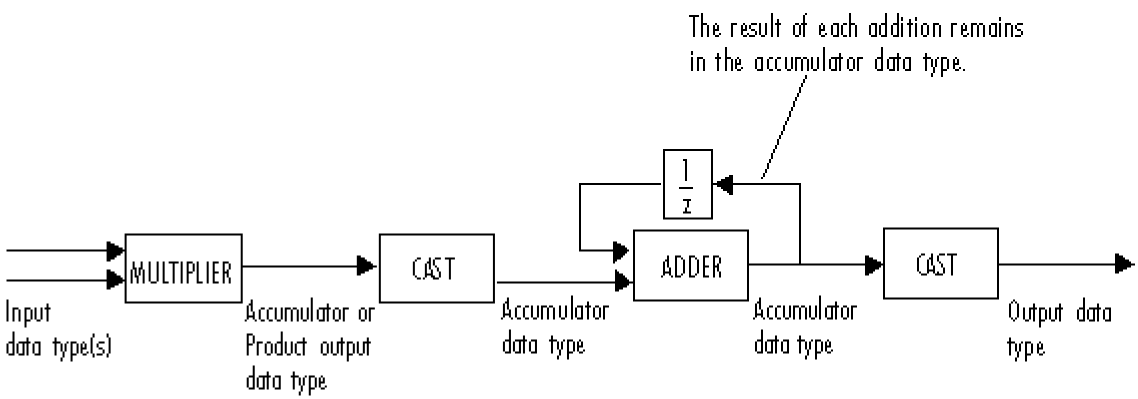
The Convolution block also accepts two column vector inputs. When u and v are column vectors with lengths M_u and M_v, the Convolution block performs the vector convolution

$$y_i = \sum_{k=0}^{\max(M_u,M_v)-1} u_k v_{(i-k)} \quad 0 \le i \le (M_u + M_v - 2)$$

The output is a (M_u+M_v-1)-by-1 column vector.

Fixed-Point Data Types

The following diagram shows the data types used within the Convolution block for fixed-point signals (time domain only).



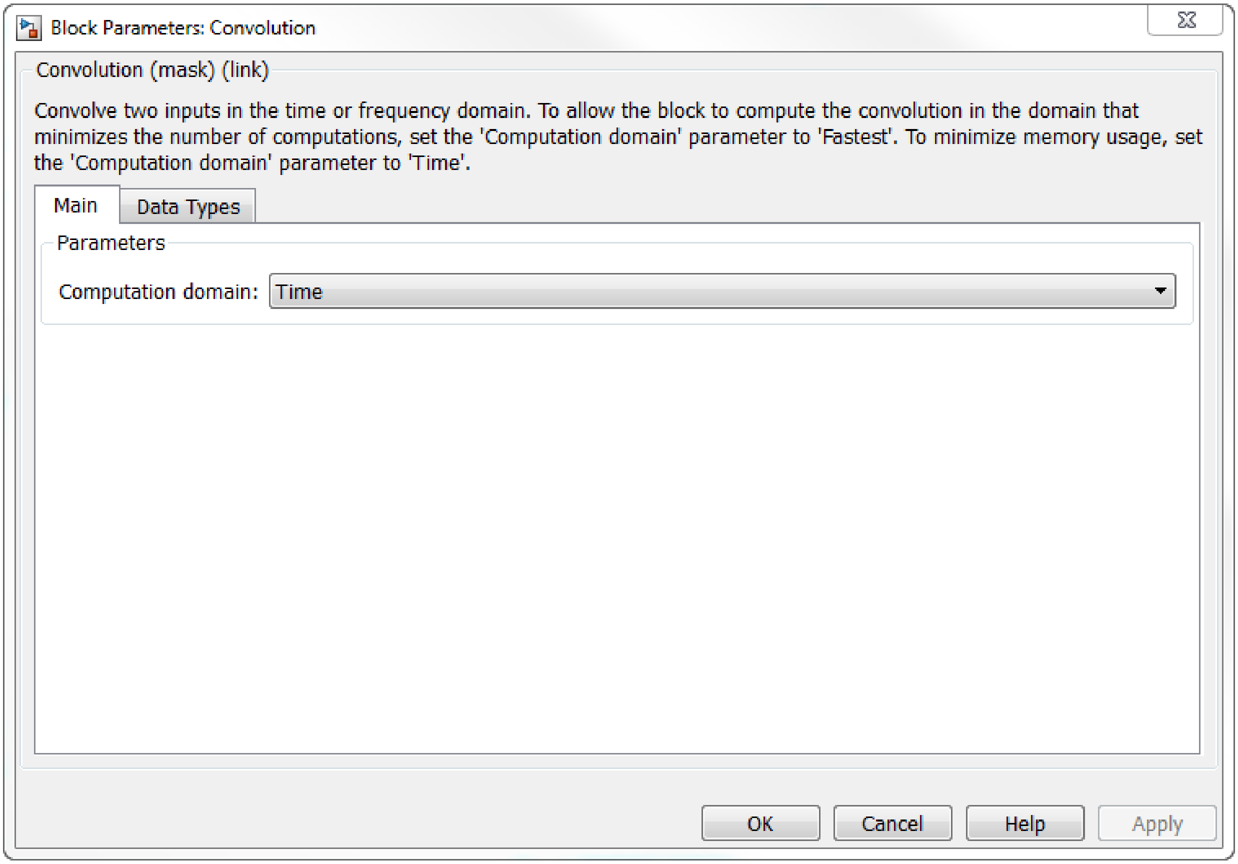
You can set the product output, accumulator, and output data types in the block dialog as discussed in the next section.

The output of the multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

Note: When one or both of the inputs are signed fixed-point signals, all internal block data types are signed fixed point. The internal block data types are unsigned fixed point only when *both* inputs are unsigned fixed-point signals.

Dialog Box

The **Main** pane of the Convolution block dialog appears as follows.

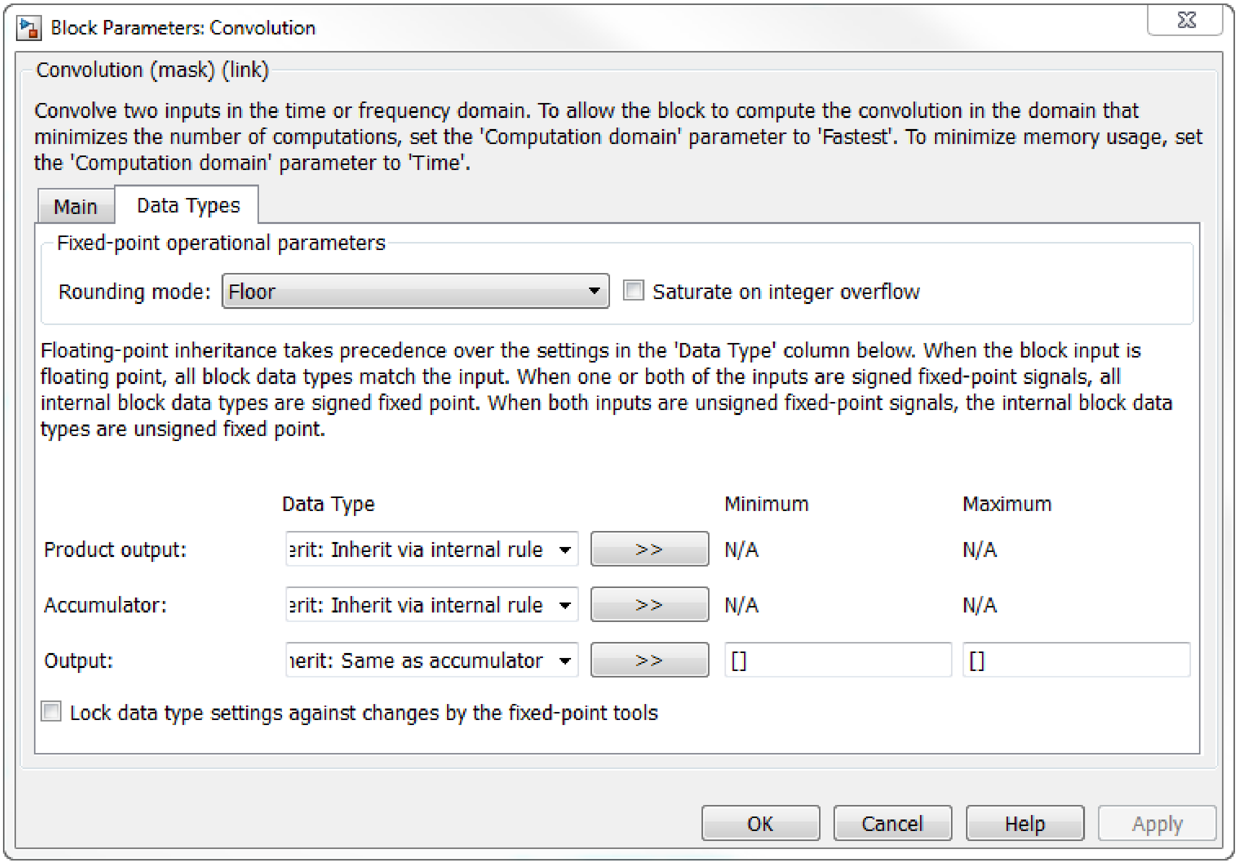


Computation domain

Set the domain in which the block computes convolutions:

- `Time` — The block computes in the time domain, which minimizes memory use.
- `Frequency` — The block computes in the frequency domain, which might require fewer computations than computing in the time domain, depending on the input length.
- `Fastest` — The block computes in the domain, which minimizes the number of computations.

The **Data Types** pane of the Convolution block dialog appears as follows.



Note: Fixed-point signals are only supported for the time domain. To use the parameters on this pane, make sure `Time` is selected for the **Computation domain** parameter on the **Main** pane.

Rounding mode

Select the rounding mode for fixed-point operations.

Note: The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output data type** is `Inherit: Inherit via internal rule`
- **Accumulator data type** is `Inherit: Inherit via internal rule`
- **Output data type** is `Inherit: Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations.

Product output

Specify the product output data type. See “Fixed-Point Data Types” on page 2-360 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-360 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output

Specify the output data type. See “Fixed-Point Data Types” on page 2-360 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

See Also

Functions
`conv`

Blocks
`Correlation` | `Discrete FIR Filter`

Introduced before R2006a

Correlation

Cross-correlation of two *N*-D arrays
Library: DSP System Toolbox / Statistics

Description

The Correlation block computes the cross-correlation of two *N*-D input arrays along the first-dimension. The computation can be done in the time domain or frequency domain. You can specify the domain through the **Computation domain** parameter. In the time domain, the block convolves the first input signal, u, with the time-reversed complex conjugate of the second input signal, v. In the frequency domain, to compute the cross-correlation, the block:

1. Takes the Fourier transform of both input signals, U and V.
2. Multiplies U and V^*, where * denotes the complex conjugate.
3. Computes the inverse Fourier transform of the product.

If you set **Computation domain** to `Fastest`, the block chooses the domain that minimizes the number of computations. For information on these computation methods, see “Algorithms” on page 2-373.

Ports

Input

`Port_1` — First data input signal

vector | matrix | *N*-D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input can be a fixed-point signal when you set the **Computation domain** to `Time`. When one or both of the input signals are complex, the output signal is also complex.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed_point`
Complex Number Support: Yes

`Port_2` — Second data input signal
vector | matrix | *N*-D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input can be a fixed-point signal when you set the **Computation domain** to `Time`. When one or both of the input signals are complex, the output signal is also complex.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed_point`
Complex Number Support: Yes

Output

`Port_1` — Cross-correlated output
vector | matrix | *N*-D array

Cross-correlated output of the data input.

When the inputs are *N*-D arrays, the block outputs an *N*-D array, where all the dimensions, except for the first dimension, match with the input array. For example,

- When the inputs u and v have dimensions M_u-by-N-by-P and M_v-by-N-by-P, respectively, the Correlation block outputs an $(M_u + M_v - 1)$-by-N-by-P array.
- When the inputs u and v have the dimensions M_u-by-N and M_v-by-N, the block outputs an $(M_u + M_v - 1)$-by-N matrix.

If one input is a column vector and the other input is an *N*-D array, the Correlation block computes the cross-correlation of the vector with each column in the *N*-D array. For example,

- When the input u is an M_u-by-1 column vector and v is an M_v-by-N matrix, the block outputs an $(M_u + M_v - 1)$-by-N matrix.
- Similarly, when u and v are column vectors with lengths M_u and M_v, respectively, the block performs the vector cross-correlation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed_point`

Complex Number Support: Yes

Parameters

Main Tab

`Computation domain` — Domain in which the block computes the cross-correlation

`Time` (default) | `Frequency` | `Fastest`

- `Time` — Computes the cross-correlation in the time domain, which minimizes the memory usage.
- `Frequency` — Computes the cross-correlation in the frequency domain. For more information, see “Algorithms” on page 2-373.
- `Fastest` — Computes the cross-correlation in the domain that minimizes the number of computations.

To cross-correlate fixed-point signals, set this parameter to `Time`.

Data Types Tab

Note Fixed-point signals are supported for the time domain only. To use these parameters, on the **Main** tab, set **Computation domain** to `Time`.

`Rounding mode` — Method of rounding operation

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Select the rounding mode for fixed-point operations.

Note: The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numerical results when all these conditions are met:

- **Product output** data type is `Inherit: Inherit via internal rule`.
- **Accumulator** data type is `Inherit: Inherit via internal rule`.
- **Output** data type is `Inherit: Same as accumulator`.

With these data type settings, the block operates in full-precision mode.

`Saturate on integer overflow` — Saturate on integer overflow

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations.

`Product output` — Product output data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Same as input` | `fixdt([],16,0)`

Product output specifies the data type of the output of a product operation in the Correlation block. For more information on the product output data type, see "Multiplication Data Types" and the 'Fixed-Point Conversion' section in "Extended Capabilities" on page 2- .

- `Inherit: Inherit via internal rule` — The block inherits the product output data type based on an internal rule. For more information on this rule, see "Inherit via Internal Rule".
- `Inherit: Same as input` — The block specifies the product output data type to be the same as the input data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink).

`Accumulator` — Accumulator data type

`Inherit: Inherit via internal rule` (default) | `Inherit: Same as input` | `Inherit: Same as product output` | `fixdt([],16,0)`

Accumulator specifies the data type of output of an accumulation operation in the Correlation block. For illustrations on how to use the accumulator data type in this block, see the 'Fixed-Point Conversion' section in "Extended Capabilities" on page 2- .

- `Inherit: Inherit via internal rule` — The block inherits the accumulator data type based on an internal rule. For more information on this rule, see “Inherit via Internal Rule”.
- `Inherit: Same as input` — The block specifies the accumulator data type to be the same as the input data type.
- `Inherit: Same as product output` — The block specifies the accumulator data type to be the same as the product output data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink).

`Output` — Output data type

`Inherit: Same as accumulator` (default) | `Inherit: Same as input` | `Inherit: Same as product output` | `fixdt([],16,0)`

Output specifies the data type of the output of the Correlation block. For more information on the output data type, see the 'Fixed-Point Conversion' section in “Extended Capabilities” on page 2- .

- `Inherit: Same as input` — The block specifies the output data type to be the same as the input data type.
- `Inherit: Same as product output` — The block specifies the output data type to be the same as the product output data type.
- `Inherit: Same as accumulator` — The block specifies the output data type to be the same as the accumulator data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink).

`Output Minimum` — Minimum value block can output
[ ] (default) | scalar

Specify the minimum value the block can output. Simulink software uses this minimum value to perform:

- Simulation range checking. See "Signal Ranges" (Simulink).
- Automatic scaling of fixed-point data types.

`Output Maximum` — Maximum value the block can output
[ ] (default) | scalar

Specify the maximum value the block can output. Simulink software uses this maximum value to perform:

- Simulation range checking. See "Signal Ranges" (Simulink).
- Automatic scaling of fixed-point data types.

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block dialog box.

Definitions

Cross-Correlation

Cross-correlation is the measure of similarity of two discrete-time sequences as a function of the lag of one relative to the other.

For two length N deterministic inputs, or realizations of jointly WSS random processes, x and y, the cross-correlation is computed using the following relationship:

$$r_{xy}(h) = \begin{cases} \sum_{n=0}^{N-h-1} x(n+h)y^*(n) & 0 \le h \le N-1 \\ r_{yx}^*(h) & -(N-1) \le h \le 0 \end{cases}$$

where h is the lag and * denotes the complex conjugate. If the inputs are realizations of jointly WSS stationary random processes, $r_{xy}(h)$ is an unnormalized estimate of the theoretical cross-correlation:

$$\rho_{xy}(h) = E\{x(n+h)y^*(n)\}$$

where E{ } is the expectation operator.

Algorithms

Time domain Computation

When you set the computation domain to time, the algorithm computes the cross-correlation of two signals in the time domain. The input signals can be fixed-point signals in this domain.

Correlate Two 2-D Arrays

When the inputs are two 2-D arrays, the jth column of the output, $y_{u,v}$, has these elements:

$$y_{uv(i,j)} = \sum_{k=0}^{\max(M_u,M_v)-1} u^*_{k,j} v_{(k+i),j} \qquad 0 \le i < M_v$$

$$y_{uv(i,j)} = y^*_{vu(-i,j)} \qquad -M_u < i < 0$$

- * denotes the complex conjugate.
- u is an M_u-by-N input matrix.
- v is an M_v-by-N input matrix.
- $y_{u,v}$ is an $(M_u + M_v - 1)$-by-N matrix.

Inputs u and v are zero when indexed outside their valid ranges.

Correlate a Column Vector with a 2-D Array

When one input is a column vector and the other input is a 2-D array, the algorithm independently cross-correlates the input vector with each column of the 2-D array. The jth column of the output, $y_{u,v}$, has these elements:

$$y_{uv(i,j)} = \sum_{k=0}^{\max(M_u,M_v)-1} \overset{*}{u_k} v_{(k+i),j} \qquad 0 \le i < M_v$$

$$y_{uv(i,j)} = \overset{*}{y}_{vu(-i,j)} \qquad -M_u < i < 0$$

- * denotes the complex conjugate.
- u is an M_u-by-1 column vector.
- v is an M_v-by-N matrix.
- $y_{u,v}$ is an $(M_u + M_v - 1)$-by-N matrix.

Inputs u and v are zero when indexed outside their valid ranges.

Correlate Two Column Vectors

When the inputs are two column vectors, the jth column of the output, $y_{u,v}$, has these elements:

$$y_{uv(i)} = \sum_{k=0}^{\max(M_u,M_v)-1} \overset{*}{u_k} v_{(k+i)} \qquad 0 \le i < M_v$$

$$y_{uv(i)} = \overset{*}{y}_{vu(-i)} \qquad -M_u < i < 0$$

- * denotes the complex conjugate.
- u is an M_u-by-1 column vector.
- v is an M_v-by-1 column vector.
- $y_{u,v}$ is an $(M_u + M_v - 1)$-by-1 column vector.

Inputs u and v are zero when indexed outside their valid ranges.

Frequency Domain Computation

When you set the computation domain to frequency, the algorithm computes the cross-correlation in the frequency domain.

To compute the cross-correlation, the algorithm:

1. Takes the Fourier transform of both input signals, U and V.
2. Multiplies U and V^*, where * denotes the complex conjugate.
3. Computes the inverse Fourier transform of the product.

In this domain, depending on the input length, the algorithm can require fewer computations.

Extended Capabilities

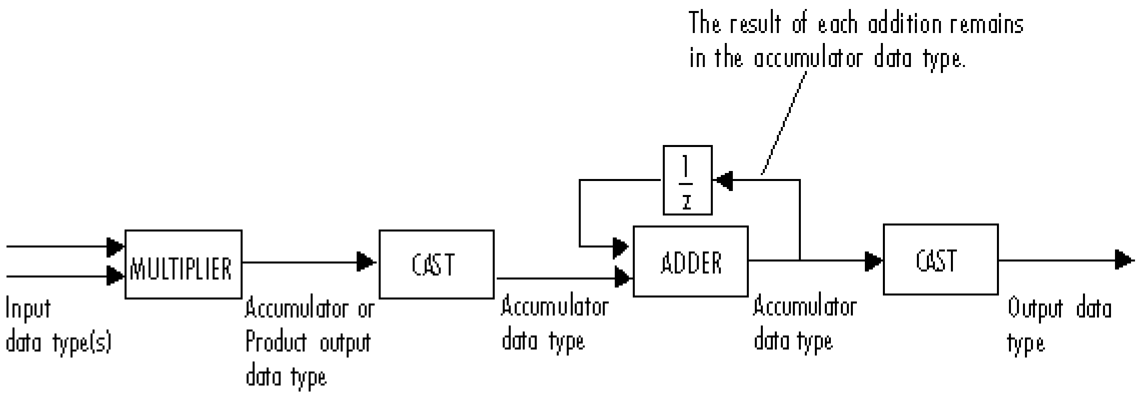
C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion

Design and simulate fixed-point algorithms using Fixed-Point Designer™.

The following diagram shows the data types the Correlation block uses for fixed-point signals (time domain only).



You can set the product output, accumulator, and output data types on the **Data Types** tab of the block.

When the input is real, the output of the multiplier is in the product output data type. When the input is complex, the output of the multiplier is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

Note: When one or both of the inputs are signed fixed-point signals, all internal block data types are signed fixed point. The internal block data types are unsigned fixed point only when both inputs are unsigned fixed-point signals.

See Also

See Also

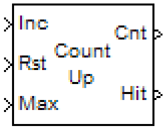
System Objects
dsp.Crosscorrelator | dsp.Autocorrelator

Blocks
`Autocorrelation`

Introduced before R2006a

Counter

Count up or down through specified range of numbers



Library

Signal Management / Switches and Counters

`dspswit3`

Description

The Counter block counts up or down through a specified range of numbers. The block enables the Inc (increment) port when you set the **Count direction** parameter to `Up`. When you set the **Count direction** parameter to `Down`, the block enables the Dec (decrement) port. If you set the **Count event** parameter to `Free running`, the block disables the Inc or Dec port, and counts at a constant time interval. For all other settings of the **Count event** parameter, the block increments or decrements the counter each time a trigger event occurs at the Inc or Dec input port. When a trigger event occurs at the optional Rst port, the block resets the counter to its initial state.

The Counter block accepts single-channel inputs. The Inc and Dec ports accept real-valued scalars or vectors. If the input to the Inc or Dec port is a vector, the block treats the vector as a frame. The Rst port only accepts real-valued scalars. The Rst port must have the same port sample time as the Inc or Dec input port. If you enable the optional Max input port, you must provide an unsigned integer input that the **Count data type** can represent.

See the following topics for more information:

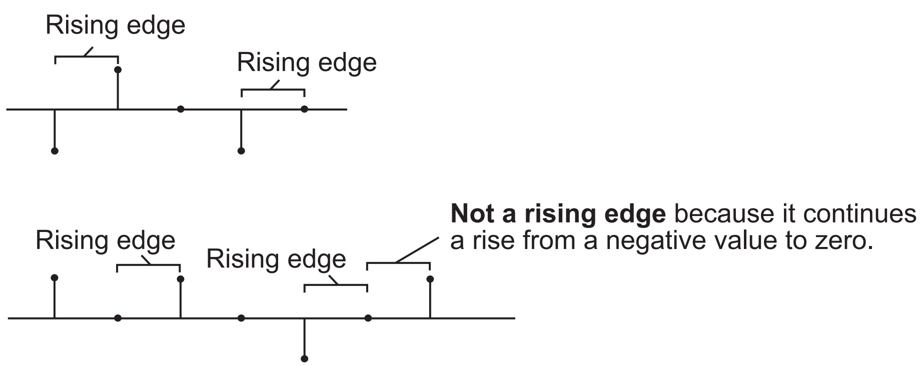
- "Setting the Count Event Parameter" on page 2-378
- "Setting the Counter Size and Initial Count Parameters" on page 2-379

- “Scalar Input Operation” on page 2-380
- “Vector Input Operation” on page 2-381
- “Free-Running Operation” on page 2-381

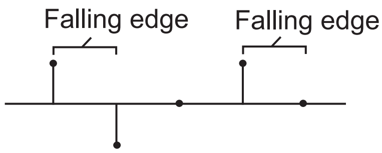
Setting the Count Event Parameter

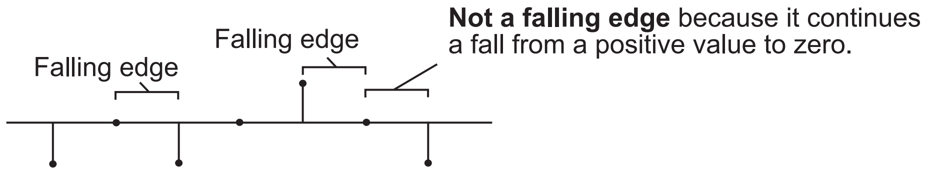
Specify the trigger event for the Inc/Dec and Rst ports by setting the **Count event** parameter to one of the following values:

- `Rising edge` — Triggers a count or reset operation when the input to the Inc/Dec or Rst port behaves in one of the following ways:
  - Rises from a negative value to a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure).



- `Falling edge` — Triggers a count or reset operation when the input to the Inc/Dec or Rst port behaves in one of the following ways:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure).





- `Either edge` — Triggers a count or reset operation when the input to the Inc/Dec or Rst port is a `Rising edge` or `Falling edge`.
- `Non-zero sample` — Triggers a count or reset operation at each sample time when the input to the Inc/Dec or Rst port is not zero.
- `Free running` — Disables the Inc/Dec port and enables the **Samples per output frame** and **Sample time** block parameters. The block increments or decrements the counter at a constant interval, T_s, which you specify using the **Sample time** parameter. For more information, see "Free-Running Operation" on page 2-381. In this mode, the block resets the counter whenever it receives a non-zero sample at the Rst port.

Setting the Counter Size and Initial Count Parameters

At the start of the simulation, the block sets the counter to the value you specify in the **Initial count** parameter. The **Initial count** can be any unsigned integer in the range defined by the **Counter size** parameter.

The **Counter size** parameter allows you to specify the range of integer values the block counts through. When the block counts through the entire counter range, the next time a trigger event occurs at the Inc/Dec port, the block resets the counter as follows:

- When you set the **Count direction** parameter to `Up` and the counter reaches the upper-limit of the counter range, the block restarts the counter at zero.
- When you set the **Count direction** parameter to `Down` and the counter reaches zero, the block restarts the counter at the upper-limit of the counter range.

You can set the **Counter size** parameter to one of the following options:

- `8 bits` — Specifies a counter with a range of `0` to `255`.
- `16 bits` — Specifies a counter with a range of `0` to `65535`.
- `32 bits` — Specifies a counter with a range of `0` to $2^{32}-1$.
- `User defined` — Enables the **Maximum count** parameter, which allows you to specify the upper-count limit as any arbitrary unsigned integer that the **Count data type** can represent. The counter values range from `0` to the value of the **Maximum count** parameter.
- `Specify via input port` — Enables the Max input port, which allows you to specify the upper-count limit as any arbitrary unsigned integer that the **Count data type** can represent. The counter values range from `0` to the value you specify as an input to the Max port.

Scalar Input Operation

When you set the **Count direction** parameter to `Up`, a trigger event at the Inc (increment) input port causes the block to increase the counter by one. Assuming no reset events occur, the block continues increasing the counter value when triggered, until the counter value reaches the upper-count limit. The next time a trigger event occurs at the Inc port, the block restarts the counter at `0` and resumes increasing the counter by one for each subsequent trigger event at the Inc port.

When you set the **Count direction** parameter to `Down`, a trigger event at the Dec (decrement) input port causes the block to decrease the counter by one. Assuming no reset events occur, the block continues decreasing the counter value when triggered until the counter value reaches zero. The next time a trigger event occurs at the Dec port, the block restarts the counter at the upper-count limit and resumes decreasing the counter by one for each subsequent trigger event at the Dec port.

Between triggering events, the block holds the output at its most recent value. The block resets the counter to its initial state when the trigger event specified by the **Count event** parameter occurs at the optional Rst input port. When the Inc/Dec and Rst ports receive trigger events simultaneously, the block first resets the counter, and then increments or decrements the counter appropriately. If you do not need to reset the counter during simulation, you can disable the Rst port by clearing the **Reset input** check box.

The **Output** parameter allows you to specify which values the block outputs.

- `Count` enables a Cnt output port on the block, which provides the current value of the counter as a scalar value. The Cnt output port has the same port sample time as the Inc/Dec input port.
- `Hit` enables a Hit output port on the block. The Hit port produces zeros while the value of the counter does not equal any of the integers you specify for the **Hit values** parameter. You can specify an integer or a vector of integers for the **Hit values** parameter. When the counter value does equal one or more of the values you specify for the **Hit values** parameter, the block outputs a value of `1` at the Hit output port. The Hit output port has the same port sample time as the Inc/Dec input port.
- `Count and Hit` enables both the Cnt and Hit output ports.

Vector Input Operation

The block treats vector inputs to the Inc/Dec port as a frame. Vector operation is the same as scalar operation, except that the block increments or decrements the counter by the total number of trigger events contained in the Inc/Dec input vector. Thus, the counter may change multiple times during the processing of a single Inc/Dec input vector.

When the block has a Hit port, the block outputs a value of `1` if any of the **Hit values** match any of the counter values during the processing of the Inc/Dec input vector.

When a trigger event splits across two consecutive vectors, that event is counted in the vector that contains the conclusion of the event. When the Rst port receives a trigger event at the same time as the Inc/Dec port, the block first resets the counter. The block then increments or decrements the counter by the number of trigger events contained in the Inc/Dec input vector.

When the input to the Inc/Dec port is a length N vector, the port sample time of the Inc/Dec input port is equal to the frame period of the input, or N times the sample time of the input signal. The port sample time of the Cnt and Hit output ports equals that of the Inc/Dec input port.

Free-Running Operation

The block operates in free-running mode when you select `Free running` for the **Count event** parameter.

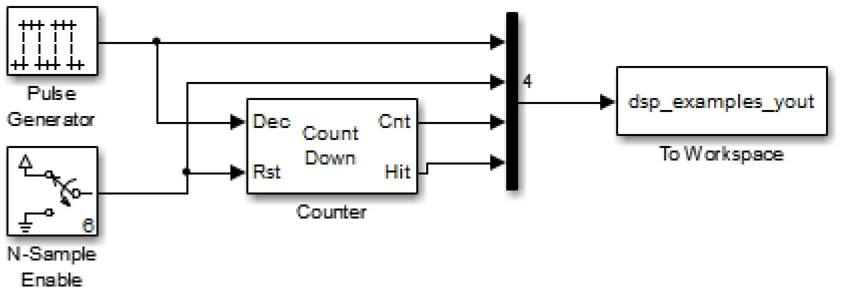
The Inc/Dec input port is disabled in this mode, and the block simply increments or decrements the counter at the constant interval, T_s, which you specify using the **Sample time** parameter.

In this mode, the Rst port always behaves as if the **Count event** parameter were set to `Non-zero sample`. Thus, the block triggers a reset event at each sample time that the Rst input is not zero.

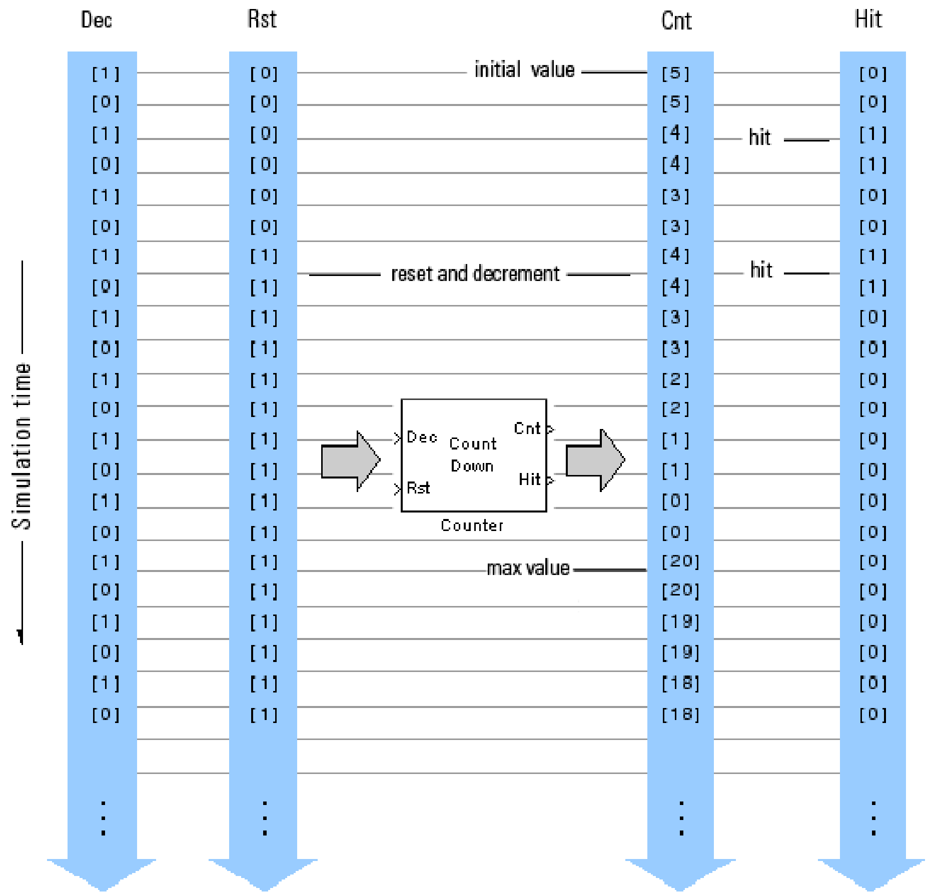
In this mode, the Cnt output is an M-by-`1` vector containing the count value at each of M consecutive sample times, where M is the value you specify for the **Samples per output frame** parameter. The Hit output is an M-by-`1` vector containing the hit status (`0` or `1`) at each of those M consecutive sample times. Both the Cnt and Hit output ports have a port sample time of $M*T_s$.

Examples

In the following model, ex_counter_ref, the Simulink Pulse Generator block drives the Dec port of the Counter block, and the `N-Sample Enable` block triggers the Rst port. All inputs to and outputs from the Counter block are multiplexed into a single To Workspace block using a 4-port Mux block.



The following figure shows the first 22 samples of the model's four-column output, `yout`.



You can see that the seventh input sample to both the Dec and Rst ports of the Counter block represent trigger events (rising edges). When this occurs, the block first resets the counter to its initial value of 5, and then immediately decrements the counter to 4. When the counter reaches its minimum value of 0, the block restarts the counter at its maximum value of 20 the next time a trigger event occurs at the Dec port.

Parameters

Count direction

Specify whether to count `Up` or `Down`. The port label on the block icon changes to Inc (increment) or Dec (decrement) based on the value of this parameter.

- When you set the **Count direction** parameter to `Up` and the counter reaches the upper-limit of the counter range, the block restarts the counter at zero the next time a trigger event occurs at the Inc port.
- When you set the **Count direction** parameter to `Down` and the counter reaches zero, the block restarts the counter at the upper-limit of the counter range the next time a trigger event occurs at the Dec port.

This parameter is tunable (Simulink) in Simulink Normal mode.

Count event

Specify the type of event that triggers the block to increment, decrement, or reset the counter when received at the Inc/Dec or Rst ports. When you set this parameter to `Free running`, the block disables the Inc/Dec port and counts at the constant interval specified by the **Sample time** parameter. For more information on all the possible settings of this parameter, see “Setting the Count Event Parameter” on page 2-378.

Counter size

Specify the range of integer values the block counts through. For more information about the valid values of this parameter, see “Setting the Counter Size and Initial Count Parameters” on page 2-379.

Maximum count

Specify the maximum value of the counter as any unsigned integer representable by the data type you specify for the **Counter data type** parameter. This parameter appears only when you set the **Counter size** parameter to `User defined`. Tunable (Simulink) in Simulink Normal mode.

Initial count

Specify the initial value of the counter. The block uses the initial value of the counter at the start of simulation and resets the counter back to that initial value each time a trigger event occurs at the Rst port. Tunable (Simulink).

Output

Select the output ports to enable. You can choose to enable the `Count`, `Hit`, or `Count and Hit` ports.

Hit values

Specify an integer or vector of integers whose occurrence in the count should be flagged by a `1` at the (optional) Hit output port. This parameter appears only when you set the **Output** parameter to `Hit` or `Count and Hit`. Tunable (Simulink).

Reset input

Select this check box to enable the Rst input port. When you enable the Rst port, the block resets the counter to its initial value each time a trigger event occurs at the Rst port. To specify the type of event that triggers a reset of the counter, set the **Count event** parameter. When you clear the **Reset input** check box, you cannot reset the counter during simulation.

Samples per output frame

Specify the number of samples, M, in each output vector. This parameter appears only when you set the **Count event** parameter to `Free running`.

Sample time

Specify the constant interval, T_s, at which the block increments or decrements the counter when in free-running mode. For example, to have the block increment the counter every `5` seconds, set the **Count direction** parameter to `Up`, the **Count event** parameter to `Free running`, and specify a value of `5` for the **Sample time** parameter. In free running mode, the sample time of the output ports is always $M*T_s$.

This parameter appears only when you set the **Count event** parameter to `Free running`.

Count data type

Specify the data type of the output at the Cnt port. This parameter appears only when you set the **Output** parameter to `Count` or `Count and Hit`.

Hit data type

Specify the data type of the output at the Hit port. This parameter appears only when you set the **Output** parameter to `Hit` or set it to `Count and Hit` with the **Count data type** parameter set to `Double`.

Supported Data Types

Port	Supported Data Types
Inc/Dec	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
	• Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Rst	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Max	• 8-, 16-, and 32-bit unsigned integers
Cnt	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Hit	• Logical • Boolean — The block might output Boolean values from the `Hit` output port depending on the setting of the **Hit data type** parameter.

See Also

Edge Detector	DSP System Toolbox
N-Sample Enable	DSP System Toolbox
N-Sample Switch	DSP System Toolbox

Introduced before R2006a

Covariance AR Estimator

Compute estimate of autoregressive (AR) model parameters using covariance method

Library

Estimation / Parametric Estimation

`dspparest3`

Description

The Covariance AR Estimator block uses the covariance method to fit an autoregressive (AR) model to the input data. This method minimizes the forward prediction error in the least squares sense.

The input must be a column vector or an unoriented vector, which is assumed to be the output of an AR system driven by white noise. This input represents a frame of consecutive time samples from a single-channel signal. The block computes the normalized estimate of the AR system parameters, $A(z)$, independently for each successive input frame.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

The order, p, of the all-pole model is specified by the **Estimation order** parameter. To guarantee a valid output, you must set the **Estimation order** parameter to be less than or equal to half the input vector length.

The top output, A, is a column vector of length p+1 with the same frame status as the input, and contains the normalized estimate of the AR model coefficients in descending powers of z.

```
[1 a(2) ... a(p+1)]
```

The scalar gain, G, is provided at the bottom output (`G`).

See the Burg AR Estimator block reference page for a comparison of the Burg AR Estimator, Covariance AR Estimator, Modified Covariance AR Estimator, and Yule-Walker AR Estimator blocks.

Parameters

Estimation order

The order of the AR model, p. To guarantee a nonsingular output, you must set p to be less than or equal to half the input length. Otherwise, the output might be singular.

References

Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
A	• Double-precision floating point • Single-precision floating point
G	• Double-precision floating point • Single-precision floating point

See Also

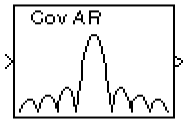
`Burg AR Estimator`	DSP System Toolbox

`Covariance Method`	DSP System Toolbox
`Modified Covariance AR Estimator`	DSP System Toolbox
`Yule-Walker AR Estimator`	DSP System Toolbox
`arcov`	Signal Processing Toolbox

Introduced before R2006a

Covariance Method

Power spectral density estimate using covariance method



Library

Estimation / Power Spectrum Estimation

`dspspect3`

Description

The Covariance Method block estimates the power spectral density (PSD) of the input using the covariance method. This method fits an autoregressive (AR) model to the signal by minimizing the forward prediction error in the least squares sense. The **Estimation order** parameter specifies the order of the all-pole model. The block computes the spectrum from the FFT of the estimated AR model parameters. To guarantee a valid output, the **Estimation order** parameter must be less than or equal to half the input vector length.

The input must be a column vector or an unoriented vector. It represents a frame of consecutive time samples from a single-channel signal. The block outputs a column vector containing the estimate of the power spectral density of the signal at N_{fft} equally spaced frequency points. The frequency points are in the range $[0,F_s)$, where F_s is the sampling frequency of the signal.

Selecting **Inherit FFT length from estimation order**, specifies that N_{fft} is one greater than the estimation order. Clearing the **Inherit FFT length from estimation order** check box allows you to use the **FFT length** parameter to specify N_{fft} as a power of 2. The block zero-pads or wraps the input to N_{fft} before computing the FFT.

When you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must hold:

- The input to the block is the original signal, with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

See the `Burg Method` block reference for a comparison of the Burg Method, Covariance Method, Modified Covariance Method, and Yule-Walker Method blocks.

Parameters

Estimation order

The order of the AR model. To guarantee a nonsingular output, the value of this parameter must be less than or equal to half the input length.

Inherit FFT length from estimation order

When selected, this option specifies that the FFT length is one greater than the estimation order.

FFT length

Enter the number of data points on which to perform the FFT, N_{fft}. When N_{fft} is larger than the input frame size, the block zero-pads each frame as needed. When N_{fft} is smaller than the input frame size, the block wraps each frame as needed. This parameter becomes visible only when you clear the **Inherit FFT length from estimation order** check box.

Inherit sample time from input

When you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must hold:

- The input to the block is the original signal, with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

Sample time of original time series

Specify the sample time of the original time-domain signal. This parameter becomes visible only when you clear the **Inherit sample time from input** check box.

References

Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L. Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

Orfanidis, S. J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

See Also

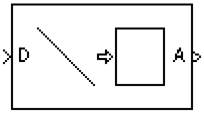
`Burg Method`	DSP System Toolbox
`Covariance AR Estimator`	DSP System Toolbox
`Modified Covariance Method`	DSP System Toolbox
`Short-Time FFT`	DSP System Toolbox
`Yule-Walker Method`	DSP System Toolbox

See “Spectral Analysis”for related information.

Introduced before R2006a

Create Diagonal Matrix

Create square diagonal matrix from diagonal elements



Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

Description

The Create Diagonal Matrix block populates the diagonal of the M-by-M matrix output with the elements contained in the length-M vector input, D. The elements off the diagonal are zero.

```
A = diag(D)     Equivalent MATLAB code
```

Supported Data Types

Port	Supported Data Types
D	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
A	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
	• Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

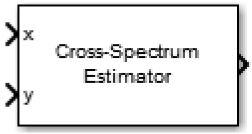
See Also

`Extract Diagonal`	DSP System Toolbox
`diag`	MATLAB

Introduced before R2006a

Cross-Spectrum Estimator

Estimate cross-power spectrum density



Library

Estimation / Power Spectrum Estimation

`dspspect3`

Description

The Cross Spectrum Estimator block outputs the frequency cross-power spectrum density of two real or complex input signals, *x* and *y*, via Welch's method of averaged modified periodograms. The input signals must be of the same size and data type.

The Cross Spectrum Estimator block computes the current power spectrum estimate by averaging the last *N* power spectrum estimates, where *N* is the number of spectral averages defined in **Number of spectral averages**. The block buffers the input data into overlapping segments. You can set the length of the data segment and the amount of data overlap through the parameters set in the block dialog box. The block computes the power spectrum based on the parameters set in the block dialog box.

Each column of the input signal is treated as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then it is interpreted as a single channel.

Algorithms

This block brings the capabilities of the dsp.CrossSpectrumEstimator System object to the Simulink environment.

For information on the algorithm used by the Cross Spectrum Estimator block, see the “Algorithms” on page 4-486 section of `dsp.CrossSpectrumEstimator`.

Parameters

Window length source

Source of the window length value. You can set this parameter to:

- `Same as input frame length` (default) — Window length is set to the frame size of the input.
- `Specify on dialog` — Window length is the value specified in **Window length**.

This parameter is nontunable.

Window length

Length of the window, in samples, used to compute the spectrum estimate, specified as a positive integer scalar greater than `2`. This parameter applies when you set **Window length source** to `Specify on dialog`. The default is `1024`. This parameter is nontunable.

Window Overlap (%)

Percentage of overlap between successive data windows, specified as a scalar in the range [`0, 100`). The default is `0`. This parameter is nontunable.

Number of spectral averages

Number of spectral averages, specified as a positive integer scalar. The cross-spectrum estimator computes the current power spectrum estimate by averaging the last *N* power spectrum estimates, where *N* is the number of spectral averages defined in **Number of spectral averages**. The default is `1`. This parameter is nontunable.

FFT length source

Source of the FFT length value. You can set this parameter to:

- `Auto` (default) — FFT length is set to the frame size of the input.
- `Property` — FFT length is the value specified in **FFT length**.

This parameter is nontunable.

FFT length

Length of the FFT used to compute the spectrum estimates, specified as a positive integer scalar. This parameter applies when you set **FFT length source** to `Property`. The default is `1024`. This parameter is nontunable.

Window function

Window function for the cross-spectrum estimator, specified as one of `Chebyshev | Flat Top | Hamming | Hann | Kaiser | Rectangular`. The default is `Hann`. This parameter is nontunable.

Sidelobe attenuation of window (dB)

Side lobe attenuation of the window, specified as real positive scalar. This parameter applies when you set **Window function** to `Chebyshev` or `Kaiser`. The default is `60`. This parameter is nontunable.

Frequency range

Frequency range of the cross-spectrum estimator. You can set this parameter to:

- `centered` (default) — The cross-spectrum estimator computes the centered two-sided spectrum of complex or real input signals, *x* and *y*. The length of the cross-spectrum estimate is equal to the FFT length. The spectrum estimate is computed over the frequency range `[-SampleRate/2 SampleRate/2]` when the FFT length is even and `[-SampleRate/2 SampleRate/2]` when FFT length is odd.
- `onesided` — The cross-spectrum estimator computes the one-sided spectrum of real input signals, *x* and *y*. When the FFT length, *NFFT* is even, length of the cross-spectrum estimate is (*NFFT*/ `2`) + `1`, and is computed over the frequency range `[0 SampleRate/2]`. When the FFT length, *NFFT* is odd, length of the cross-spectrum estimate is (*NFFT* + 1)/ `2`, and is computed over the frequency range `[0 SampleRate/2]`.
- `twosided` — The cross-spectrum estimator computes the two-sided spectrum of complex or real input signals, *x* and *y*. The length of the cross-spectrum estimate is equal to the FFT length. The spectrum estimate is computed over the frequency range `[0 SampleRate]`, where `SampleRate` is the sample rate of the input signal.

This parameter is nontunable.

Inherit sample rate from input

When you select this check box, the block's sample rate is computed as *N*/*Ts*, where *N* is the frame size of the input signal, and *Ts* is the sample time of the input signal. When you clear this check box, the block sample rate is the value specified in **Sample rate (Hz)**. By default, this check box is selected.

Sample rate (Hz)

Sample rate of the input signal, specified as a positive scalar value. The default is `44100`. This parameter applies when you clear the **Inherit sample rate from input** check box. This parameter is nontunable.

Simulate using

Type of simulation to run. You can set this parameter to:

- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

References

[1] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996.

[2] Kay, Steven M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1999.

[3] Stoica, Petre, and Randolph L. Moses. *Spectral Analysis of Signals*. Englewood Cliffs, NJ: Prentice Hall, 2005.

[4] Welch, P. D. "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short Modified Periodograms". *IEEE Transactions on Audio and Electroacoustics*. Vol. 15, No. 2, June 1967, pp. 70–73.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- When the FFT length is not a power of two, the executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- When the FFT length is a power of two, you can generate standalone C and C++ code from this block.

See Also

See Also

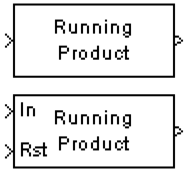
System Objects
dsp.CrossSpectrumEstimator

Blocks
`Discrete Transfer Function Estimator` | `Periodogram` | `Spectrum Analyzer`

Introduced in R2015a

Cumulative Product

Cumulative product of channel, column, or row elements



Library

Math Functions / Math Operations

`dspmathops`

Description

The Cumulative Product block computes the cumulative product along the specified dimension of the input or across time (running product).

The input can be a vector or matrix. The output always has the same dimensions, rate, data type, and complexity as the input.

Input and Output Characteristics

Valid Input

The Cumulative Product block accepts vector or matrix inputs containing real or complex values.

Valid Reset Signal

The optional reset port, `Rst`, accepts scalar values, which can be any built-in Simulink data type including `boolean`. The rate of the input to the Rst port must be the same or slower than that of the input data signal. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time.

Output Characteristics

The output always has the same dimensions, rate, data type, and complexity as the data signal input.

Computing the Running Product Along Channels of the Input

When you set the **Multiply input along** parameter to `Channels (running product)`, the block computes the cumulative product of the elements in each input channel. The running product of the current input takes into account the running product of all previous inputs. In this mode, you must also specify a value for the **Input processing** parameter. When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block computes the running product along each column of the current input. When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block computes a running product for each element of the input across time. See the following sections for more information:

-
-
-

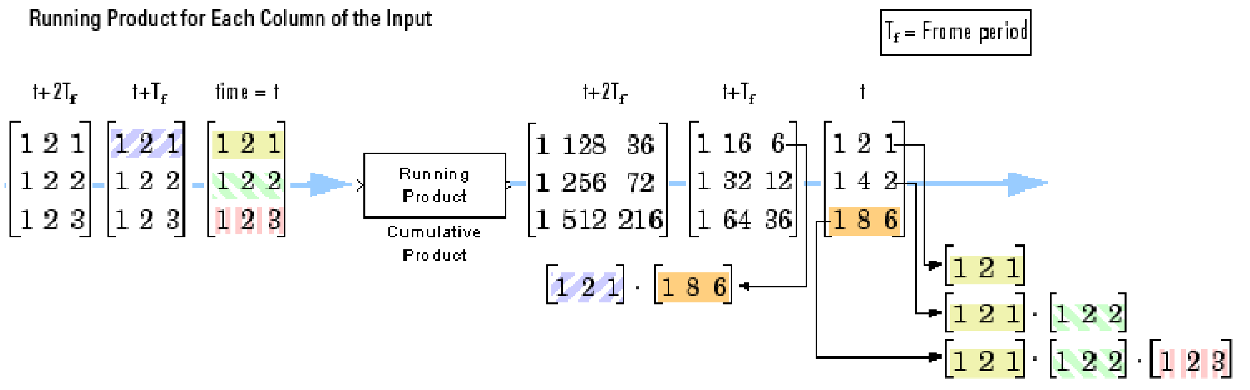
Computing the Running Product for Each Column of the Input

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each input column as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- The first row of the first output is the same as the first row of the first input.
- The first row of each subsequent output is the element-wise product of the first row of the current input (time t), and the last row of the previous output (time t - T_f, where T_f is the frame period).
- The output has the same size, dimension, data type, and complexity as the input.

Given an M-by-N matrix input, u, the output, y, is an M-by-N matrix whose first row has elements

$$y_{1,j}(t) = u_{1,j}(t) \cdot y_{M,j}\left(t - T_f\right)$$



Computing the Running Product for Each Element of the Input

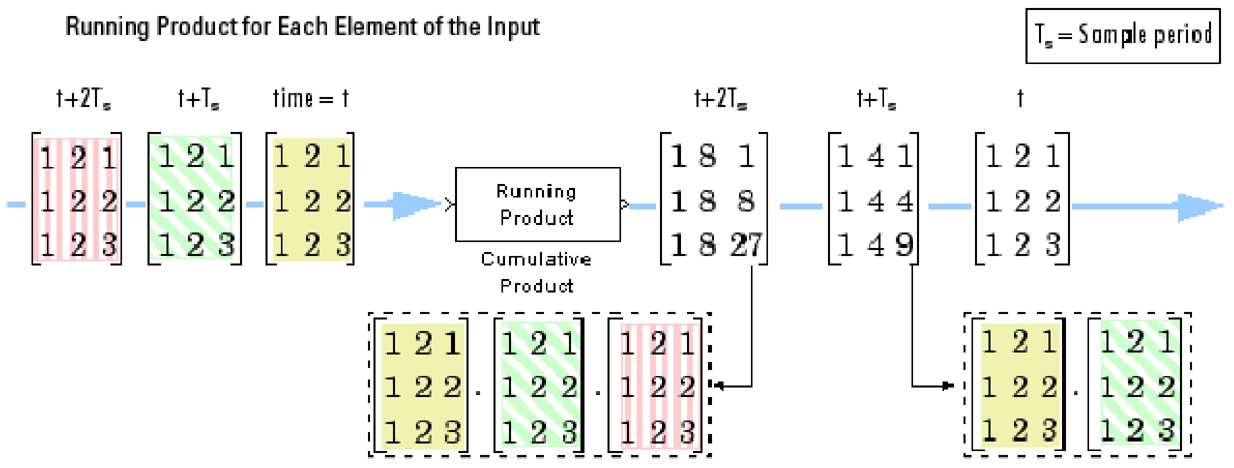
When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats each element of the input matrix as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- The first output is the same as the first input.
- Each subsequent output is the element-wise product of the current input (time t) and the previous output (time t - T_s, where T_s is the sample period).
- The output has the same size, dimension, data type, and complexity as the input.

Given an M-by-N matrix input, u, the output, y, is an M-by-N matrix with the elements

$$y_{i,j}(t) = u_{i,j}(t) \cdot y_{i,j}(t - T_s) \quad \begin{array}{l} 1 \le i \le M \\ 1 \le j \le N \end{array}$$

For convenience, the block treats length-M unoriented vector inputs as M-by-1 column vectors when multiplying along channels. In such cases, the output is a length-M unoriented vector.



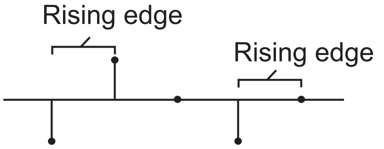
Resetting the Running Product

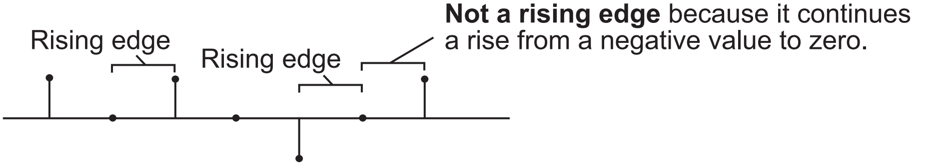
When you are computing the running product, you can configure the block to reset the running product whenever it detects a reset event at the optional `Rst` port. The rate of the input to the Rst port must be the same or slower than that of the input data signal. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. The input to the `Rst` port can be of the Boolean data type.

If a reset event occurs while the block is performing sample-based processing, the block initializes the current output to the values of the current input. If a reset event occurs while the block is performing frame-based processing, the block initializes the first row of the current output to the values in the first row of the current input.

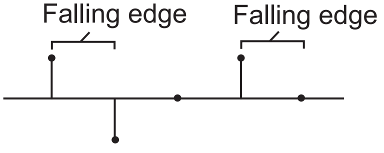
The **Reset port** parameter specifies the reset event, which can be one of the following:

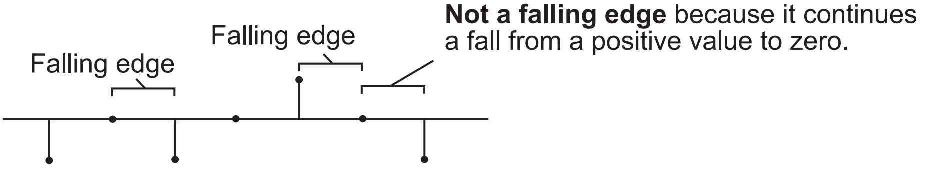
- `None` disables the `Rst` port.
- `Rising edge` — Triggers a reset operation when the `Rst` input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)





- `Falling edge` — Triggers a reset operation when the `Rst` input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)





- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above)
- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero

---

Note: When you run simulations in Simulink `MultiTasking` mode, reset signals have a one-sample latency. Thus, when the block detects a reset event, a one-sample delay occurs at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

Multiplying Along Columns

When you set the **Multiply input along** parameter to `Columns`, the block computes the cumulative product of each column of the input. In this mode, the current cumulative product is independent of the cumulative products of previous inputs.

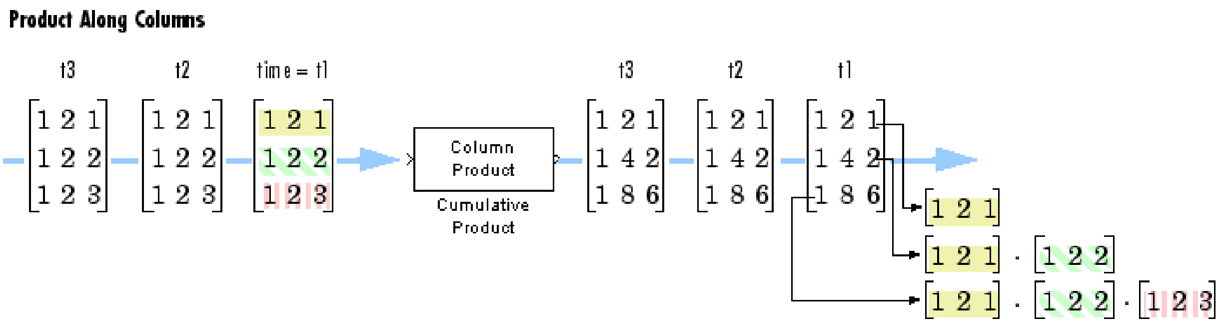
```
y = cumprod(u)      % Equivalent MATLAB code
```

The output has the same size, dimension, data type, and complexity as the input. The mth output row is the element-wise product of the first m input rows.

Given an M-by-N input, u, the output, y, is an M-by-N matrix whose jth column has elements

$$y_{i,j} = \prod_{k=1}^{i} u_{k,j} \qquad 1 \le i \le M$$

When multiplying along columns, the block treats length-M unoriented vector inputs as M-by-1 column vectors.



Multiplying Along Rows

When you set the **Multiply input along** parameter to `Rows`, the block computes the cumulative product of the row elements. In this mode, the current cumulative product is independent of the cumulative products of previous inputs.

```
y = cumprod(u,2)      % Equivalent MATLAB code
```

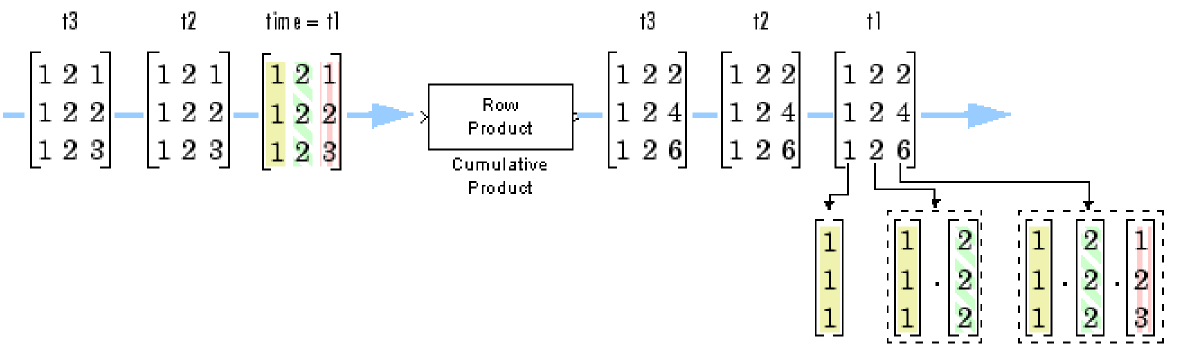
The output has the same size, dimension, and data type as the input. The nth output column is the element-wise product of the first n input columns.

Given an M-by-N matrix input, u, the output, y, is an M-by-N matrix whose ith row has elements

$$y_{i,j} = \prod_{k=1}^{j} u_{i,k} \qquad 1 \le j \le N$$

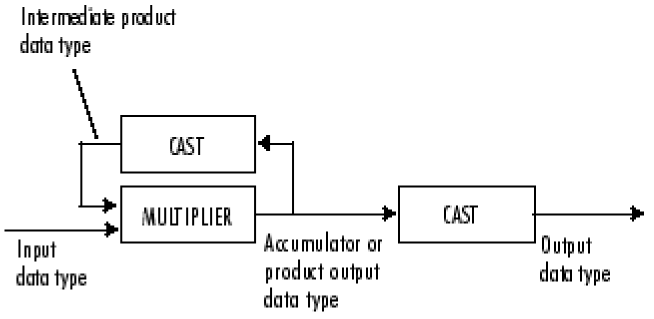
When you multiply along rows, the block treats length-N unoriented vector inputs as 1-by-N row vectors.

Product Along Rows



Fixed-Point Data Types

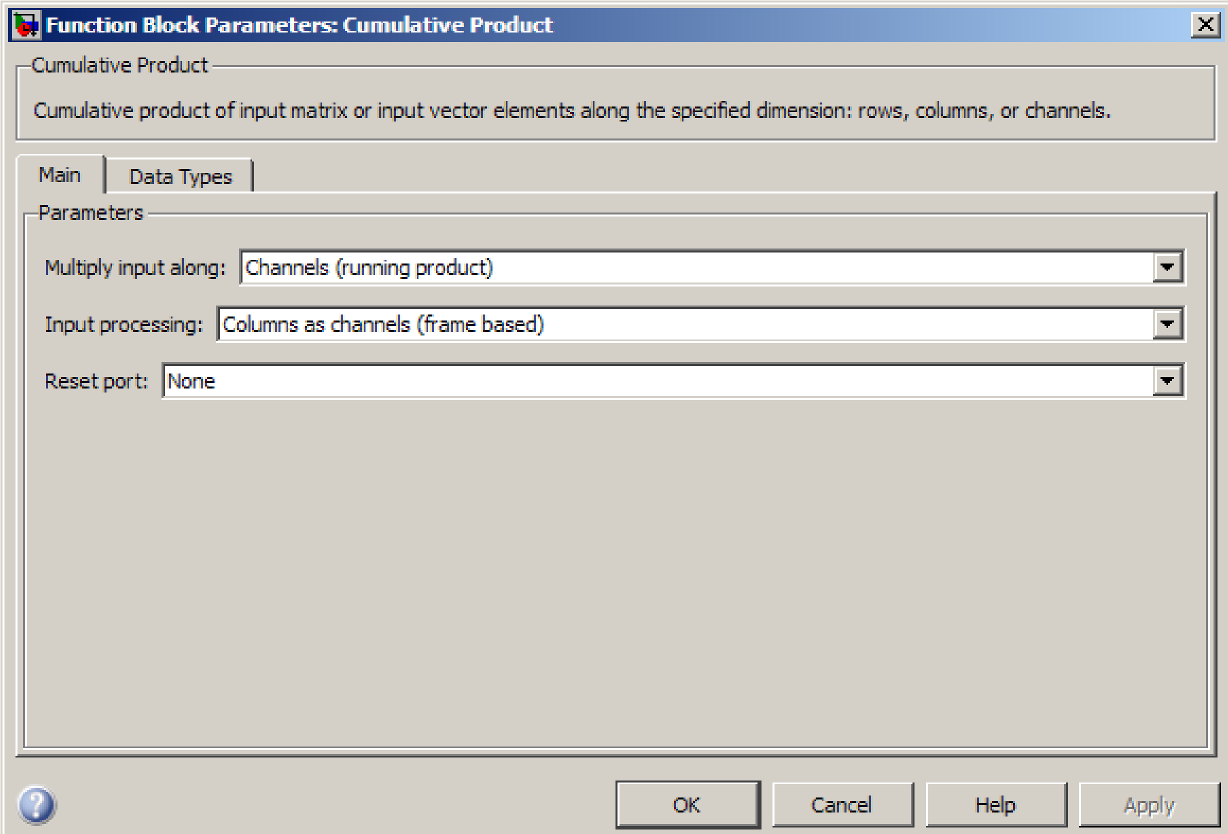
The following diagram shows the data types used within the Cumulative Product block for fixed-point signals.



The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types". You can set the accumulator, product output, intermediate product, and output data types in the block dialog as discussed in "Dialog Box" on page 2-408.

Dialog Box

The **Main** pane of the Cumulative Product block dialog appears as follows.



Multiply input along

Specify the dimension along which to compute the cumulative product. You can choose to multiply along `Channels (running product)`, `Columns`, or `Rows`. For more information, see the following sections:

- "Computing the Running Product Along Channels of the Input" on page 2-402
- "Multiplying Along Columns" on page 2-406
- "Multiplying Along Rows" on page 2-406

Input processing

Specify how the block should process the input when computing the running product along the channels of the input. You can set this parameter to one of the following options:

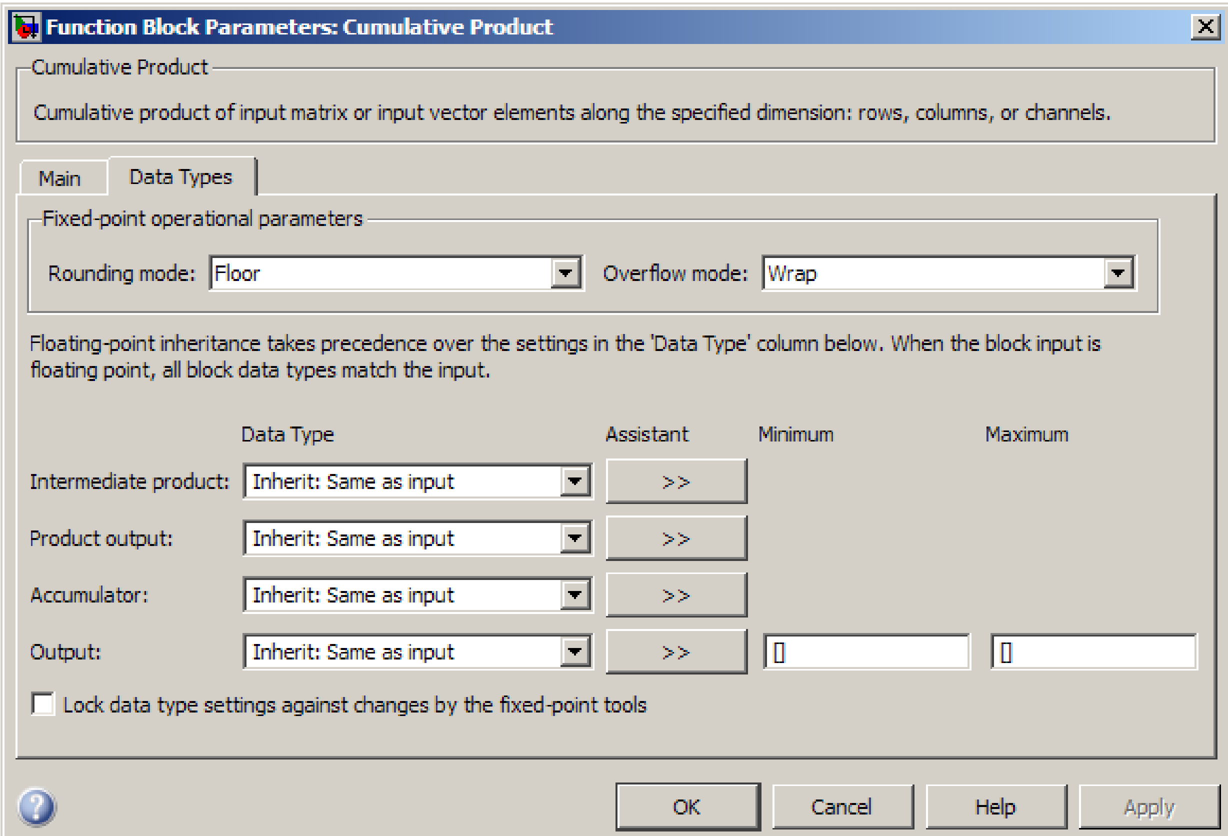
- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

This parameter is available only when you set the **Multiply input along** parameter to `Channels (running product)`.

Reset port

Determines the reset event that causes the block to reset the product along channels. The rate of the input to the Rst port must be the same or slower than that of the input data signal. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. This parameter appears only when you set the **Multiply input along** parameter to `Channels (running product)`. For more information, see .

The **Data Types** pane of the Cumulative Product block dialog appears as follows.



Note: Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Intermediate product

Specify the intermediate product data type. As shown in “Fixed-Point Data Types” on page 2-407, the output of the multiplier is cast to the intermediate product data type before the next element of the input is multiplied into it. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-407 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-407 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-407 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Input and Output Ports	Supported Data Types
Data input port, `In`	• Double-precision floating point • Single-precision floating point

Input and Output Ports	Supported Data Types
	• Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Reset input port, `Rst`	All built-in Simulink data types: • Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output port	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

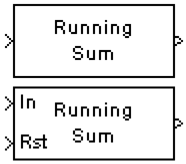
See Also

`Cumulative Sum`	DSP System Toolbox
`Matrix Product`	DSP System Toolbox
`cumprod`	MATLAB

Introduced before R2006a

Cumulative Sum

Cumulative sum of channel, column, or row elements



Library

Math Functions / Math Operations

`dspmathops`

Description

The Cumulative Sum block computes the cumulative sum along the specified dimension of the input or across time (running sum).

The inputs can be a vector or a matrix. The output always has the same dimensions, rate, data type, and complexity as the input.

Input and Output Characteristics

Valid Input

The Cumulative Sum block accepts vector or matrix inputs containing real or complex values.

Valid Reset Signal

The optional reset port, `Rst`, accepts scalar values, which can be any built-in Simulink data type including `boolean`. The rate of the input to the Rst port must be the same or slower than that of the input data signal. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time.

Output Characteristics

The output always has the same dimensions, rate, data type, and complexity as the data signal input.

Computing the Running Sum Along Channels of the Input

When you set the **Sum input along** parameter to `Channels (running sum)`, the block computes the cumulative sum of the elements in each input channel. The running sum of the current input takes into account the running sum of all previous inputs. In this mode, you must also specify a value for the **Input processing** parameter. When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block computes the running sum along each column of the current input. When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block computes a running sum for each element of the input across time. See the following sections for more information:

-
-
-

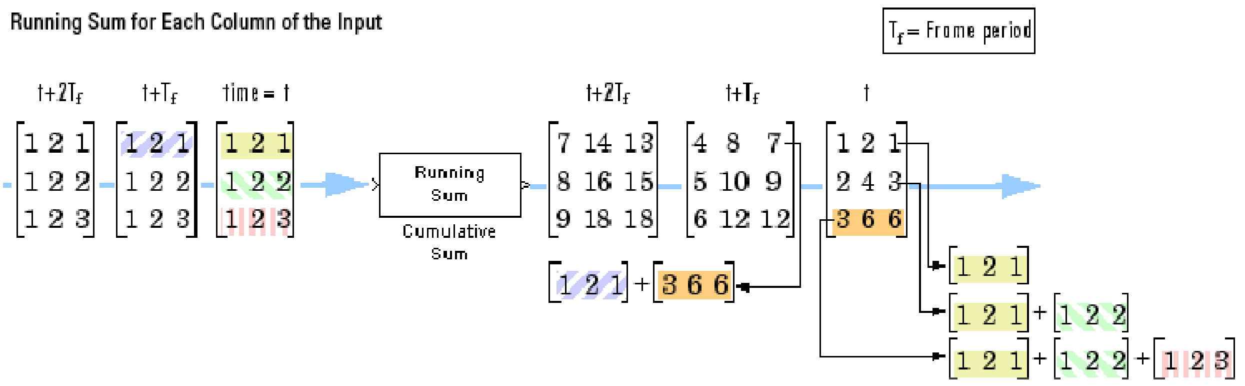
Computing the Running Sum for Each Column of the Input

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each input column as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- The first row of the first output is the same as the first row of the first input.
- The first row of each subsequent output is the sum of the first row of the current input (time t), and the last row of the previous output (time t - T_f, where T_f is the frame period).
- The output has the same size, dimension, data type, and complexity as the input.

Given an M-by-N matrix input, u, the output, y, is an M-by-N matrix whose first row has elements

$$y_{1,j}(t) = u_{1,j}(t) + y_{M,j}(t - T_f)$$



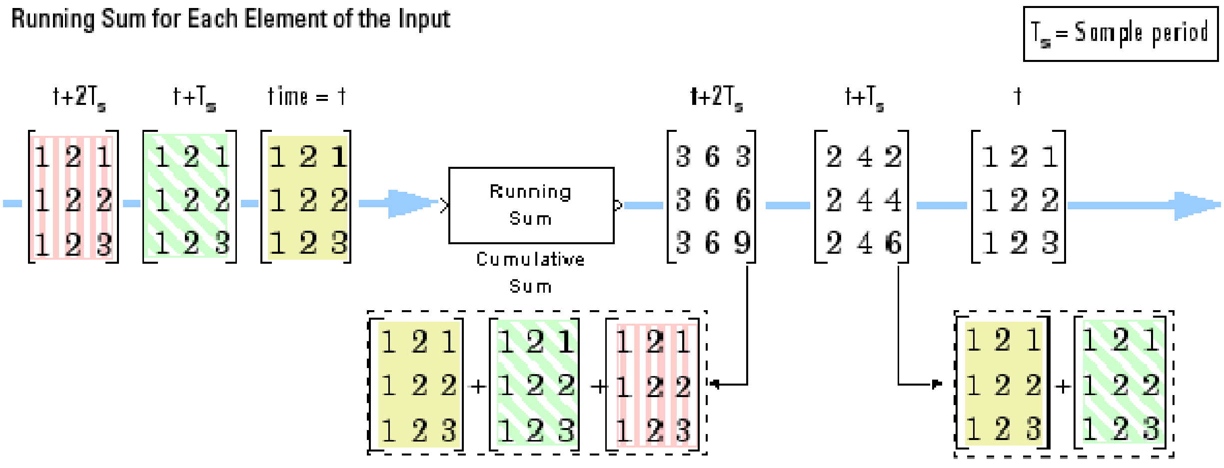
Computing the Running Sum for Each Element of the Input

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats each element of the input matrix as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- The first output is the same as the first input.
- Each subsequent output is the sum of the current input (time t) and the previous output (time t - T_s, where T_s is the sample period).
- The output has the same size, dimension, data type, and complexity as the input.

Given an M-by-N matrix input, u, the output, y, is an M-by-N matrix with the elements

$$y_{i,j}(t) = u_{i,j}(t) + y_{i,j}(t - T_s) \qquad \begin{array}{l} 1 \le i \le M \\ 1 \le j \le N \end{array}$$



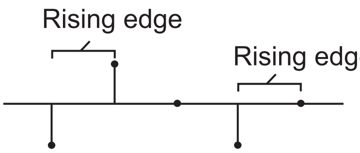
Resetting the Running Sum

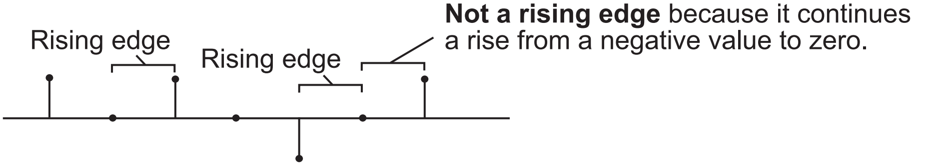
When you are computing the running sum, you can configure the block to reset the running sum whenever it detects a reset event at the optional `Rst` port. The rate of the input to the Rst port must be the same or slower than that of the input data signal. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. The reset sample time must be a positive integer multiple of the input sample time. The input to the `Rst` port can be of the `boolean` data type.

If a reset event occurs while the block is performing sample-based processing, the block initializes the current output to the values of the current input. If a reset event occurs while the block is performing frame-based processing, the block initializes the first row of the current output to the values in the first row of the current input.

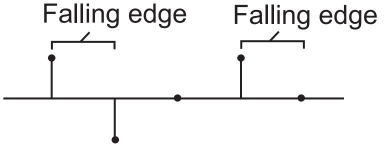
The **Reset port** parameter specifies the reset event, which can be one of the following:

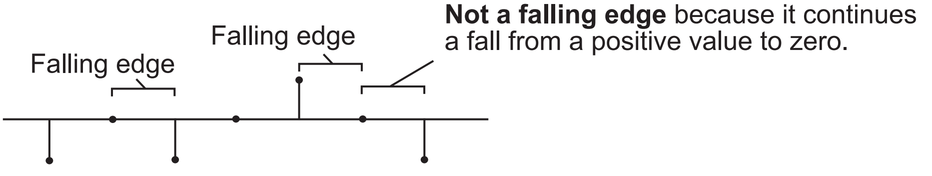
- `None` disables the `Rst` port.
- `Rising edge` — Triggers a reset operation when the `Rst` input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)





- `Falling edge` — Triggers a reset operation when the `Rst` input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)





- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above)
- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero

---

Note: When you run simulations in the Simulink `MultiTasking` mode, reset signals have a one-sample latency. Thus, when the block detects a reset event, a one-sample delay occurs at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

Summing Along Columns

When you set the **Sum input along** parameter to `Columns`, the block computes the cumulative sum of each column of the input. In this mode, the current cumulative sum is independent of the cumulative sums of previous inputs.

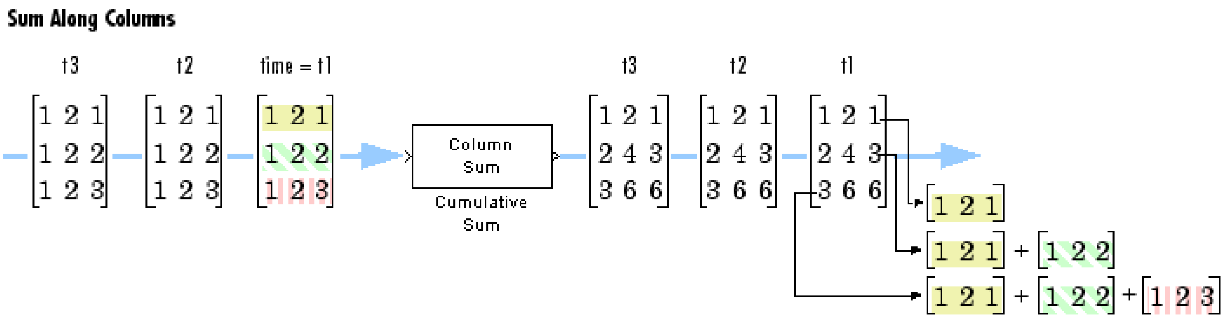
```
y = cumsum(u)      % Equivalent MATLAB code
```

The output has the same size, dimension, data type, and complexity as the input. The mth output row is the sum of the first m input rows.

Given an M-by-N input, u, the output, y, is an M-by-N matrix whose jth column has elements

$$y_{i,j} = \sum_{k=1}^{j} u_{k,j} \qquad 1 \le i \le M$$

The block treats length-M unoriented vector inputs as M-by-1 column vectors when summing along columns.



Summing Along Rows

When you set the **Sum input along** parameter to `Rows`, the block computes the cumulative sum of the row elements. In this mode, the current cumulative sum is independent of the cumulative sums of previous inputs.

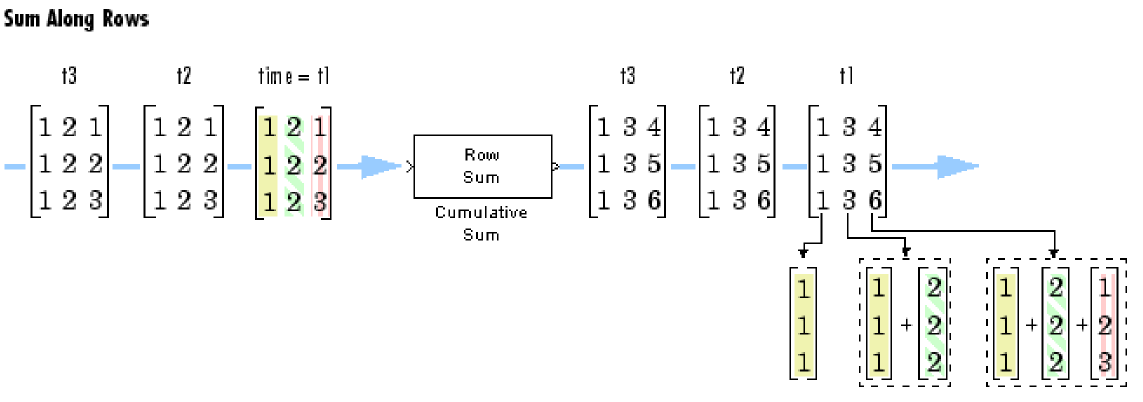
```
y = cumsum(u,2)      % Equivalent MATLAB code
```

The output has the same size, dimension, and data type as the input. The *n*th output column is the sum of the first *n* input columns.

Given an *M*-by-*N* input, *u*, the output, *y*, is an *M*-by-*N* matrix whose *i*th row has elements

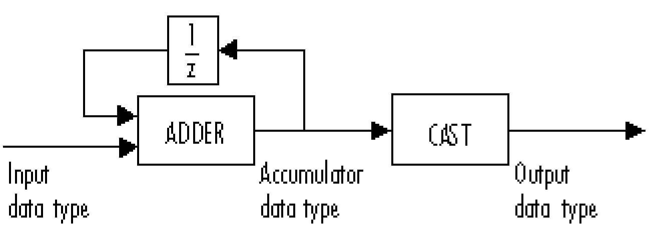
$$y_{i,j} = \sum_{k=1}^{j} u_{i,k} \qquad 1 \le j \le N$$

When you sum along rows, the block treats length-*N* unoriented vector inputs as 1-by-*N* row vectors.



Fixed-Point Data Types

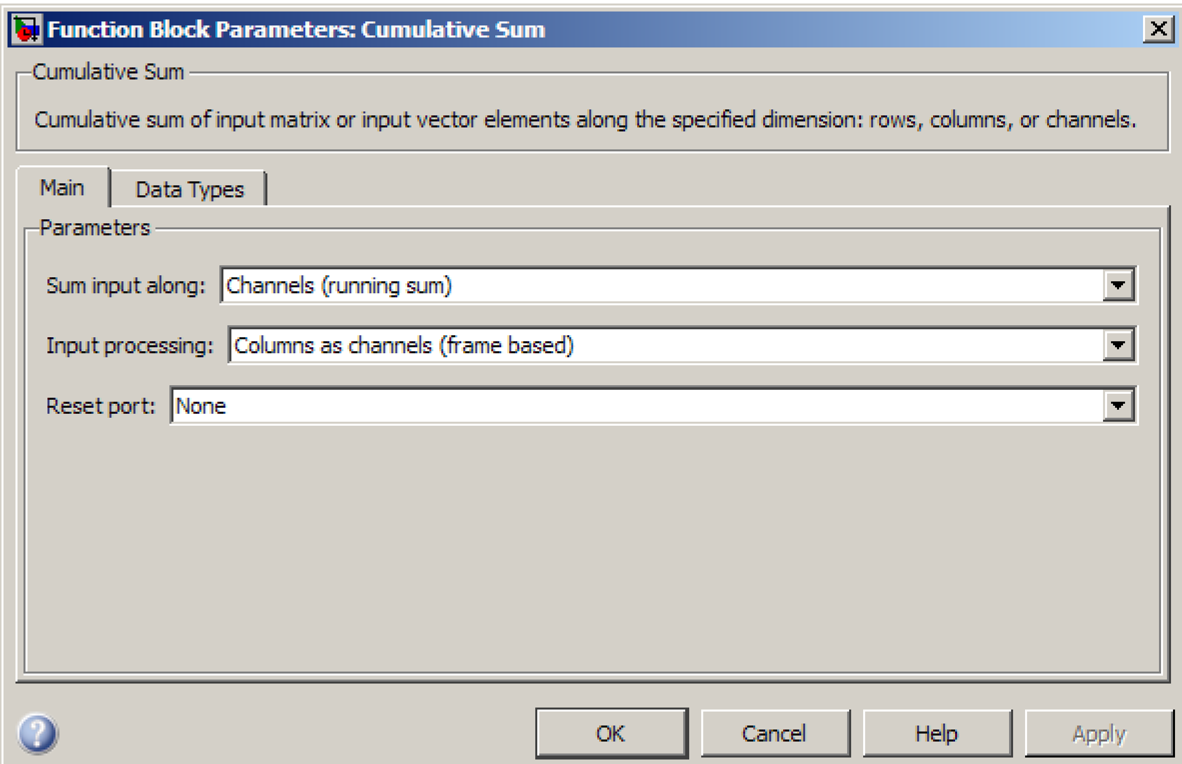
The following diagram shows the data types used within the Cumulative Sum block for fixed-point signals.



You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 2-422.

Dialog Box

The **Main** pane of the Cumulative Sum block dialog appears as follows.



Sum input along

Specify the dimension along which to compute the cumulative summations. You can choose to sum along `Channels (running sum)`, `Columns`, or `Rows`. For more information, see the following sections:

- "Computing the Running Sum Along Channels of the Input" on page 2-416

- "Summing Along Columns" on page 2-420
- "Summing Along Rows" on page 2-420

Input processing

Specify how the block should process the input when computing the running sum along the channels of the input. You can set this parameter to one of the following options:

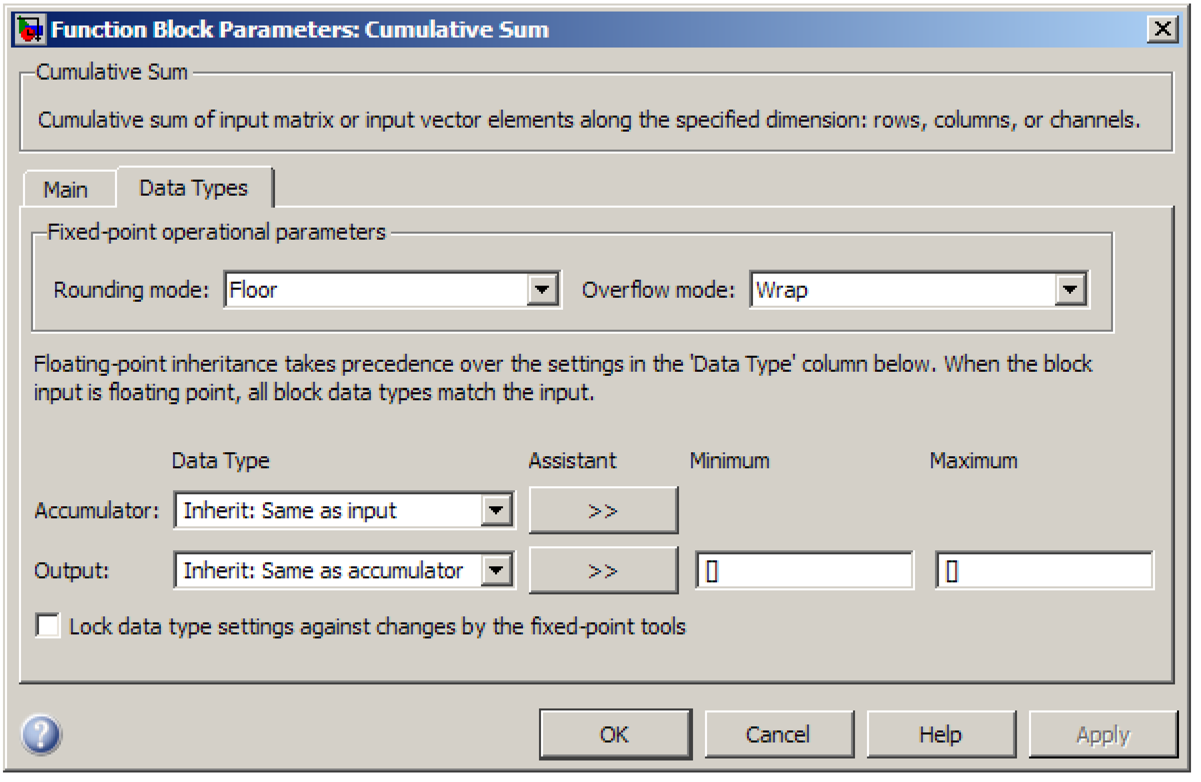
- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

This parameter is available only when you set the **Sum input along** parameter to `Channels (running sum)`.

Reset port

Determines the reset event that causes the block to reset the sum along channels. The rate of the input to the Rst port must be the same or slower than that of the input data signal. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. This parameter appears only when you set the **Sum input along** parameter to `Channels (running sum)`. For more information, see .

The **Data Types** pane of the Cumulative Sum block dialog appears as follows.



Note: Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-421 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button `>>` to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-421 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button `>>` to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Input and Output Ports	Supported Data Types
Data input port, `In`	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Reset input port, `Rst`	All built-in Simulink data types: • Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output port	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

`Cumulative Product`	DSP System Toolbox
`Difference`	DSP System Toolbox
`Matrix Sum`	DSP System Toolbox
`cumsum`	MATLAB

Introduced before R2006a

Data Type Conversion

Convert input signal to specified data type

Library

Signal Management / Signal Attributes

`dspsigattribs`

Description

The Data Type Conversion block is an implementation of the Simulink Data Type Conversion block. See Data Type Conversion for more information.

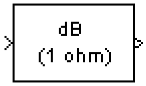
HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Data Type Conversion.

Introduced in R2008a

dB Conversion

Convert magnitude data to decibels (dB or dBm)



Library

Math Functions / Math Operations

`dspmathops`

Description

The dB Conversion block converts a linearly scaled power or amplitude input to dB or dBm. The reference power is 1 Watt for conversions to dB and 1 mWatt for conversions to dBm. The **Input signal** parameter specifies whether the input is a power signal or a voltage signal, and the **Convert to** parameter controls the scaling of the output. When selected, the **Add eps to input to protect against "log(0) = -inf"** parameter adds a value of `eps` to all power and voltage inputs. When this option is not enabled, zero-valued inputs produce `-inf` at the output.

The output is the same size as the input.

Power Inputs

Select `Power` as the **Input signal** parameter when the input, `u`, is a real, nonnegative, power signal (units of watts). When the **Convert to** parameter is set to `dB`, the block performs the dB conversion

```
y = 10*log10(u)       % Equivalent MATLAB code
```

When the **Convert to** parameter is set to `dBm`, the block performs the dBm conversion

```
y = 10*log10(u) + 30
```

The dBm conversion is equivalent to performing the dB operation *after* converting the input to milliwatts.

Voltage Inputs

Select `Amplitude` as the **Input signal** parameter when the input, `u`, is a real voltage signal (units of volts). The block uses the scale factor specified in ohms by the **Load resistance** parameter, `R`, to convert the voltage input to units of power (watts) before converting to dB or dBm.

When the **Convert to** parameter is set to `dB`, the block performs the dB conversion

```
y = 10*log10(abs(u)^2/R)
```

When the **Convert to** parameter is set to `dBm`, the block performs the dBm conversion

```
y = 10*log10(abs(u)^2/R) + 30
```

The dBm conversion is equivalent to performing the dB operation *after* converting the `(abs(u)^2/R)` result to milliwatts.

Parameters

Convert to

The logarithmic scaling to which the input is converted, `dB` or `dBm`. The reference power is 1 W for conversions to dB and 1 mW for conversions to dBm. Tunable (Simulink).

Input signal

The type of input signal, `Power` or `Amplitude`.

Load resistance

The scale factor used to convert voltage inputs to units of power. Tunable (Simulink).

Add eps to input to protect against "log(0) = -inf"

When selected, adds `eps` to all input values (power or voltage). Tunable (Simulink).

Supported Data Types

- Double-precision floating point
- Single-precision floating point

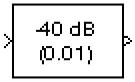
See Also

dB Gain	DSP System Toolbox
Math Function	Simulink
`log10`	MATLAB

Introduced before R2006a

dB Gain

Apply decibel gain



Library

Math Functions / Math Operations

`dspmathops`

Description

The dB Gain block multiplies the input by the decibel values specified in the **Gain** parameter. For an M-by-N input matrix u with elements u_{ij}, the **Gain** parameter can be a real M-by-N matrix with elements g_{ij} to be multiplied element-wise with the input, or a real scalar.

$$y_{ij} = u_{ij} 10^{(g_{ij}/k)}$$

The value of k is 10 for power signals (select `Power` as the **Input signal** parameter) and 20 for voltage signals (select `Amplitude` as the **Input signal** parameter).

The value of the equivalent linear gain

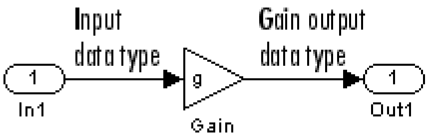
$$g_{ij}^{lin} = 10^{(g_{ij}/k)}$$

is displayed in the block icon below the dB gain value. The output is the same size as the input.

The dB Gain block supports real and complex floating-point and fixed-point data types.

Fixed-Point Data Types

The following diagram shows the data types used within the dB Gain subsystem block for fixed-point signals.



The settings for the fixed-point parameters of the Gain block in the diagram above are as follows:

- Integer rounding mode: `Floor`
- Saturate on integer overflow — unselected
- Parameter data type mode — `Inherit via internal rule`
- Output data type mode — `Inherit via internal rule`

See the Gain reference page for more information.

Parameters

Gain

The dB gain to apply to the input, a scalar or a real *M*-by-*N* matrix. Tunable (Simulink).

Input signal

The type of input signal: `Power` or `Amplitude`. Tunable (Simulink).

Note: This block does not support tunability in generated code.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)

- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

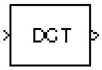
See Also

dB Conversion	DSP System Toolbox
Math Function	Simulink
`log10`	MATLAB

Introduced before R2006a

DCT

Discrete cosine transform (DCT) of input



Library

Transforms

`dspxfrm3`

Description

The DCT block computes the unitary discrete cosine transform (DCT) of each channel in the M-by-N input matrix, u.

```
y = dct(u)     % Equivalent MATLAB code
```

For all N-D input arrays, the block computes the DCT across the first dimension. The size of the first dimension (frame size), must be a power of two. To work with other frame sizes, use the `Pad` block to pad or truncate the frame size to a power-of-two length.

When the input to the DCT block is an M-by-N matrix, the block treats each input column as an independent channel containing M consecutive samples. The block outputs an M-by-N matrix whose lth column contains the length-M DCT of the corresponding input column.

$$y(k,l) = w(k) \sum_{m=1}^{M} u(m,l) \cos \frac{\pi(2m-1)(k-1)}{2M}, \quad k = 1,\ldots,M$$

where

$$w(k) = \begin{cases} \dfrac{1}{\sqrt{M}}, & k = 1 \\ \sqrt{\dfrac{2}{M}}, & 2 \le k \le M \end{cases}$$

The **Sine and cosine computation** parameter determines how the block computes the necessary sine and cosine values. This parameter has two settings, each with its advantages and disadvantages, as described in the following table.

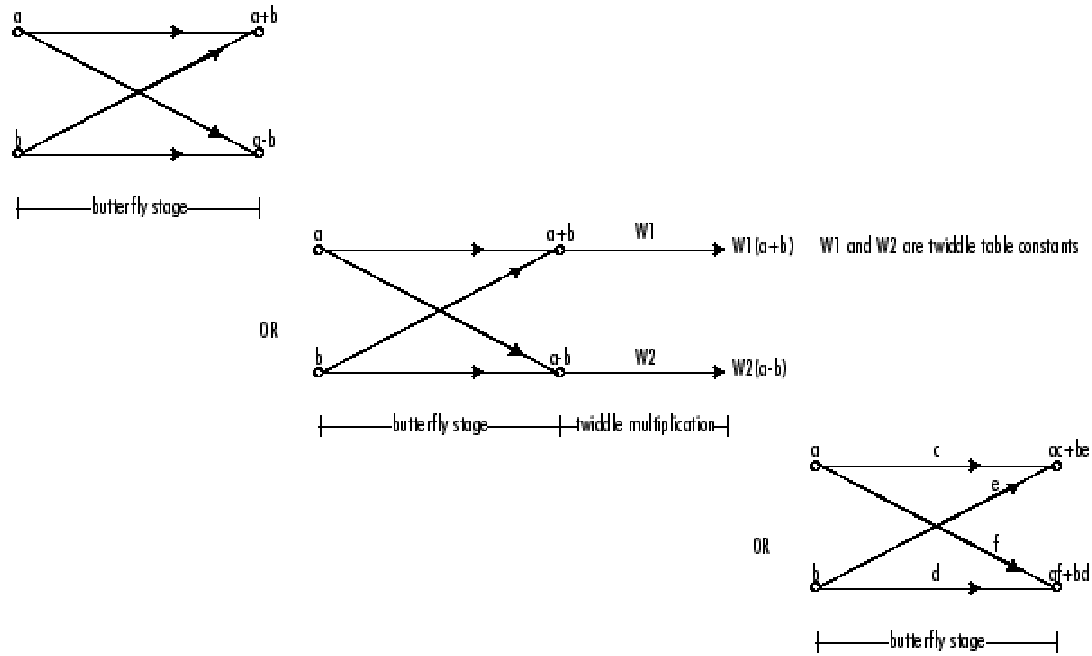
Sine and Cosine Computation Parameter Setting	Sine and Cosine Computation Method	Effect on Block Performance
`Table lookup`	The block computes and stores the trigonometric values before the simulation starts, and retrieves them during the simulation. When you generate code from the block, the processor running the generated code stores the trigonometric values computed by the block in a speed-optimized table, and retrieves the values during code execution.	The block usually runs much more quickly, but requires extra memory for storing the precomputed trigonometric values.
`Trigonometric fcn`	The block computes sine and cosine values during the simulation. When you generate code from the block, the processor running the generated code computes the sine and cosine values while the code runs.	The block usually runs more slowly, but does not need extra data memory. For code generation, the block requires a support library to emulate the trigonometric functions, increasing the size of the generated code.

This block supports Simulink virtual buses.

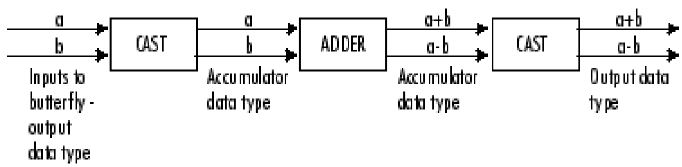
Fixed-Point Data Types

The following diagrams show the data types used within the DCT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the DCT block dialog as discussed in “Dialog Box” on page 2-437.

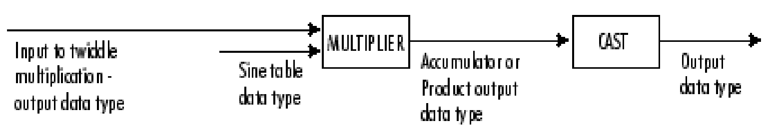
Inputs to the DCT block are first cast to the output data type and stored in the output buffer. Each butterfly stage processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type.



Butterfly Stage Data Types



Twiddle Multiplication Data Types

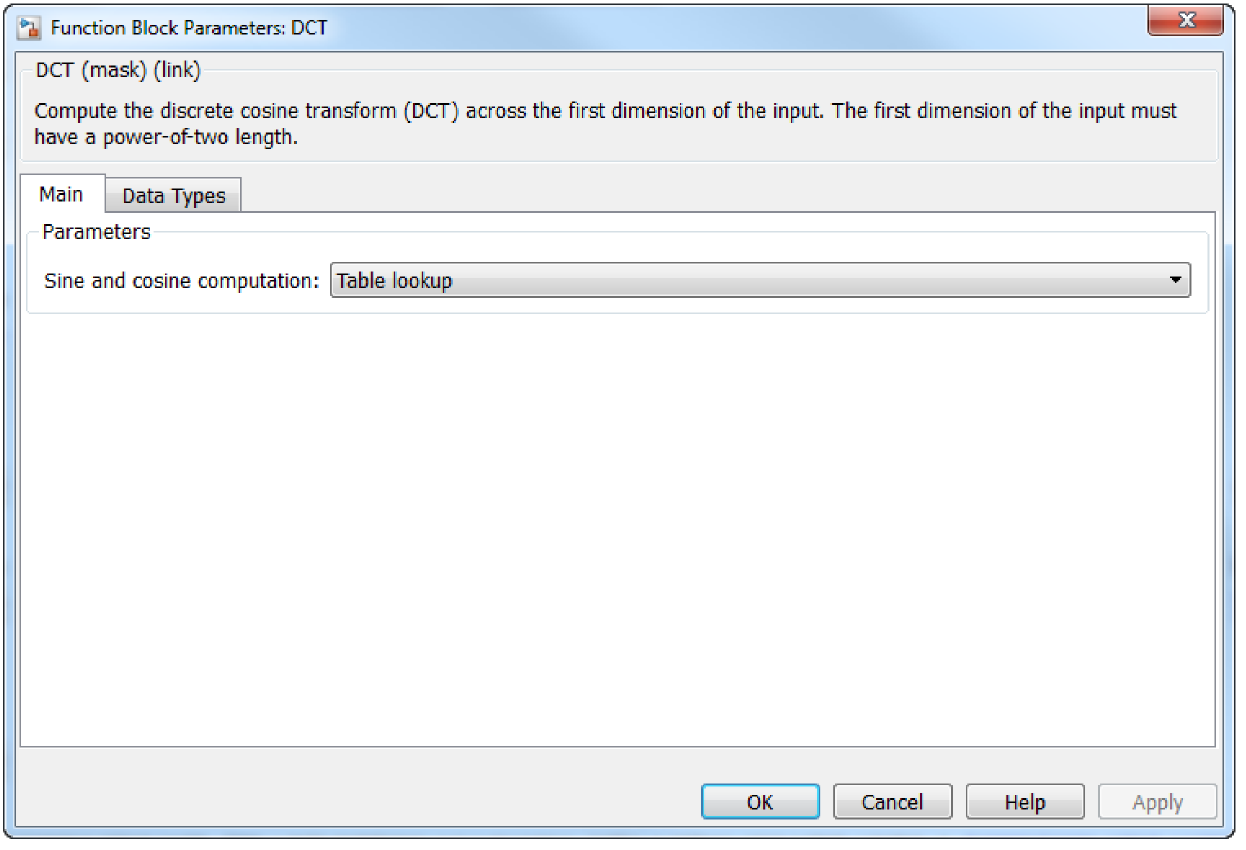


The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

Note: When the block input is fixed point, all internal data types are signed fixed point.

Dialog Box

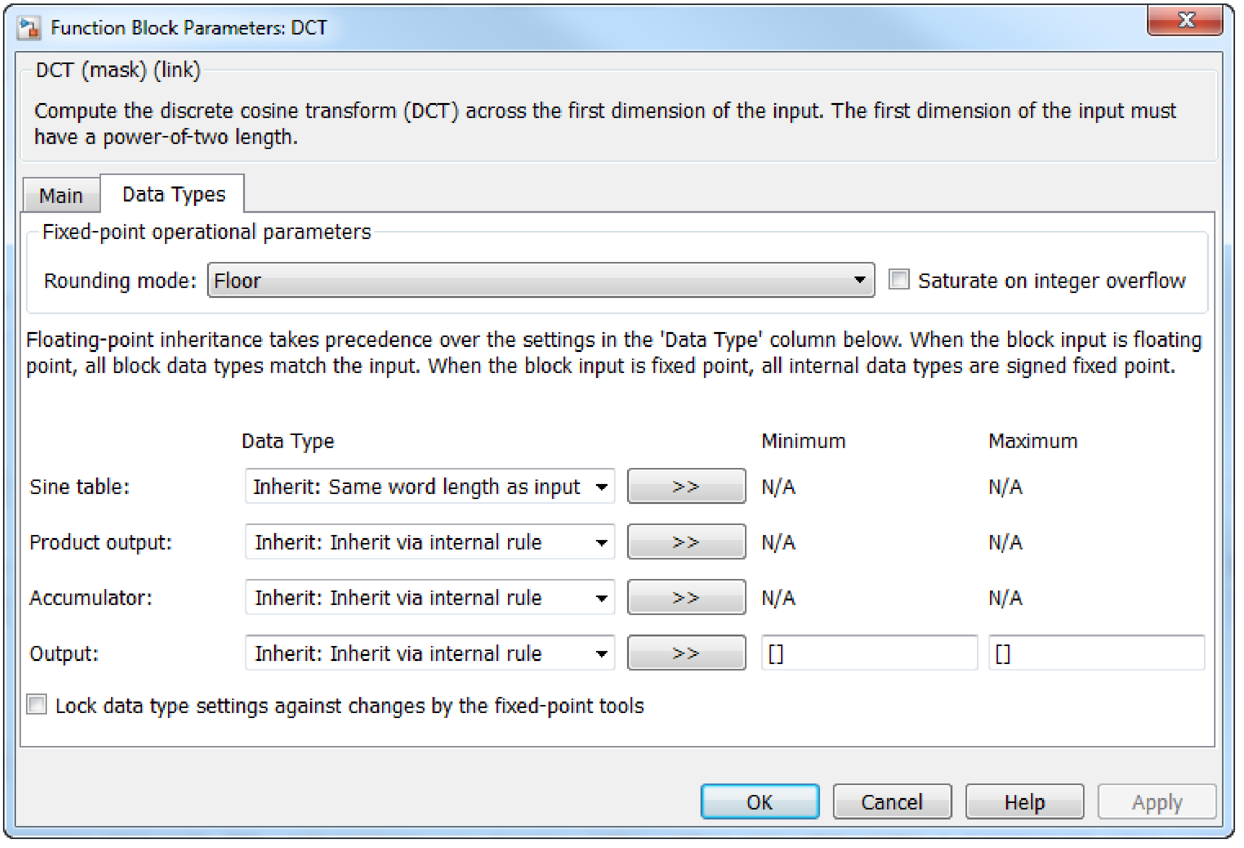
The **Main** pane of the DCT block dialog appears as follows.



Sine and cosine computation

Sets the block to compute sines and cosines by either looking up sine and cosine values in a speed-optimized table (`Table lookup`), or by making sine and cosine function calls (`Trigonometric fcn`). See the table in the "Description" on page 2-434 section.

The **Data Types** pane of the DCT block dialog appears as follows.



Rounding mode

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameter; they always round to `Nearest`.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations. The sine table values do not obey this parameter; instead, they are always saturated.

Sine table data type

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

The sine table values do not obey the **Rounding mode** and **Saturate on integer overflow** parameters; instead, they are always saturated and rounded to `Nearest`.

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-435 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-435 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-435 for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`.

  When you select `Inherit: Inherit via internal rule`, the block calculates the output word length and fraction length automatically. The internal rule first calculates an ideal output word length and fraction length using the following equations:

$$WL_{ideal\ output} = WL_{input} + floor(\log_2(DCT\ length - 1)) + 1$$

$$FL_{ideal\ output} = FL_{input}$$

  Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

See Also

See Also

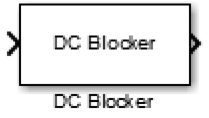
Functions
`dct`

Blocks
`Complex Cepstrum` | `FFT` | `IDCT` | `Real Cepstrum`

Introduced before R2006a

DC Blocker

Block DC component



Library

Signal Operations

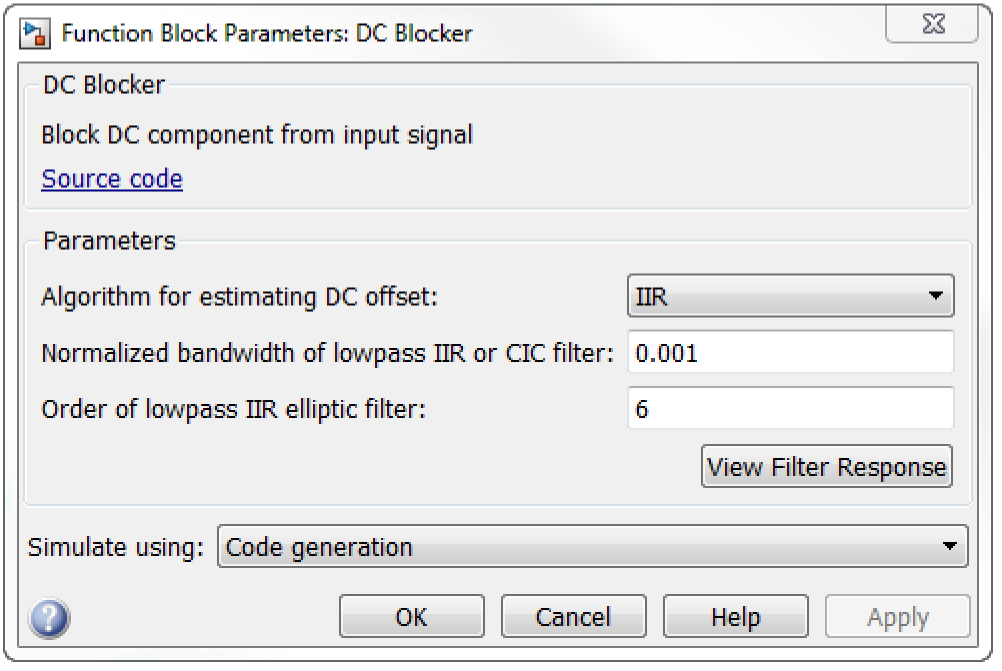
`dspsigops`

Description

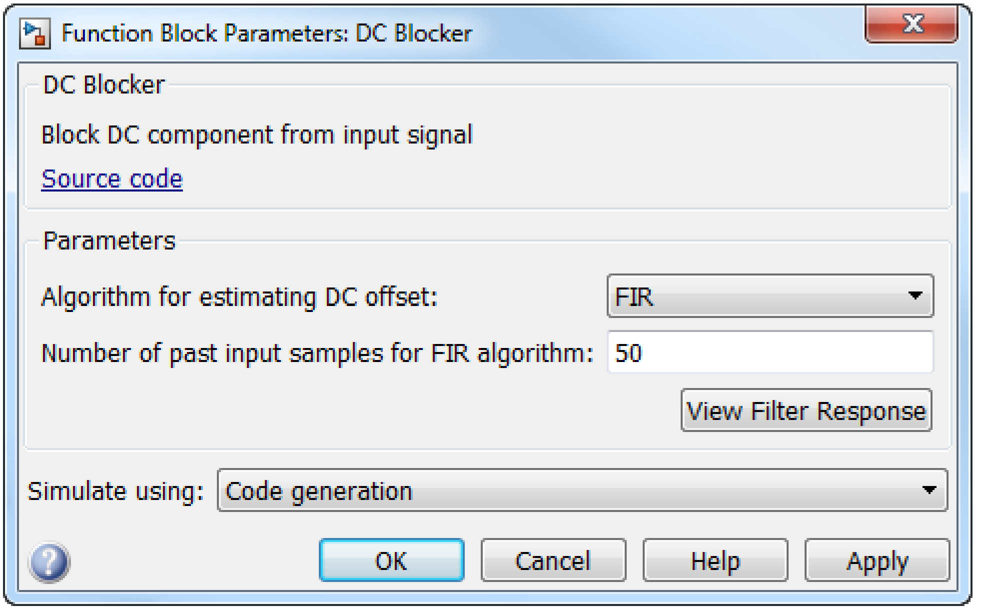
The DC Blocker block removes the DC component of the input signal.

Dialog Box

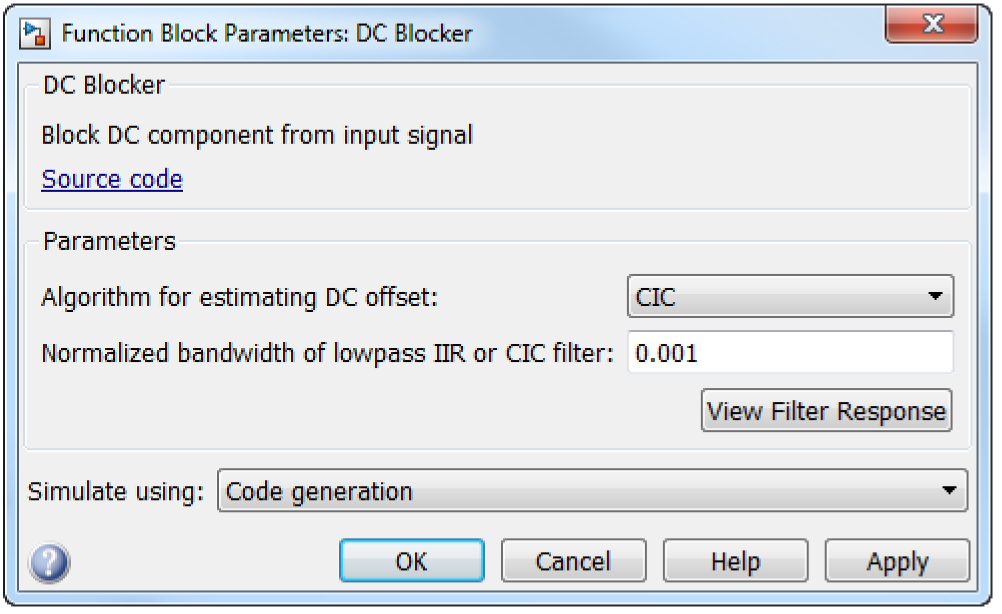
The DC Blocker dialog box changes based on how the DC offset is estimated. The dialog box for the `IIR` method is shown below.



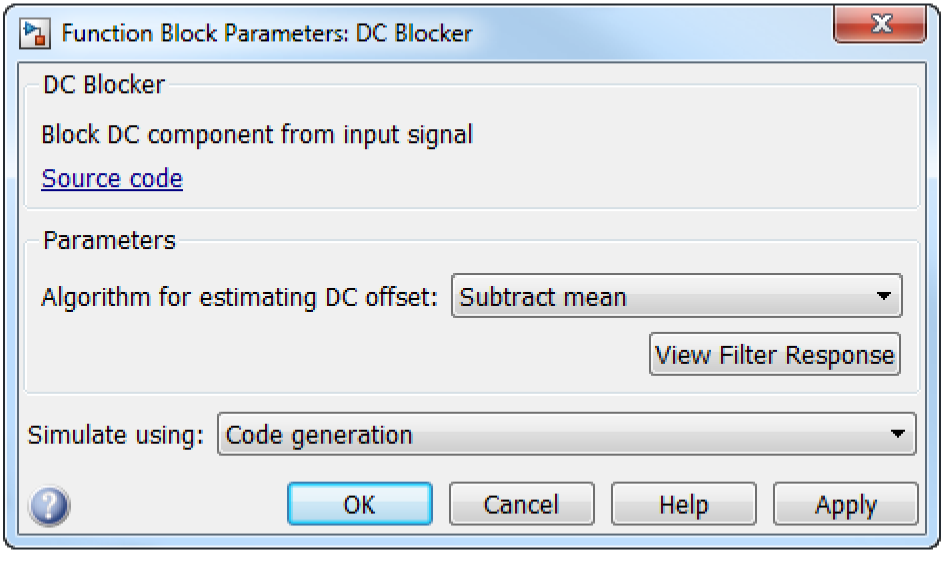
The dialog box for the `FIR` method is shown below.



The dialog box for the `CIC` method is shown below.



The dialog box for the `Subtract mean` method is shown below.



Algorithm for estimating DC offset

Specify the algorithm used for estimating the DC offset. Select from the following:

- `IIR` uses a recursive estimate based on a narrow, lowpass elliptic filter. This algorithm typically uses less memory than FIR and is more efficient.
- `FIR` uses a nonrecursive, moving-average estimate. This algorithm typically uses more memory than IIR and is less efficient.
- `CIC` uses a lowpass filter that does not employ any multipliers. If the algorithm is `CIC`, then fixed-point data must be input to the DC Blocker.
- `Subtract mean` computes the means of the columns of the input matrix and subtracts the means from the input. This method does not retain state between inputs. For example, if the input is `[1 2 3 4; 3 4 5 6]`, then the DC Blocker block in `Subtract mean` mode outputs `[-1 -1 -1 -1; 1 1 1 1]`.

Normalized bandwidth of lowpass IIR or CIC filter

Specify the normalized filter bandwidth as a real scalar greater than 0 and less than 1. The DC Blocker uses this parameter only when the estimation algorithm is set to `IIR` or `CIC`.

Order of lowpass IIR elliptic filter

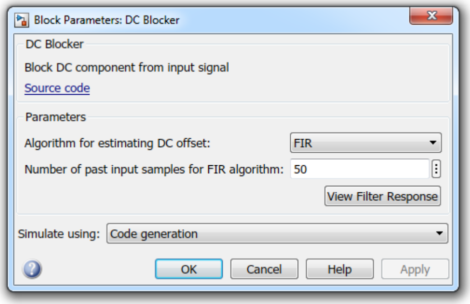
Specify the filter order as an integer greater than 3. The DC Blocker uses this parameter only when the estimation algorithm is set to `IIR`.

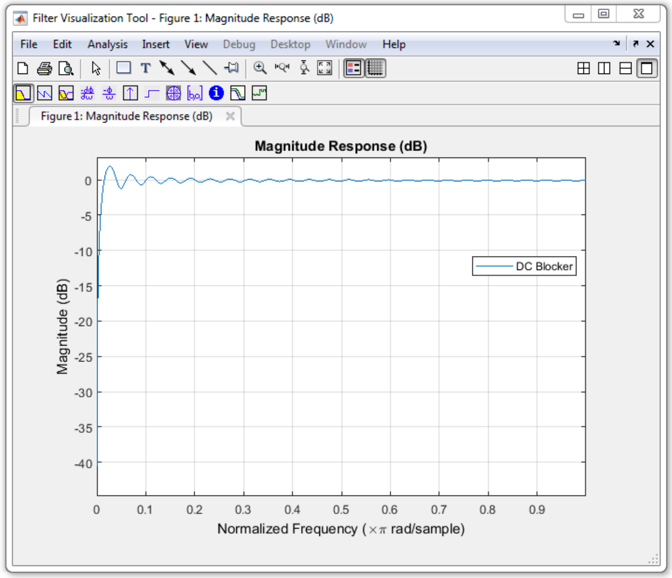
Number of past input samples for FIR algorithm

Specify, as a positive integer, the number of samples to use when the estimation algorithm is set to `FIR`.

View Filter Response

Opens the `fvtool` and displays the magnitude response of the DC Blocker. The response is based on the block parameters. Changes made to these parameters update `fvtool`.





To update the magnitude response while `fvtool` is running, modify the block parameters and click **Apply**.

Simulate using

Select the simulation type from the following:

- `Code generation` (default)
- `Interpreted execution`

Examples

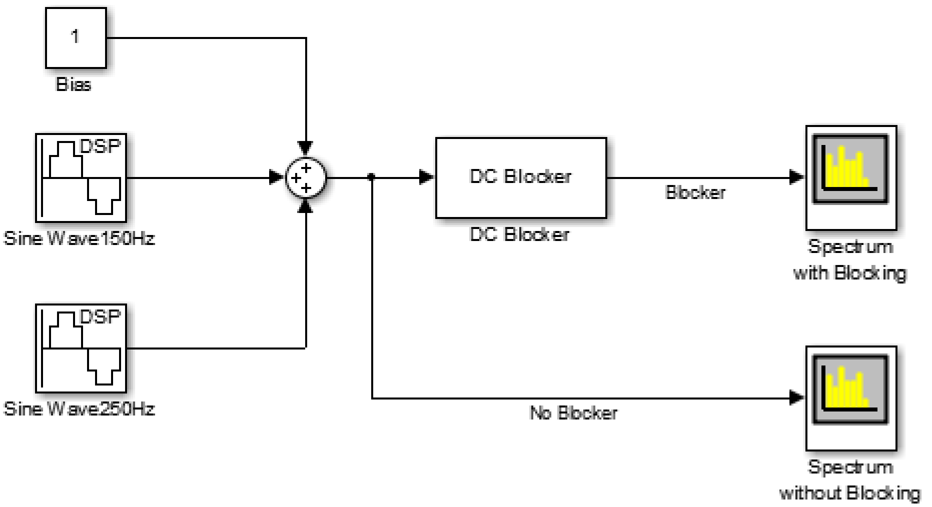
Use DC Blocker to Remove DC Component of Signal

This example shows how to use the DC Blocker to remove the DC component of a signal.

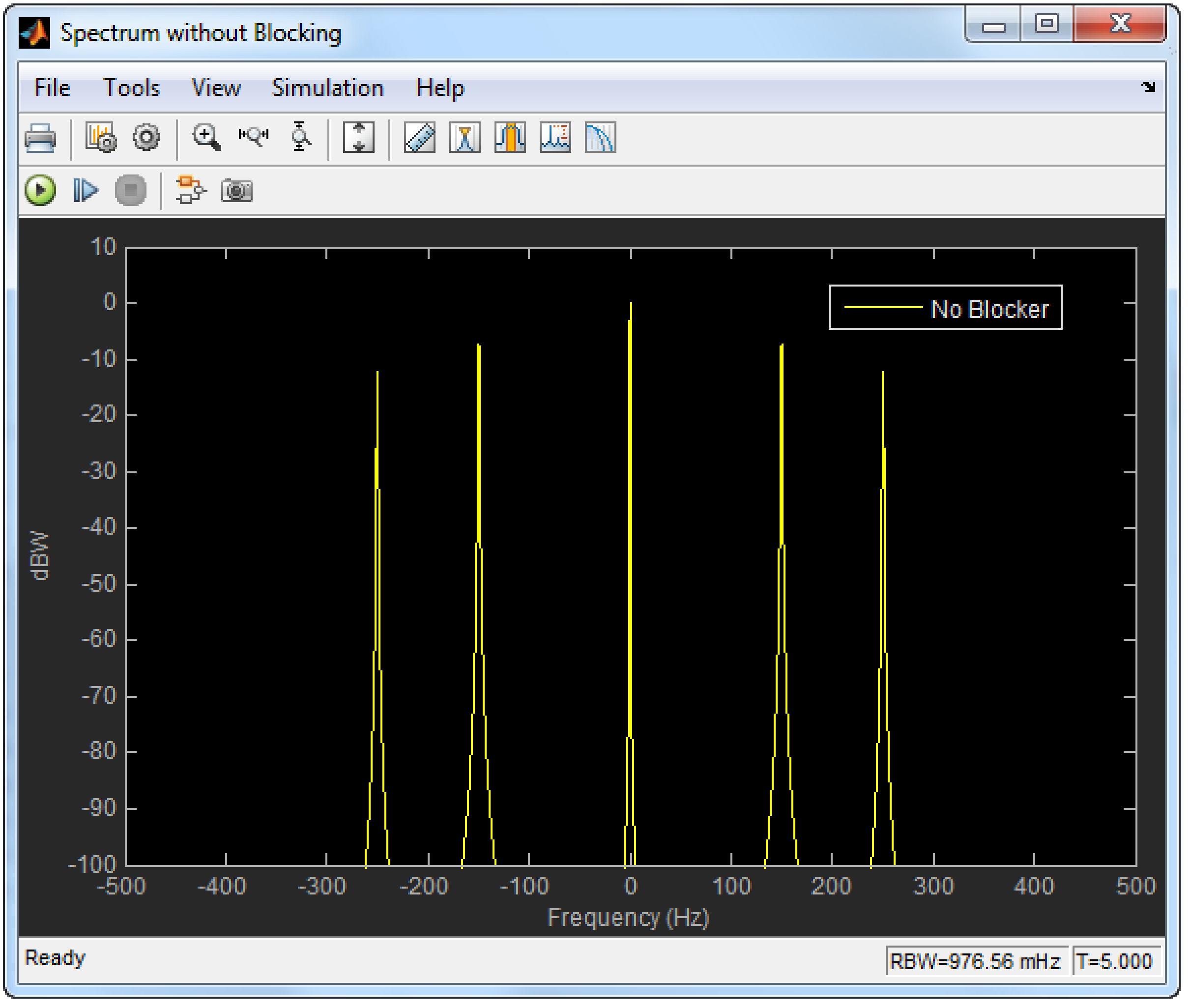
Load the DC Blocker example by typing `ex_dc_blocker` in the MATLAB command prompt.

The spectral output from the DC Blocker is displayed in Spectrum with Blocking, while the spectrum of the input signal is displayed in Spectrum without Blocking.

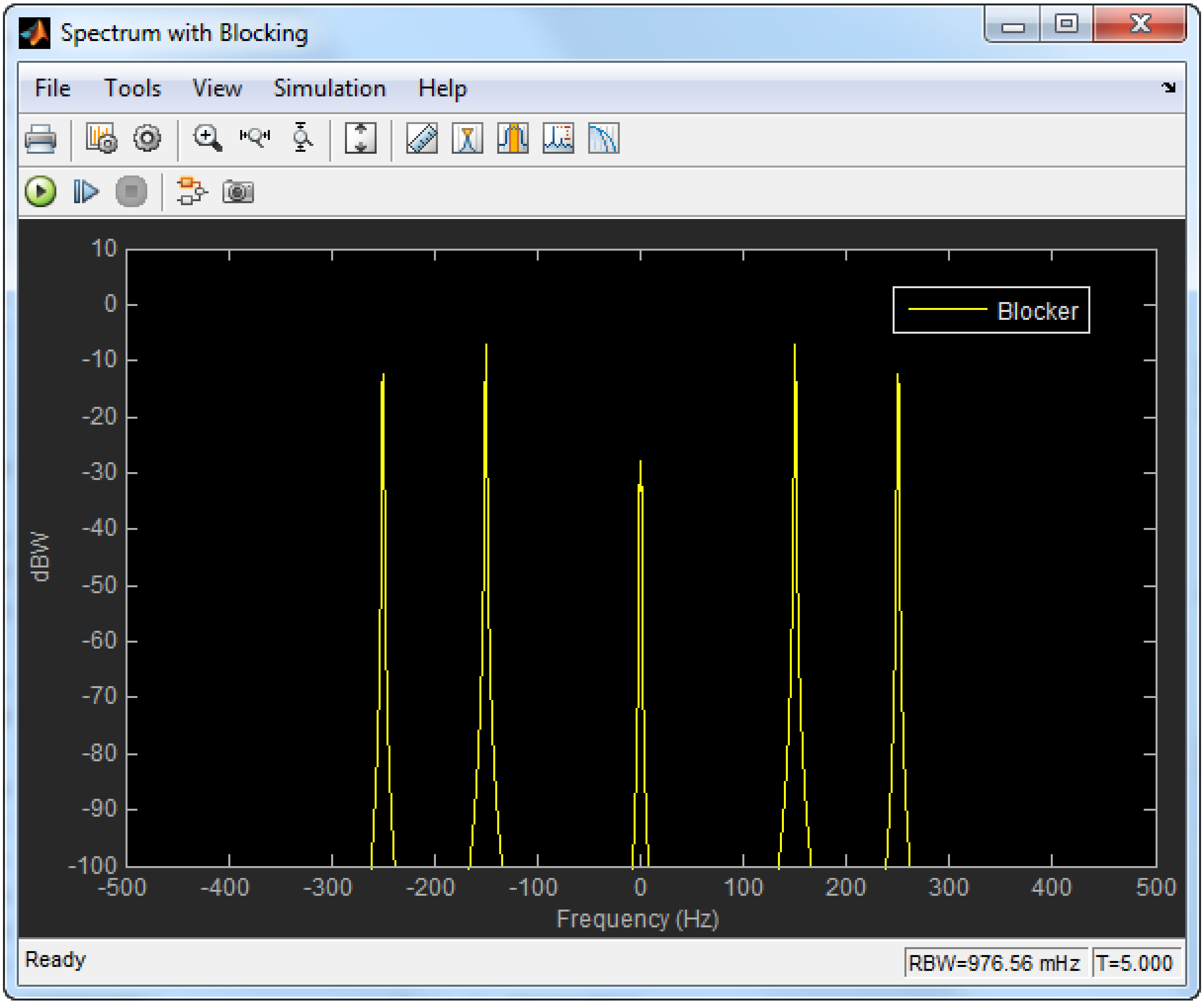
The two sine wave sources are set to use 1000 samples per frame because the `Subtract mean` estimation algorithm requires a statistically significant number of samples to calculate a valid mean.



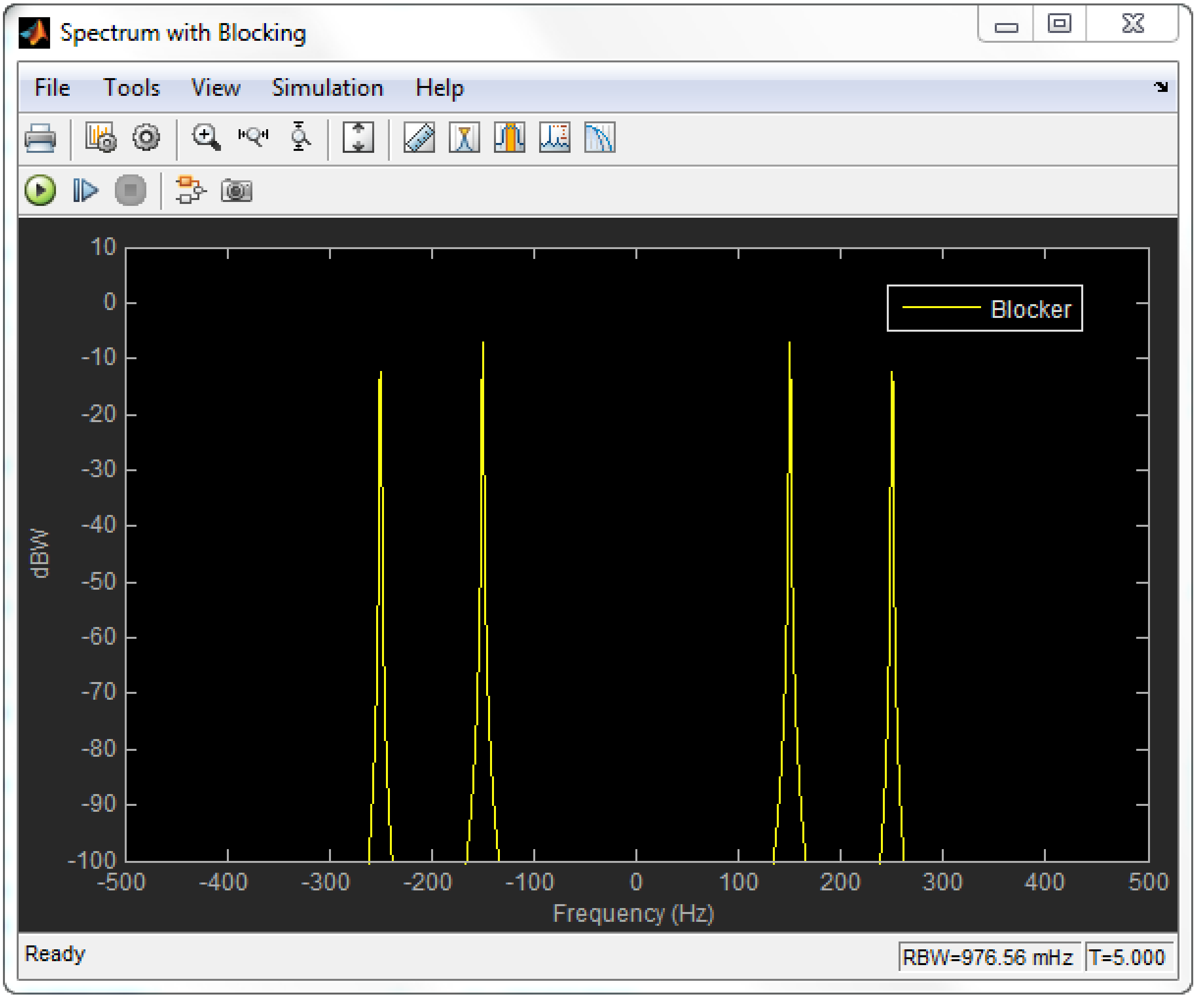
In the model, run the simulation. The spectrum of the input signal shows tones at 150 Hz and 250 Hz and a significant (0 dBW) DC component.



Using the default `IIR` setting for the DC Blocker estimation algorithm, the tones at 150 Hz and 250 Hz are unaffected while the DC component has been attenuated by 30 dB.



Select the DC Blocker block by double-clicking on it and change the algorithm type from `IIR` to `Subtract mean`. Rerun the simulation. The spectral output from the DC Blocker shows that the `Subtract mean` estimation method results in a DC component of less than −100 dBW.



Try all three estimation methods. Modify the `IIR` and `FIR` parameters to illustrate the performance of the DC Blocker using the various estimation techniques.

DC Blocker with Fixed Point Data

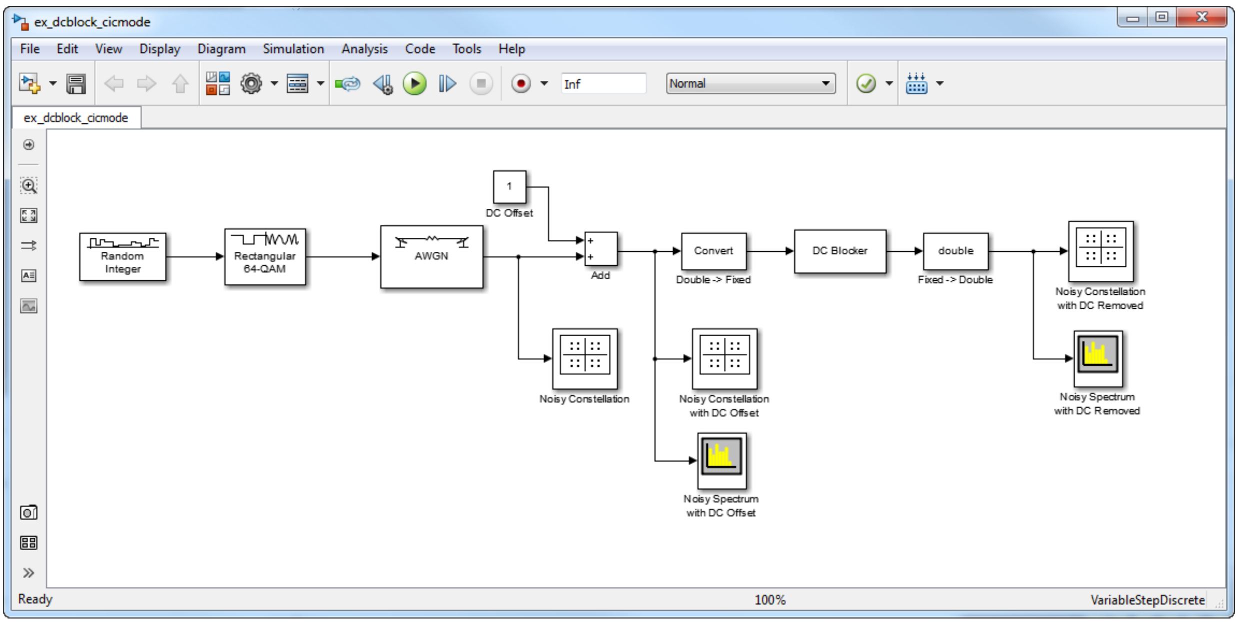
This example shows how to use the DC Blocker to remove a DC offset from fixed point data.

Load the DC Blocker example by typing `ex_dcblock_cicmode` in the MATLAB command prompt.

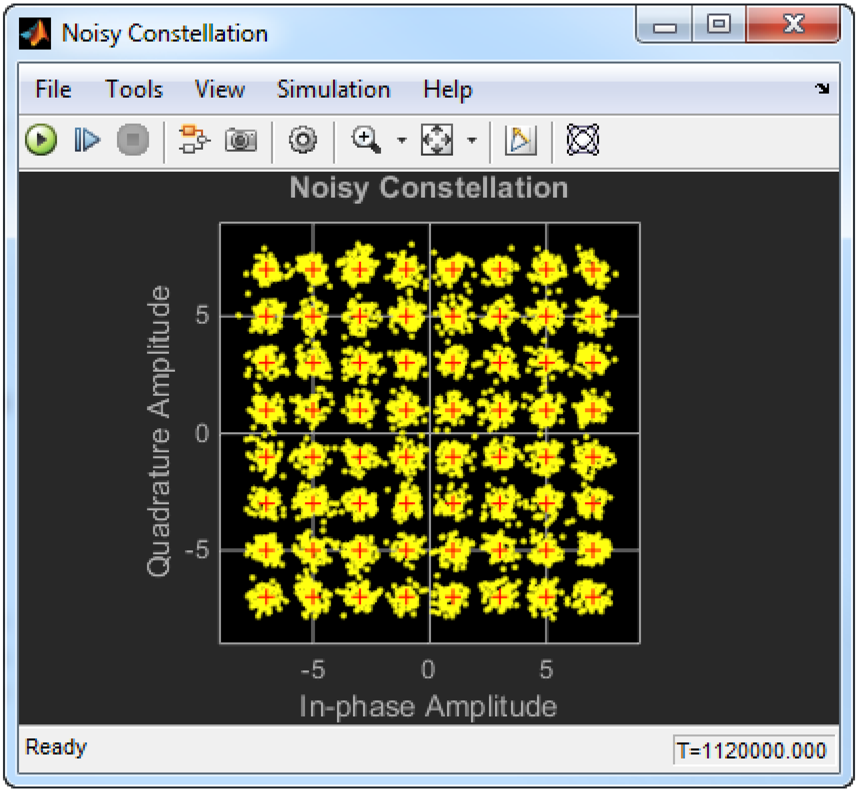
In the model:

- 64-QAM data passes through an AWGN channel.
- A DC offset of 1 is added to the signal .
- The Double -> Fixed block converts the data to 16-bit fixed point.
- The fixed-point data passes through the DC Blocker, which has the CIC algorithm selected, to remove the DC offset.
- The Fixed -> Double block converts the data back to floating point.

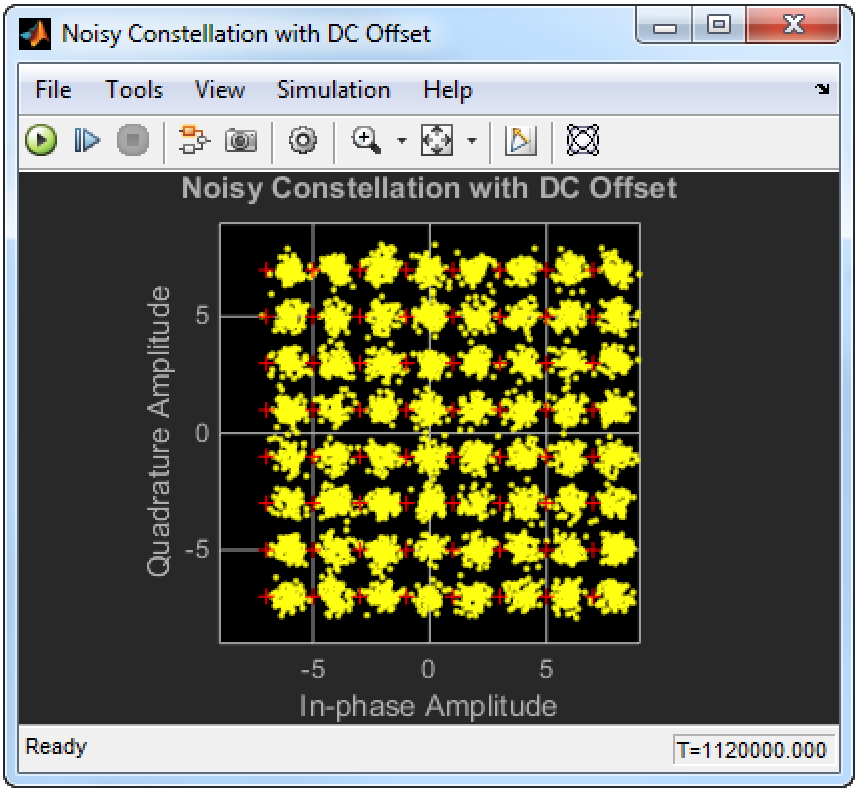
Constellation diagrams and spectrum analyzers are used to show the improvements from the DC Blocker.



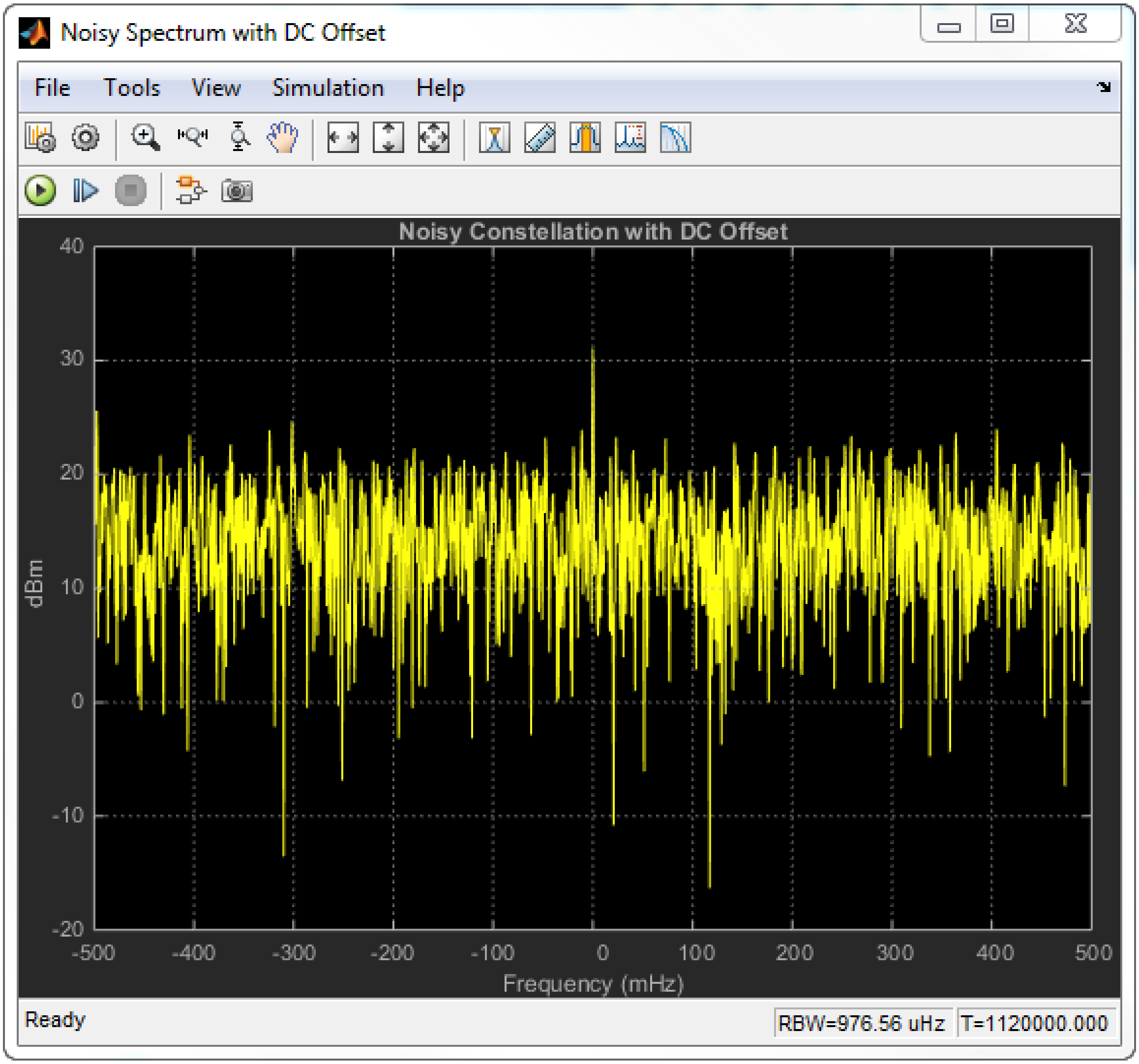
Run the simulation. The first constellation diagram, Noisy Constellation, shows a 64-QAM signal with white noise.



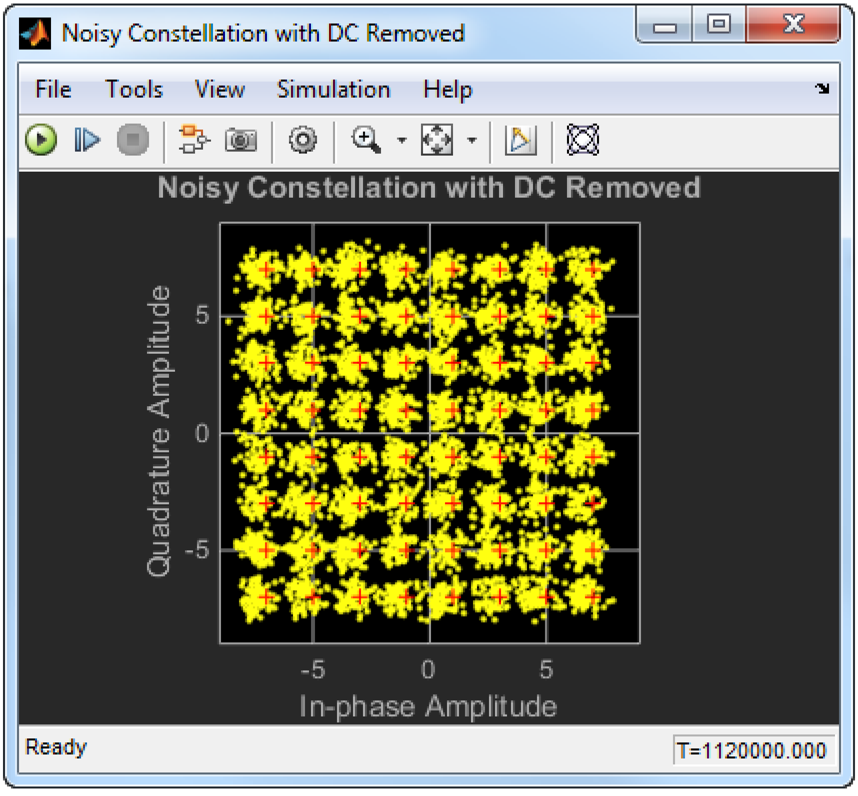
Observe the constellation diagram of the signal after the DC offset of 1 has been applied. The signal, represented by the yellow data points, has shifted one unit to the right.



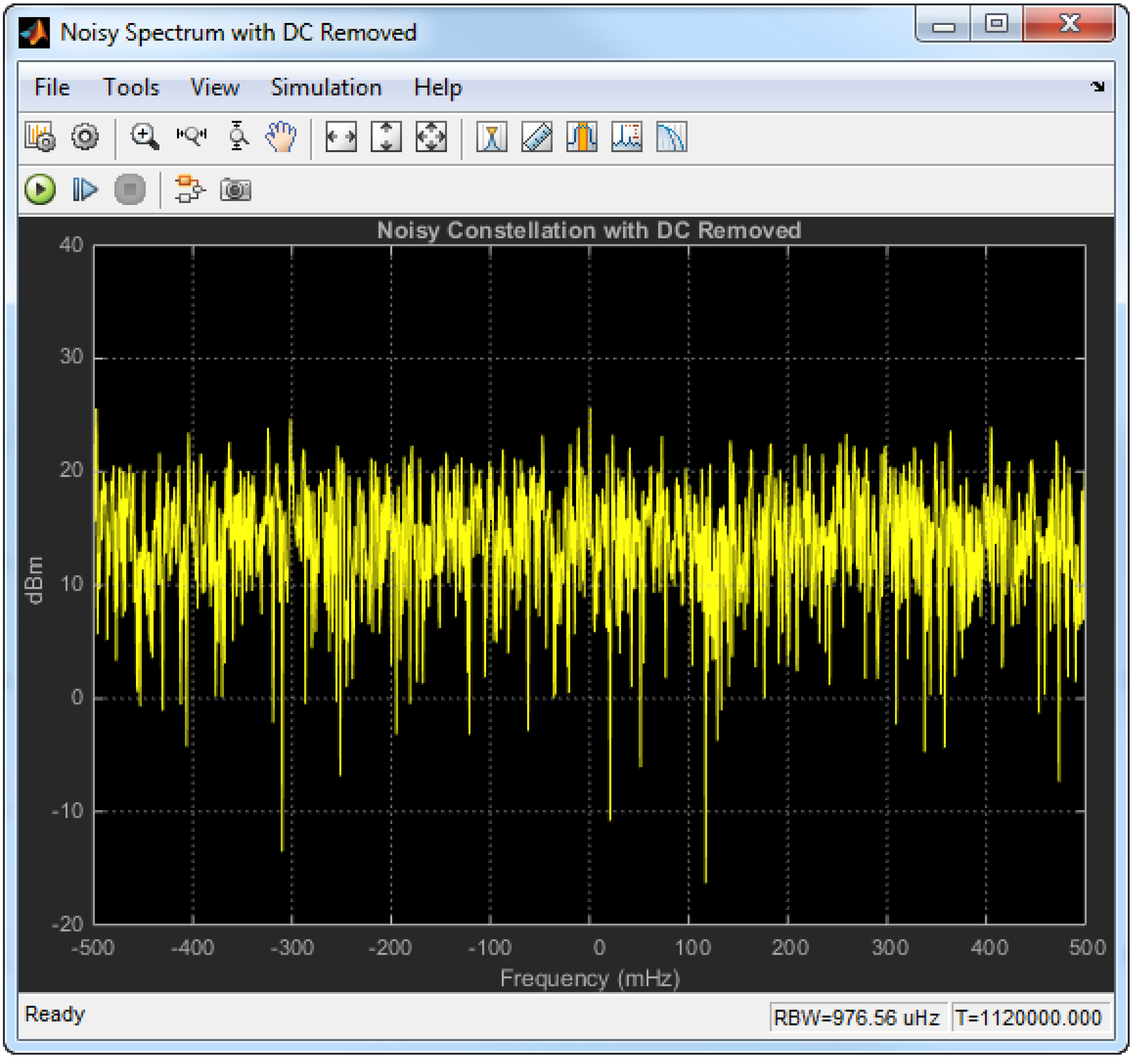
Look at the spectrum of the noisy signal with the DC offset. Notice that the signal has a peak at 0 Hz.



Observe the noisy constellation after the DC offset is removed. The signal has shifted back to the left so that the data clusters are aligned with their corresponding reference points.



Observe the spectrum of the noisy signal after the DC Blocker removes the offset. The spectral peak at 0 Hz has been removed.


Noisy Spectrum with DC Removed
File
Tools
View
Simulation
Help
Noisy Constellation with DC Removed
40
30
20
10
0
-10
-20
dBm
-500
-400
-300
-200
-100
0
100
200
300
400
500
Frequency (mHz)
Ready
RBW=976.56 uHz
T=1120000.000

To visualize the efficiency of the DC Blocker under different conditions, try changing the DC offset or the **Normalized bandwidth of lowpass IIR or CIC filter** parameter.

Algorithms

This block implements the algorithm, inputs, and outputs described on the dsp.DCBlocker reference page. The object properties correspond to the block parameters.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see DC Blocker.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed) • 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed) • 8-, 16-, and 32-bit signed integers

References

[1] Nezami, M., “Performance Assessment of Baseband Algorithms for Direct Conversion Tactical Software Defined Receivers: I/Q Imbalance Correction, Image Rejection, DC Removal, and Channelization”, MILCOM, 2002.

See Also

See Also

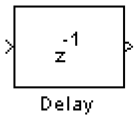
System Objects

dsp.DCBlocker | dsp.BiquadFilter | dsp.FIRFilter

Introduced in R2014a

Delay

Delay discrete-time input by specified number of samples or frames



Note: The Delay block from the `dspsigops` library has been replaced by the Delay block from the Discrete library in Simulink. Existing instances of the `dspsigops` Delay block will be replaced with Simulink Delay block when there is an exact match in functionality between the two blocks. For new models, use the Delay block from the Discrete library in Simulink.

Library

Signal Operations

`dspsigops`

Description

The Delay block delays a discrete-time input by the number of samples or frames specified in the **Delay units** and **Delay** parameters. The **Delay** value must be an integer value greater than or equal to zero. When you enter a value of zero for the **Delay** parameter, any initial conditions you might have entered have no effect on the output.

The Delay block allows you to set the initial conditions of the signal that is being delayed. The initial conditions must be numeric.

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each column of the M-by-N input matrix as an independent channel. The block delays each channel of the input as specified by the **Delay** parameter.

The **Delay** parameter can be a scalar integer by which the block equally delays all channels or a vector whose length is equal to the number of channels.

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be constant or varying along each channel. See the "Frame-Based Processing Examples" on page 2-463 section for more information.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats each element of the N-D input array as an independent channel. Thus, the total number of channels in the input is equal to the product of the input dimensions. The dimension of the output is the same as that of the input.

The **Delay** parameter can be a scalar integer by which to equally delay all channels or an N-D array of the same dimensions as the input array, containing nonnegative integers that specify the number of sample intervals to delay each channel of the input.

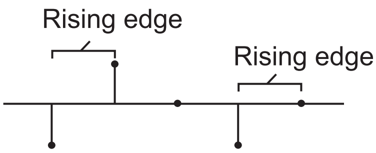
There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be the same or different within a channel. See the "Sample-Based Processing Examples" on page 2-467 section for more information.

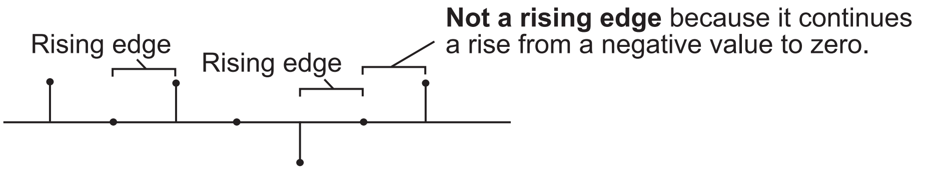
Resetting the Delay

The Delay block resets the delay whenever it detects a reset event at the optional `Rst` port. The reset sample time must be a positive integer multiple of the input sample time.

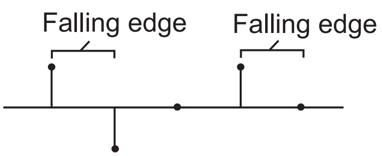
The reset event is specified by the **Reset port** parameter, and can be one of the following:

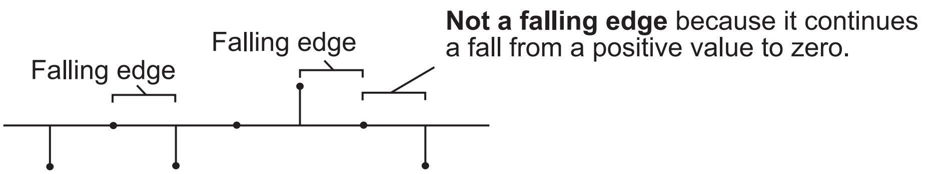
- `None` disables the `Rst` port.
- `Rising edge` triggers a reset operation when the `Rst` input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)





- `Falling edge` triggers a reset operation when the `Rst` input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)





- `Either edge` triggers a reset operation when the `Rst` input is `Rising edge` or `Falling edge` (as described earlier).
- `Non-zero sample` triggers a reset operation at each sample time that the `Rst` input is not zero.

Note When running simulations in the Simulink MultiTasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Time-Based Scheduling and Code Generation" (Simulink Coder).

This block supports Simulink virtual buses.

Examples

Frame-Based Processing Examples

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be constant or varying along each channel. The next sections describe the behavior of the block for each of these four cases:

- "Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel" on page 2-463
- "Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel" on page 2-464
- "Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel" on page 2-465
- "Case 4 — Use Different Initial Conditions for Each Channel and Within a Channel" on page 2-466

Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel

Enter a scalar value for the initial conditions. This value is used as the constant initial condition value for each of the channels.

For example, suppose your input is a matrix and you set the **Input processing** parameter to `Columns as channels (frame based)`.

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \ldots$$

You want the initial conditions of your three-channel signal to be identical and zero for the first frame:

1 Set the **Delay (frames)** parameter to `1`.
2 Clear the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.
3 Set the **Initial conditions** parameter to a scalar value of `0`.

   The output of the delay block is

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \ldots$$

   `0`, the scalar initial condition value, is used across the channels and within the channels for the first frame. This frame is the output at sample time zero.

Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel

The initial conditions must be a vector of length N, where $N \geq 1$. N is also equal to the number of channels in your signal. These initial condition values are used as the constant initial condition value for each of the channels.

For example, suppose your input is a matrix and you set the **Input processing** parameter to `Columns as channels (frame based)`.

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \ldots$$

You want the initial conditions of your three-channel signal to be `[0 10 20]` for the first frame:

1 Set the **Delay (frames)** parameter to `1`.
2 Select the **Specify different initial conditions for each channel** check box.
3 Clear the **Specify different initial conditions within a channel** check box.

4 Set the **Initial conditions** parameter to `[0 10 20]`.

The output of the delay block is

$$\begin{bmatrix} 0 & 10 & 20 \\ 0 & 10 & 20 \\ 0 & 10 & 20 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \ldots$$

The initial condition vector expands to create the frame that is output at sample time zero. Different initial conditions are used for each channel, but the same initial condition value is used with a channel.

Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel

In this case, the **Delay** parameter can be a scalar integer by which to equally delay all channels or a vector whose length is equal to the number of channels. All the values of this vector must be equal.

Enter the initial conditions as a vector. These values are used as the initial condition value along each of the channels to be delayed. The initial condition vector must have length equal to the value of the **Delay (frames)** parameter multiplied by the frame length. For example, if you want to delay your signal by two frames with frame length two and an initial condition value of 3, enter your initial condition vector as `[3 3 3 3]`.

For example, suppose your input is a matrix and you set the **Input processing** parameter to `Columns as channels (frame based)`.

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \ldots$$

You want the initial conditions of your three-channel signal to be the same along each of the channels to be delayed:

1 Set the **Delay (frame)** parameter to `1`.

2 Clear the **Specify different initial conditions for each channel** check box.

3. Select the **Specify different initial conditions within a channel** check box.
4. Set the **Initial conditions** parameter to `[10 20 30]`.

   The output of the delay block is

$$\begin{bmatrix} 10 & 10 & 10 \\ 20 & 20 & 20 \\ 30 & 30 & 30 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \ldots$$

   The initial condition vector defines the initial condition values within each of the three channels. The same initial conditions are used for each channel, but different initial condition values are used with a channel.

Case 4 — Use Different Initial Conditions for Each Channel and Within a Channel

Enter a cell array for your initial condition values. Or, when you have a scalar delay value, you can enter the initial conditions as a matrix.

For example, suppose your input is a matrix and you set the **Input processing** parameter to `Columns as channels (frame based)`.

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \ldots$$

You want the initial conditions of your three-channel signal to be different for each channel and along each channel.

1. Set the **Delay (frames)** parameter to `1`.
2. Select the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.
3. Set the **Initial conditions** parameter to either `[10 20 30; 40 50 60; 70 80 90]` or `{[10 40 70];[20 50 80];[30 60 90]}`. Each cell of the cell array represents the delay along one channel.

   Regardless of whether you use a matrix or cell array, the output of the delay block is

$$\begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix} \ldots$$

The initial condition matrix is the output at sample time zero. The elements of the initial condition cell array define the initial condition values within each channel. The first element, a vector, represents the initial conditions within channel 1. The second element, a vector, represents the initial conditions within channel 2, and so on. Different initial conditions are used for each channel and within the channels.

Sample-Based Processing Examples

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be the same or different along each channel. The next sections describe the behavior of the block for each of these four cases:

- "Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel" on page 2-467
- "Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel" on page 2-468
- "Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel" on page 2-469
- "Case 4 — Use Different Initial Conditions for Each Channel and Within a Channel" on page 2-470

Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel

Enter a scalar value for the initial conditions. This value is used as the constant initial condition value for each of the channels.

For example, suppose your input is a matrix and you set the **Input processing** parameter to `Elements as channels (sample based)`.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \ldots$$

You want the initial conditions of your four-channel signal to be identical and zero for the first two samples:

1 Set the **Delay (samples)** parameter to 2.
2 Clear the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes.
3 Set the **Initial conditions** parameter to a scalar value of 0.

   The output of the delay block is

$$\begin{bmatrix}0 & 0\\0 & 0\end{bmatrix},\begin{bmatrix}0 & 0\\0 & 0\end{bmatrix},\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix},\begin{bmatrix}2 & 2\\2 & 2\end{bmatrix},\begin{bmatrix}3 & 3\\3 & 3\end{bmatrix},\ldots$$

   0, the scalar initial condition value, is used for each channel and within the channels. It is the output at sample time zero and sample time one.

Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel

The initial conditions must be an N-D array for N-D input. The initial conditions must have the same dimensions as the input data. These initial condition values are used as the constant initial condition value for each of the channels.

For example, suppose your input is a matrix and you set the **Input processing** parameter to `Elements as channels (sample based)`.

$$\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix},\begin{bmatrix}2 & 2\\2 & 2\end{bmatrix},\begin{bmatrix}3 & 3\\3 & 3\end{bmatrix},\ldots$$

You want the initial conditions of your four-channel signal to be

$$\begin{bmatrix}7 & 9\\11 & 13\end{bmatrix}$$

for the first two samples:

1 Set the **Delay (samples)** parameter to 2.

2 Select the **Specify different initial conditions for each channel** check box.

3 Clear the **Specify different initial conditions within a channel** check box.

4 Set the **Initial conditions** parameter to `[7 9; 11 13]`.

The output of the delay block is

$$\begin{bmatrix} 7 & 9 \\ 11 & 13 \end{bmatrix}, \begin{bmatrix} 7 & 9 \\ 11 & 13 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \ldots$$

The initial condition matrix is the output at sample time zero and sample time one. Different initial conditions are used for each channel; the same initial condition value is used within a channel.

Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel

In this case, for N-D sample-based inputs, the initial conditions parameter must be a vector whose length is equal to the delay value, specified by the **Delay** parameter. The values in this vector are used as the initial condition values along each of the channels to be delayed.

For example, suppose your input is a matrix and you set the **Input processing** parameter to `Elements as channels (sample based)`.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \ldots$$

You want the initial conditions of your four channel signal to be the same along each of the channels to be delayed:

1 Set the **Delay (samples)** parameter to `2`.

2 Clear the **Specify different initial conditions for each channel** check box.

3 Select the **Specify different initial conditions within a channel** check box.

4 Set the **Initial conditions** parameter to `[10 20]`.

The output of the delay block is

$$\begin{bmatrix}10 & 10\\10 & 10\end{bmatrix},\begin{bmatrix}20 & 20\\20 & 20\end{bmatrix},\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix},\begin{bmatrix}2 & 2\\2 & 2\end{bmatrix},\begin{bmatrix}3 & 3\\3 & 3\end{bmatrix},\ldots$$

The first element of the initial conditions vector is the output, for all channels, at sample time zero. The second element of the initial conditions vector is the output, for all channels, at sample time one. The same initial conditions are used for each channel, but different initial condition values are used within a channel.

Case 4 — Use Different Initial Conditions for Each Channel and Within a Channel

Enter a cell array for your initial condition values. The cell array must be the same size as your input signal. Each cell of the cell array represents the delay values for one channel, and must be a vector of size equal to the delay value. If you have a vector or scalar input and a scalar delay value, you can enter the initial conditions as a matrix.

For example, suppose your input is a matrix and you set the **Input processing** parameter to `Elements as channels (sample based)`.

$$[1\quad 1],\ \ [2\quad 2],\ \ [3\quad 3],\ldots$$

You want the initial conditions of your two channel signal to be different for each channel and along each channel:

1. Set the **Delay (samples)** parameter to `2`.
2. Select the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes.
3. Set the **Initial conditions** parameter to `[10 20; 30 40]`.

   The output of the delay block is

   $$[10\quad 20],\ \ [30\quad 40],\ \ [1\quad 1],\ \ [2\quad 2]\ldots$$

   The first row of the initial conditions vector is the output at sample time zero. The second row of the initial conditions vector is the output at sample time one. Different initial conditions are used for each channel and within the channels.

In addition, suppose your input is a matrix and you set the **Input processing** parameter to `Elements as channels (sample based)`.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \ldots$$

You want the initial conditions of your two-channel signal to be different for each channel and along each channel:

1. Set the **Delay (samples)** parameter to `2`.
2. Select the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.
3. Set the **Initial conditions** parameter to `{[11 15] [12 16]; [13 17] [14 18]}`. The dimensions of the cell array match the dimensions of the input. Also, each element of the cell array represents the initial conditions within one channel.

   The output of the delay block is

   $$\begin{bmatrix} 11 & 12 \\ 13 & 14 \end{bmatrix}, \begin{bmatrix} 15 & 16 \\ 17 & 18 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \ldots$$

   Each element of the cell array represents the initial conditions within a channel. The first element, a vector, represents the initial conditions within channel 1. The second element, a vector, represents the initial conditions within channel 2, and so on. Different initial conditions are used for each channel and within the channels.

Parameters

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note: The option `Inherit from input (this choice will be removed - see release notes)` will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Delay units

Select whether you want to delay your input by a specified number of `Samples` or `Frames`. This parameter appears only when you set the **Input processing** parameter to `Columns as channels (frame based)`.

Delay (samples) or Delay (frames)

See "Sample-Based Processing" on page 2-461 and "Frame-Based Processing" on page 2-460 for a description of what format to use for each configuration of the block dialog.

Specify different initial conditions for each channel

Select this check box when you want the initial conditions to vary across the channels. When you do not select this check box, the initial conditions are the same across the channels.

Specify different initial conditions within a channel

Select this check box when you want the initial conditions to vary within the channels. When you do not select this check box, the initial conditions are the same within the channels.

Initial conditions

Enter a scalar, vector, matrix, or cell array of initial condition values, depending on your choice for the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes. See "Sample-Based Processing" on page 2-461 and "Frame-Based Processing" on page 2-460 for a description of what format to use for each configuration of the block dialog.

Reset port

Determines the reset event that causes the block to reset the delay. For more information, see "Resetting the Delay" on page 2-461.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

For more information on implementations, properties, and restrictions for HDL code generation, see Delay.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

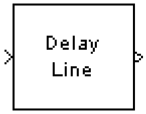
See Also

dsp.Delay | Unit Delay | Variable Integer Delay | `Variable Fractional Delay`

Introduced before R2006a

Delay Line

Rebuffer sequence of inputs



Library

Signal Management / Buffers

`dspbuff3`

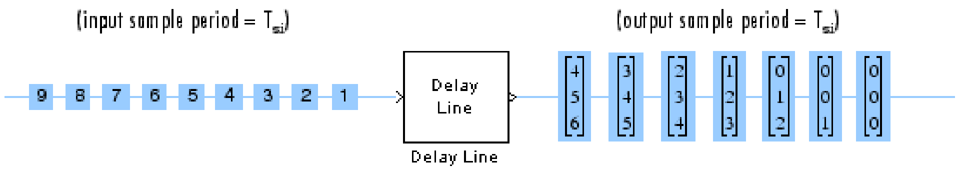
Description

The Delay Line block rebuffers a sequence of M_i-by-N matrix inputs into a sequence of M_o-by-N matrix outputs, where M_o is the output frame size you specify in the **Delay line size** parameter. Depending on whether M_o is greater than, less than, or equal to the input frame size, M_i, the output frames can be underlapped or overlapped. The block always performs frame-based processing and rebuffers each of the N input channels independently.

When $M_o > M_i$, the output frame overlap is the difference between the output and input frame size, M_o-M_i. When $M_o < M_i$, the output is underlapped; the Delay Line block discards the first M_i-M_o samples of each input frame so that only the last M_o samples are buffered into the corresponding output frame. When $M_o = M_i$, the output data is identical to the input data, but is delayed by the latency of the block. Due to the block's latency, the outputs are always delayed by one frame, the entries of which you specify in the **Initial conditions** parameter (see “Initial Conditions” on page 2-475).

The output frame period is equal to the input frame period (T_{fo}=T_{fi}). The output sample period, T_{so}, is therefore equal to T_{fi}/M_o, or equivalently, $T_{si}(M_i/M_o)$

In the most typical use, each output differs from the preceding output by only one sample, as illustrated below for scalar input.



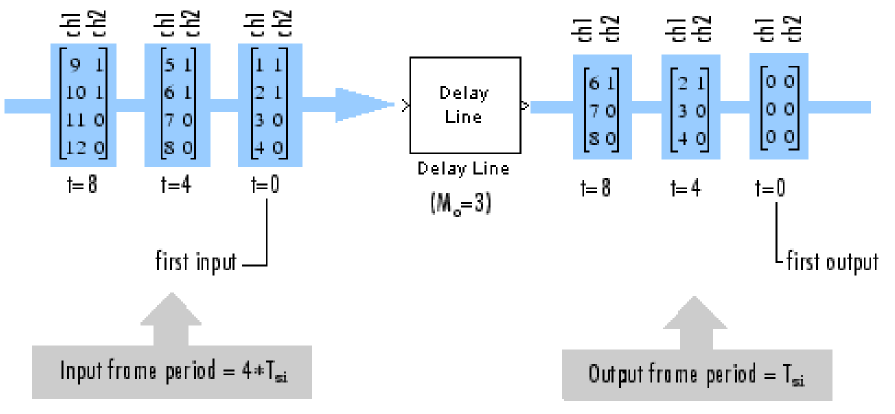
Note that the first output of the block in the example above is all zeros; this is because the **Initial Conditions** parameter is set to zero.

Initial Conditions

The Delay Line block's buffer is initialized to the value specified by the **Initial conditions** parameter. The block outputs this buffer at the first simulation step (t=0). When the block's output is a vector, the **Initial conditions** can be a vector of the same size, or a scalar value to be repeated across all elements of the initial output. When the block's output is a matrix, the **Initial conditions** can be a matrix of the same size, a vector (of length equal to the number of matrix rows) to be repeated across all columns of the initial output, or a scalar to be repeated across all elements of the initial output.

Examples

In the following ex_delayline_ref2 model, the block rebuffers a two-channel input with a **Delay line size** of 3.



The first output frame in this example is due to the latency of the Delay Line block; it is all zeros because the **Initial conditions** parameter is set to zero. Because the input frame size of 4 is larger than the output frame size of 3, only the last three samples in each input frame are propagated to the corresponding output frame. The frame periods of the input and output are the same, and the output sample period is $T_{si}(M_i/M_o)$, or 4/3 the input sample period.

Parameters

Delay line size

Specify the number of rows in output matrix, M_o.

Initial conditions

Specify the value of the block's initial output. When the block outputs a vector, the **Initial conditions** can be a vector of the same size, or a scalar value to be repeated across all elements of the initial output. When the block outputs a matrix, the **Initial conditions** can be a matrix of the same size, a vector (of length equal to the number of matrix rows) to be repeated across all columns of the initial output, or a scalar to be repeated across all elements of the initial output.

Allow direct feedthrough

When you select this check box, the input data is not delayed by an extra frame before it is available at the output buffer. Instead, the input data is available immediately at the output port of the block.

Show En_Out port for selectively enabling output

When you select this check box, the En_Out port appears on the block icon. This block uses a circular buffer internally even though the output is linear. This means that for valid output, data from the circular buffer has to be linearized. The En_Out port determines whether or not a valid output needs to be computed based on the value of its `Boolean` input. If the input value to the En_Out port is `1`, the block output is linearized, and thus is valid. Otherwise, the output is not linearized, and is invalid. This allows the block to be more efficient when the tapped Delay Line's output is not required at each sample time.

Note that when the input value to the En_Out port is `0`, the block can give different results depending on the state of the model. The results can appear to match valid results or can be invalid, and they cannot be predicted. You should ignore the block output in all cases when the input to the En_Out port is `0`.

Hold previous value when the output is disabled

This parameter only appears and applies when the **Show En_Out port for selectively enabling output** parameter is selected. Use this parameter to specify the block output at those time steps when the internal state buffer is not being linearized to output valid data.

When you do not select this check box, the block memory is free to be used by other parts of the model, and the signal on the output port is invalid. When you select this check box, the most recent valid value is held on the output port, and slightly more memory is used by the block.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

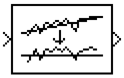
See Also

`Buffer`	DSP System Toolbox

Introduced before R2006a

Detrend

Remove linear trend from vectors



Library

Statistics

`dspstat3`

Description

The Detrend block removes a linear trend from the length-M input vector, u, by subtracting the straight line that best fits the data in the least squares sense.

The least squares line, $\hat{u} = ax + b$, is the line with parameters a and b that minimizes the quantity

$$\sum_{i=1}^{M}(u_i - u_i)^2$$

for M evenly-spaced values of x, where u_i is the ith element in the input vector. The output, $y = u\text{-}\hat{u}$, is always an M-by-1 column vector.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

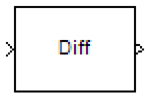
Cumulative Sum	DSP System Toolbox

Difference	DSP System Toolbox
Least Squares Polynomial Fit	DSP System Toolbox
Unwrap	DSP System Toolbox
`detrend`	MATLAB

Introduced before R2006a

Difference

Compute element-to-element difference along specified dimension of input



Library

Math Functions / Math Operations

`dspmathops`

Description

The Difference block computes the difference between adjacent elements in rows, columns, or a specified dimension of the input array `u`. You can configure the block to compute the difference only within the current input, or across consecutive inputs (running difference).

Basic Operation

When you set the **Running difference** parameter to `No`, the block computes the difference between adjacent elements in the specified dimension of the current input. In this mode, the block can compute the difference along the columns, rows, or a specified dimension of the input.

Columnwise Differencing

When you set the **Difference along** parameter to `Columns`, the block computes differences between adjacent elements in each column of the input.

```
y = diff(u)      % Equivalent MATLAB code
```

For M-by-N inputs, the output is an (M-1)-by-N matrix whose jth column has the following elements:

$$y_{i,j} = u_{i+1,j} - u_{i,j} \quad 1 \le i \le (M-1)$$

Rowwise Differencing

When you set the **Difference along** parameter to `Rows`, the block computes differences between adjacent elements in each row of the input.

```
y = diff(u,[],2)      % Equivalent MATLAB code
```

The output is an M-by-(N-1) matrix whose ith row has the following elements:

$$y_{i,j} = u_{i,j+1} - u_{i,j} \quad 1 \le j \le (N-1)$$

Differencing Along Arbitrary Dimensions

When you set the **Difference along** parameter to `Specified dimension`, the behavior of the block is an extension of the rowwise differencing described earlier. The block computes differences between adjacent elements along the dimension you specify in the **Dimension** parameter.

```
y = diff(u,[],d) % Equivalent MATLAB code where d is the dimension
```

The output is an array whose length in the specified dimension is one less than that of the input, and whose lengths in other dimensions are unchanged. For example, consider an M-by-N-by-P-by-R input array with elements *u(i,j,k,l)* and assume that **Dimension** is `3`. The output of the block is an M-by-N-by-(P–1)-by-R array with the following elements:

$$y_{i,j,k,l} = u_{i,j,k+1,l} - u_{i,j,k,l} \quad 1 \le k \le (P-1)$$

Running Operation

When you set the **Running difference** parameter to `Yes`, the block computes the running difference along the columns of the input.

For an M-by-N input matrix, the output is an M-by-N matrix whose jth column has the following elements:

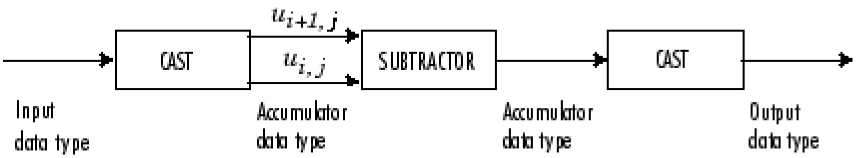
$$y_{i,j} = u_{i+1,j} - u_{i,j} \quad 2 \le i \le (M-1)$$

In running mode, the first element of the output for each column is the first input element minus the last input element of the previous frame. For the first frame, the block subtracts zero from the first input element.

$$y_{1,j}(t) = u_{1,j}(t) - u_{M,j}(t - T_f)$$

Fixed-Point Data Types

The following diagram shows the data types used within the Difference block for fixed-point signals.

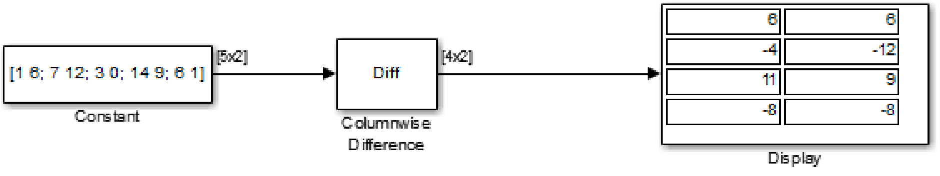


You can set the accumulator and output data types in the block dialog as discussed in “Dialog Box” on page 2-483 .

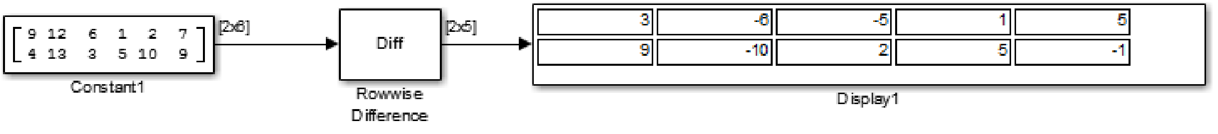
Examples

For an example showing the modes of the Difference block, open ex_difference.

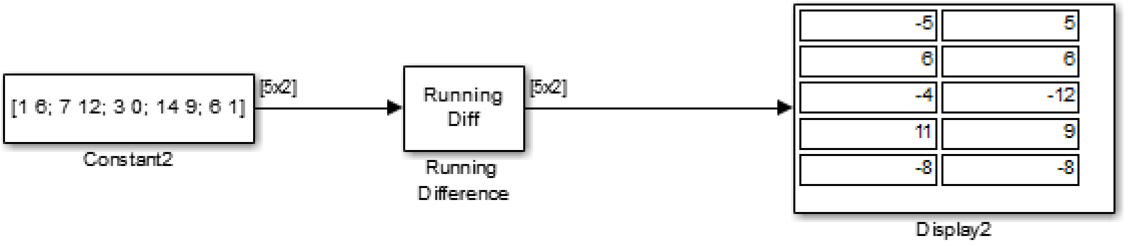
The following figure shows the output of the block when it is differencing along columns in nonrunning mode.



The following figure shows the output of the block when it is differencing along rows in nonrunning mode.

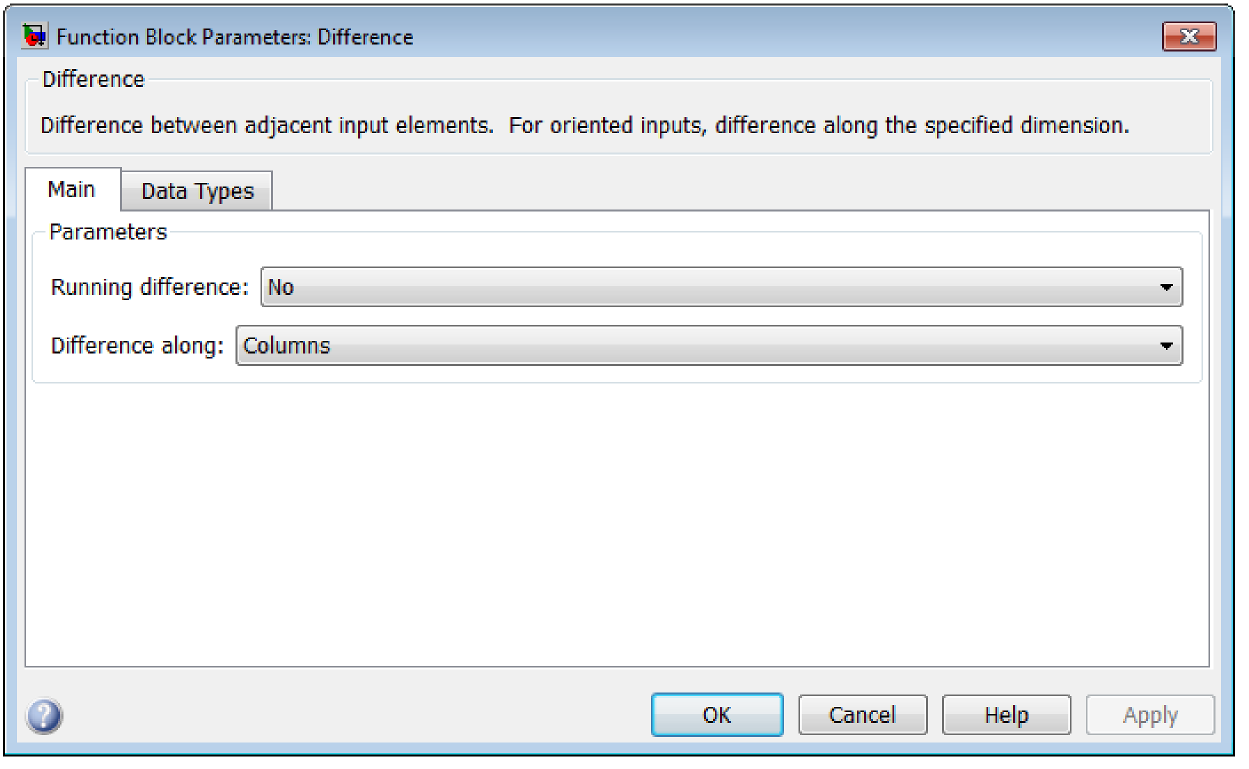


The following figure shows the second frame of block output when the block is computing the running difference.



Dialog Box

The **Main** pane of the Difference block appears as follows.



Running difference

Specify whether or not the block computes a running difference. When you select `No`, the block computes the difference between adjacent elements in the current input. When you select `Yes`, the block computes the running difference across consecutive inputs. In running mode, the block always computes the difference along the columns of the input. See "Running Operation" on page 2-481 for more information.

Difference along

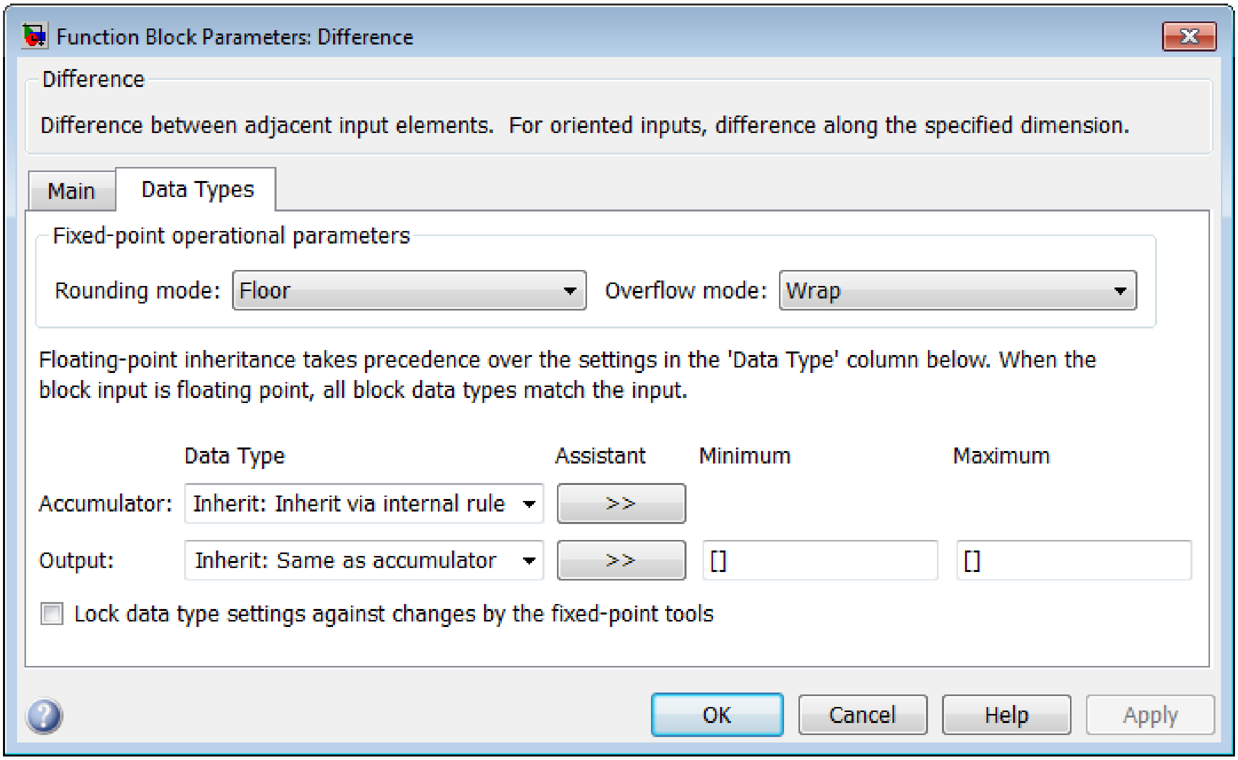
Specify whether the block computes the difference along the columns, rows, or specified dimension of the input.

Dimension

Specify the one-based dimension along which to compute element-to-element differences.

This parameter is only visible when you select `Specified dimension` for the **Difference along** parameter.

The **Data Types** pane of the Difference block appears as follows.



Note: Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-482 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-482 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

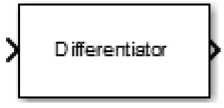
See Also

Cumulative Sum	DSP System Toolbox
`diff`	MATLAB

Introduced before R2006a

Differentiator Filter

Direct form FIR fullband differentiator filter



Library

Filtering / Filter Designs

`dspfdesign`

Description

The Differentiator Filter block applies a fullband differentiator filter on the input signal to differentiate all its frequency components. The block uses an FIR equiripple filter design to design the differentiator filter. The ideal frequency response of the differentiator is $D(\omega) = j\omega$ for $-\pi \leq \omega \leq \pi$.

You can design the filter with minimum order or with a specifies order.

The input signal can be a real- or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel.

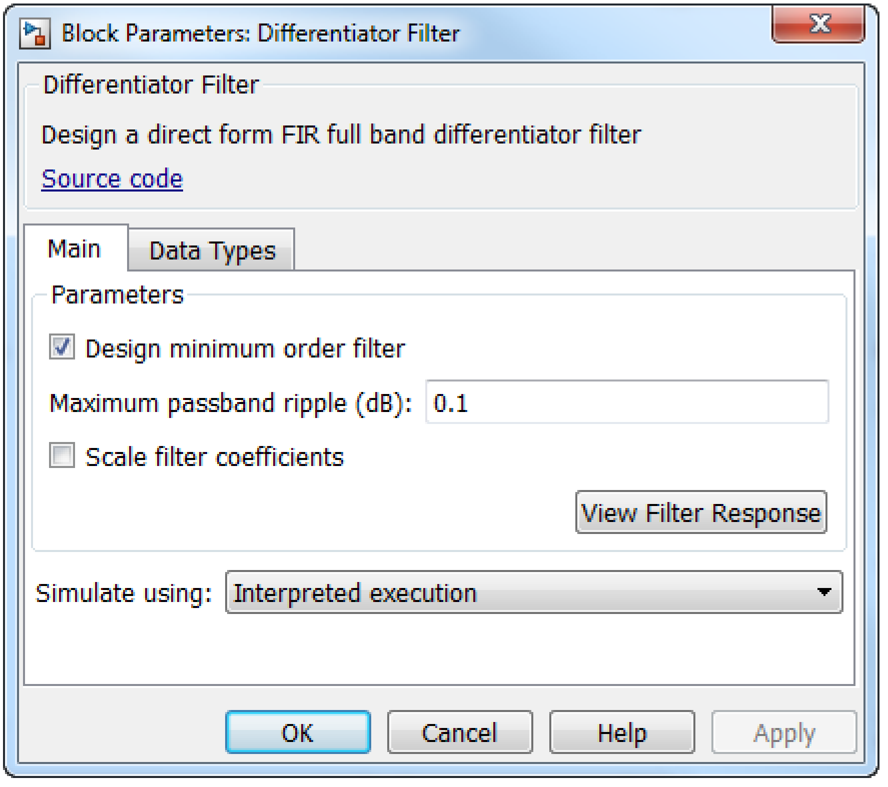
This block supports variable-size input, enabling you to change the channel length during simulation. The output port properties, such as data type, complexity, and dimension, are identical to the input port properties. The block supports fixed-point operations.

Examples

- “Group Delay Estimation”

Dialog Box

Main Tab



Design minimum order filter

When you select this check box, the block designs a filter with the minimum order, with the passband ripple specified in **Maximum passband ripple (dB)**. When you clear this check box, specify the order of the filter in **Filter order**.

By default, this check box is selected.

Filter order

Filter order of the differentiator filter, specified as an odd positive scalar integer. You can specify the filter order only when **Design minimum order filter** check box is not selected. The default is 31.

Maximum passband ripple (dB)

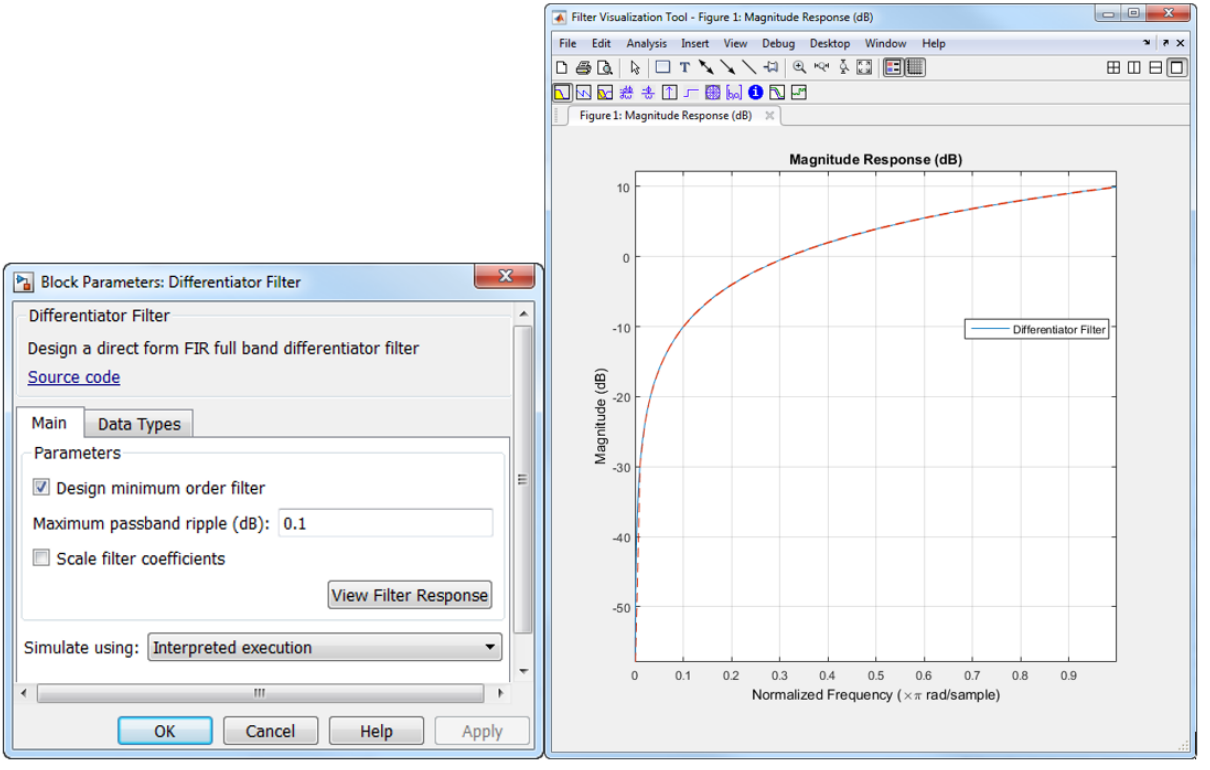
Maximum ripple of the filter response in the passband, specified as a real positive scalar in dB. The default is `0.1`.

Scale filter coefficients

When you select this check box, the filter coefficients are scaled to preserve the input dynamic range. By default, this check box is not selected.

View Filter Response

Opens the Filter Visualization Tool (`fvtool`) and displays the magnitude and phase response of the Differentiator Filter block. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

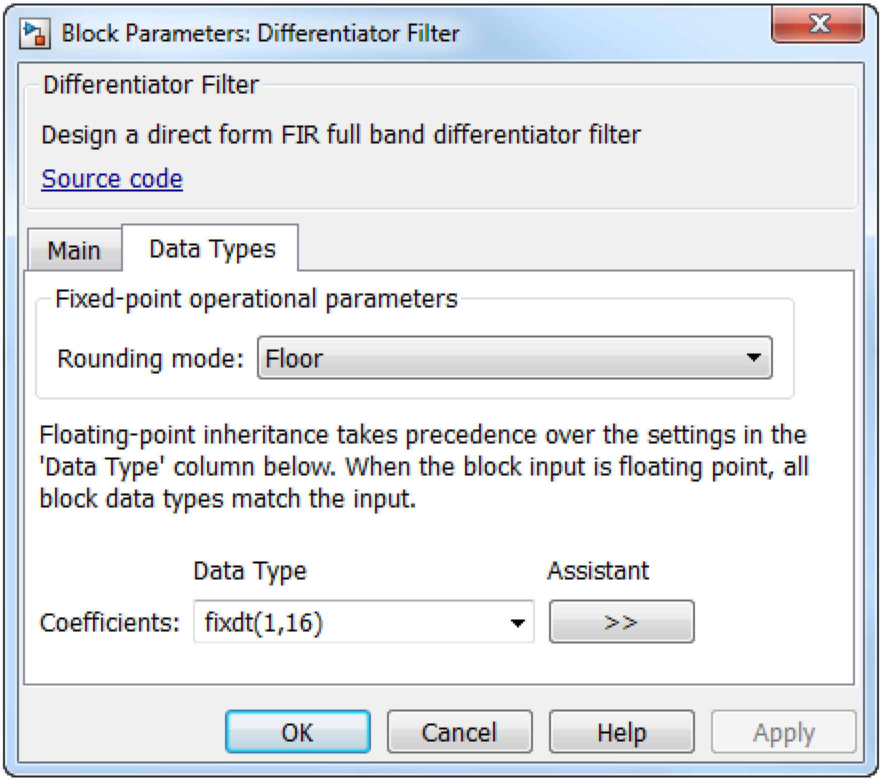
- `Interpreted execution` (default)

  Simulate model using the MATLAB interpreter. This option shortens startup time and has faster simulation speed than `Code generation`.

- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster subsequent simulations.

Data Types Tab



Rounding mode

Rounding method for the output fixed-point operations. The rounding methods are `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, and `Zero`. The default is `Floor`.

Coefficients

Fixed-point data type of the coefficients, specified as one of the following:

- `fixdt(1,16)` (default) — Signed fixed-point data type of word length `16`, with binary point scaling. The block determines the fraction length automatically from the coefficient values such that the coefficients occupy the maximum representable range without overflowing.

- `fixdt(1,16,0)` — Signed fixed-point data type of word length `16` and fraction length `0`. You can change the fraction length to any other integer value.
- `<data type expression>` — Specify the data type using an expression that evaluates to a data type object, for example, numeric type (`fixdt([ ],16, 15)`). Specify the sign mode of this data type as `[ ]` or `true`.
- `Refresh Data Type` — Refresh to the default data type.

Click the **Show data type assistant** button >> to display the data type assistant, which helps you set the stage input parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

The word length of the output is same as the word length of the input. The fraction length of the output is computed such that the entire dynamic range of the output can be represented without overflow. For details on how the block computes the fraction length, see “Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters”.

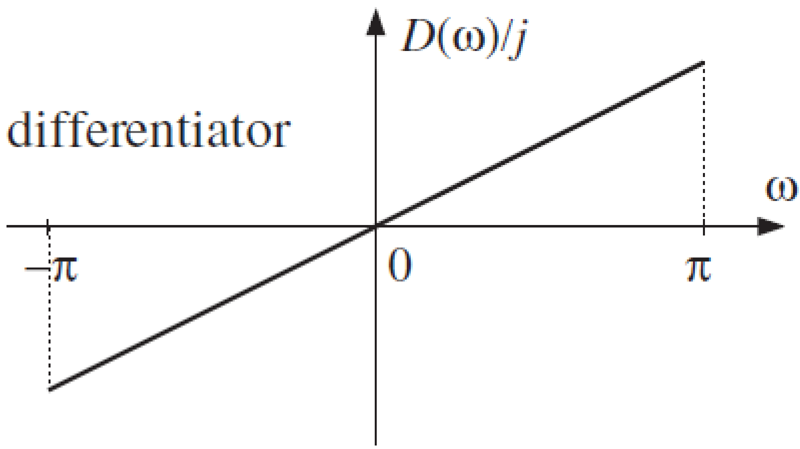
Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed or unsigned)
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed or unsigned)

Algorithms

Differentiator Filter

Differentiator computes the derivative of a signal. The frequency response of an ideal differentiator filter is given by $D(\omega) = j\omega$, defined over the Nyquist interval $-\pi \le \omega \le \pi$.



The frequency response is antisymmetric and is linearly proportional to the frequency.

`dsp.Differentiator` object acts as a differentiator filter. This object condenses the two-step process into one. For the minimum order design, the object uses generalized Remez FIR filter design algorithm. For the specified order design, the object uses the Parks-McClellan optimal equiripple FIR filter design algorithm. The filter is designed as a linear phase Type-IV FIR filter with a Direct form structure.

The ideal differentiator has an antisymmetric impulse response given by $d(n) = -d(-n)$. Hence $d(0) = 0$. The differentiator must have zero response at zero frequency.

Linear-Phase FIR Differentiator Filter

The impulse response of an antisymmetric linear-phase FIR filter is given by $h(n) = -h(M-1-n)$, where M is the length of the filter. Because the filter is antisymmetric, you can use this type of FIR filter to design the linear-phase FIR differentiators.

Consider the design of linear-phase FIR differentiators based on the Chebyshev approximation criterion.

If *M* is odd, the real-valued frequency response of the FIR filter, $H_r(\omega)$, has the characteristics that $H_r(0) = 0$ and $H_r(\pi) = 0$. This filter satisfies the condition of zero response at zero frequency. However, it is not fullband because $H_r(\pi) = 0$. This differentiator has a linear response over the limited frequency range $[0\ 2\pi f_p]$, where f_p is the bandwidth of the differentiator. The absolute error between the desired response and the Chebyshev approximation increases as ω increases from 0 to $2\pi f_p$.

If *M* is even, the real-valued frequency response of the FIR filter, $H_r(\omega)$, has the characteristics that $H_r(0) = 0$ and $H_r(\pi) \neq 0$. This filter satisfies the condition of zero response at zero frequency. It is fullband and this design results in a significantly smaller approximation error than comparable odd-length differentiators. Hence, even-length (odd order) differentiators are preferred in practical systems.

See Also

`Variable Bandwidth FIR Filter` | `Variable Bandwidth IIR Filter` | `Biquad Filter` | dsp.Differentiator | `Highpass Filter`

Introduced in R2015b

Differentiator Filter (Obsolete)

Design differentiator filter

Note: The Differentiator Filter (Obsolete) block has been replaced by the `Differentiator Filter` block. Existing instances of the Differentiator Filter (Obsolete) block will continue to operate. For new models, use the Differentiator Filter block.

Library

Filtering / Filter Designs

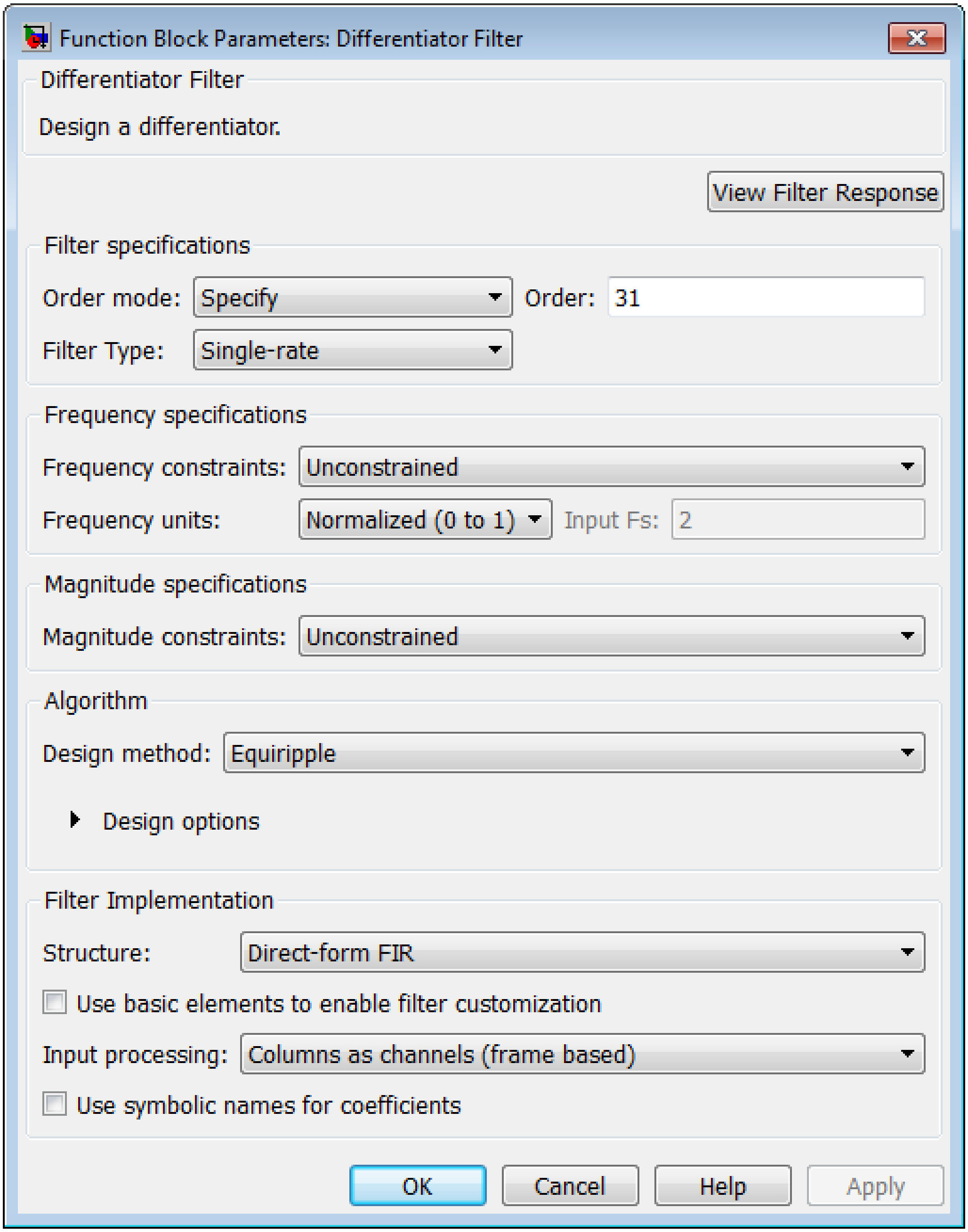
`dspfdesign`

Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See “Differentiator Filter Design — Main Pane” on page 5-742 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.


Function Block Parameters: Differentiator Filter
Differentiator Filter
Design a differentiator.
View Filter Response
Filter specifications
Order mode:
Specify
Order:
31
Filter Type:
Single-rate
Frequency specifications
Frequency constraints:
Unconstrained
Frequency units:
Normalized (0 to 1)
Input Fs:
2
Magnitude specifications
Magnitude constraints:
Unconstrained
Algorithm
Design method:
Equiripple
Design options
Filter Implementation
Structure:
Direct-form FIR
Use basic elements to enable filter customization
Input processing:
Columns as channels (frame based)
Use symbolic names for coefficients
OK
Cancel
Help
Apply

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

Order mode

Select either `Minimum` or `Specify` (the default). Selecting `Specify` enables the **Order** option so you can enter the filter order.

Order

Enter the filter order. This option is enabled only if you set the **Order mode** to `Specify`. The default order is 31.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

Frequency constraints

This option is only available when you specify the order of the filter design. Supported options are `Unconstrained` and `Passband edge and stopband edge`.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands. These parameters are only available for minimum-order designs.

Magnitude constraints

This option is only available when you specify the order of your filter design. The available **Magnitude constraints** depend on the value of the **Frequency constraints** parameter. When you set the **Frequency constraints** parameter to `Unconstrained`, the **Magnitude constraints** parameter must also be `Unconstrained`. When you set the **Frequency constraints** parameter to `Passband edge and stopband edge`, the **Magnitude constraints** parameter can be `Unconstrained`, `Passband ripple`, or `Stopband attenuation`.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop2

Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Wpass

Passband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to `Passband edge and stopband edge` and the **Design method** is `Equiripple`.

Wstop

Stopband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to `Passband edge and stopband edge` and the **Design method** is `Equiripple`.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note: The `Inherited (this choice will be removed — see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2006b

Digital Down-Converter

Translate digital signal from Intermediate Frequency (IF) band to baseband and decimate it

Library

Signal Operations

`dspsigops`

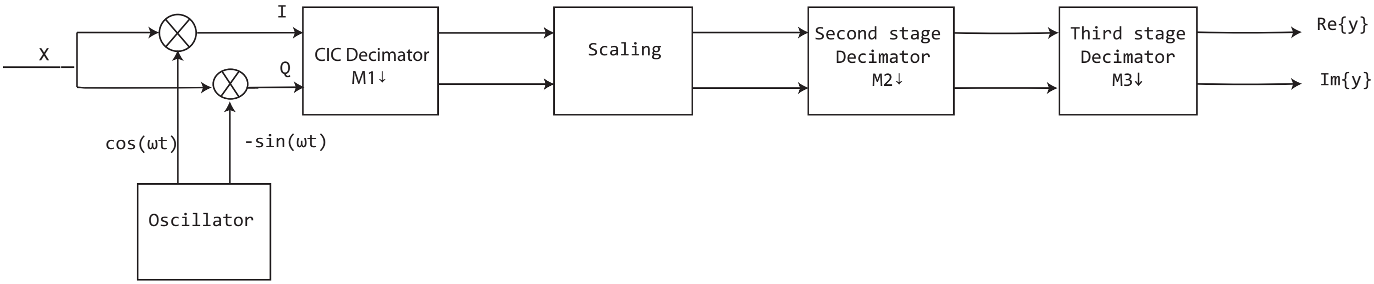
Description

The Digital Down-Converter (DDC) block converts a digitized real signal, centered at an intermediate frequency (IF) to a baseband complex signal centered at zero frequency. The DDC block downsamples the frequency down-converted signal using a cascade of three decimation filters. This block designs the decimation filters according to the filter parameters set in the block dialog box.

Structure

This block brings the capabilities of dsp.DigitalDownConverter System object to the Simulink environment.

The DDC block consists of a CIC decimator, a CIC compensator, and a FIR decimator. You can bypass the FIR Decimator, depending on how you set the DDC block parameters.



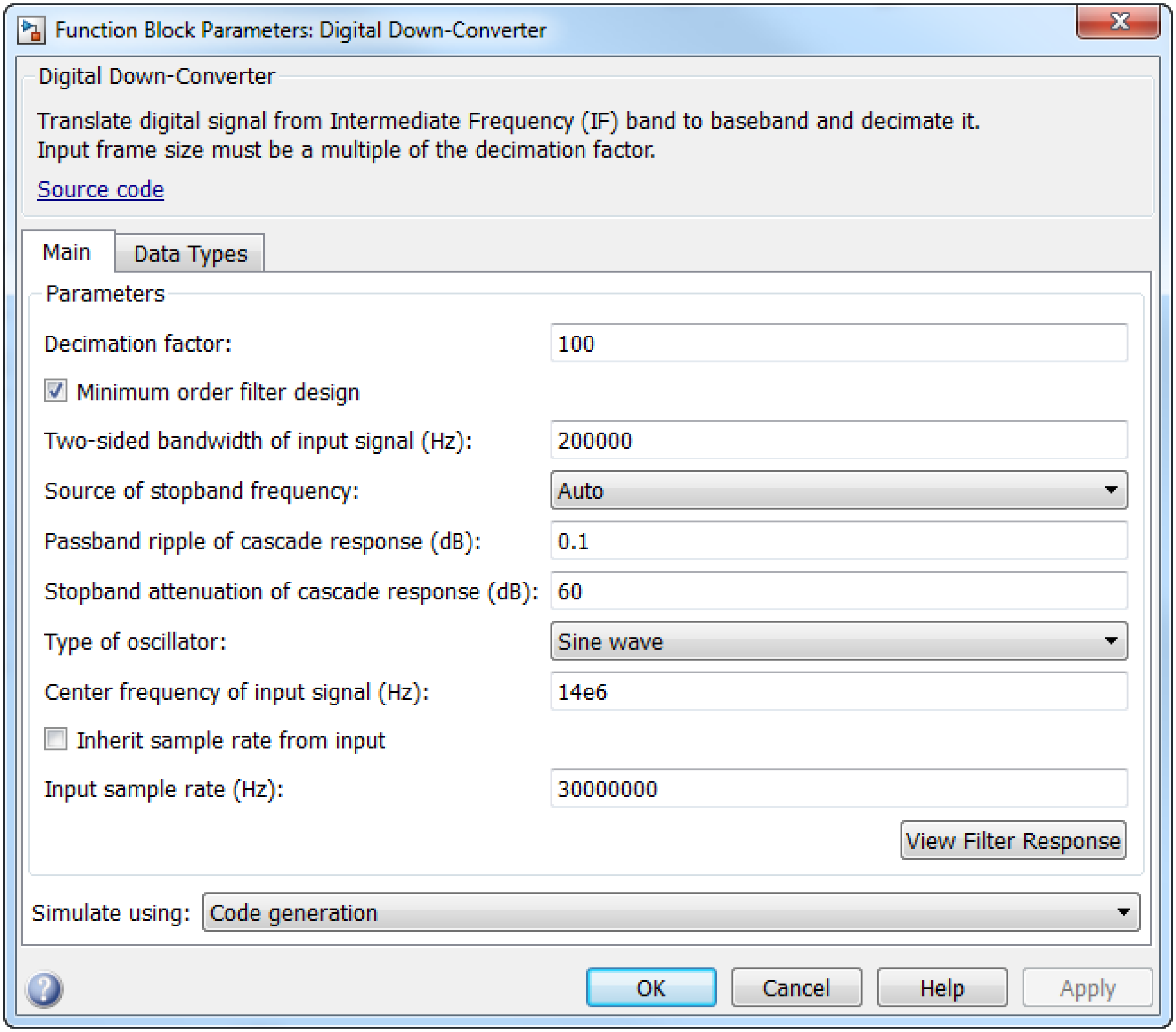
For more information on the structure that the DDC block uses, including the flow of fixed-point input, see the “Construction” on page 4-559 section in `dsp.DigitalDownConverter`.

Examples

- “Digital Up and Down Conversion for Family Radio Service”

Dialog Box

Main Tab



Decimation factor

Decimation factor, specified as a positive integer scalar, or as a 1-by-2 or 1-by-3 vector of positive integers. The default is 100.

When you set this parameter to a scalar, the block chooses the decimation factors for each of the three filtering stages.

When you set this parameter to a 1-by-2 vector, the block bypasses the third filter stage and sets the decimation factor of the first and second filtering stages to the values in the first and second vector elements, respectively. Both elements of **Decimation factor** must be greater than 1.

When you set this parameter to a 1-by-3 vector, the *i*th element of the vector specifies the decimation factor for the *i*th filtering stage. The first and second elements of **Decimation factor** must be greater than 1, and the third element must be 1 or 2.

Minimum order filter design

When you select this check box, the block designs filters with the minimum order that meets the requirements specified in these parameters:

- **Passband ripple of cascade response (dB)**
- **Stopband attenuation of cascade response (dB)**
- **Two sided bandwidth of input signal (Hz)**
- **Source of stopband frequency**
- **Stopband frequency (Hz)**

When you clear this check box, the block designs filters with orders that you specify in **Number of sections of CIC decimator**, **Order of CIC compensation filter stage**, and **Order of third filter stage**. The filter designs meet the passband and stopband frequency specifications that you set in **Two sided bandwidth of input signal (Hz)**, **Source of stopband frequency**, and **Stopband frequency (Hz)**. By default, this check box is selected.

Number of sections of CIC decimator

Number of sections in the CIC decimator, specified as a positive integer scalar. This parameter applies when you clear the **Minimum order filter design** check box. The default is 3.

Order of CIC compensation filter stage

Order of the CIC compensation filter stage, specified as a positive integer scalar. This parameter applies when you clear the **Minimum order filter design** check box. The default is 12.

Order of third filter stage

Order of the third filter stage, specified as an even positive integer scalar. When you specify **Decimation factor** as a 1-by-2 vector, the block ignores the value of **Order of third filter stage** because the block bypasses the third filter stage. This parameter applies when you clear the **Minimum order filter design** check box. The default is `10`.

Two sided bandwidth of input signal (Hz)

Two sided bandwidth of the input signal, specified as a positive integer scalar. The block sets the passband frequency of the cascade of filters to half the value that you specify in this parameter. Set the value of this parameter to less than **Input sample rate/Decimation factor**. When you select the **Inherit sample rate from input** check box, then set this value to less than ((1/*Ts*) / **Decimation factor**), where *Ts* is the sample time of the input signal. The default is `200` kHz.

Source of stopband frequency

Source of the stopband frequency, specified as `Auto` or `Property`. The default is `Auto`.

When you set this parameter to `Auto`, the block places the cutoff frequency of the cascade filter response at approximately $F_c = SampleRate / M/2$ Hz, where M is the total decimation factor specified in **Decimation factor**. *SampleRate* is computed as 1/ *Ts*, where *Ts* is the sample time of the input signal. The block computes the stopband frequency as $F_{stop} = F_c + (TW / 2)$. TW is the transition bandwidth of the cascade response, computed as $2\times(F_c - F_p)$, where the passband frequency, F_p, equals *Bandwidth*/2.

When you set this parameter to `Property`, specify the source in **Stopband frequency (Hz)**.

Stopband frequency (Hz)

Stopband frequency, specified as a double-precision positive scalar. This parameter applies when you set the **Source of stopband frequency** to `Property`. The default is `150` kHz.

Passband ripple of cascade response (dB)

Passband ripple of the cascade response, specified as a double-precision positive scalar. When you select the **Minimum order filter design**, the block designs the filters so that the cascade response meets the passband ripple that you specify in **Passband ripple of cascade response (dB)**. This parameter applies when you select the **Minimum order filter design**. The default is `0.1` dB.

Stopband attenuation of cascade response (dB)

Stopband attenuation of the cascade response, specified as a double-precision positive scalar. When you select the **Minimum order filter design**, the block designs the filters so that the cascade response meets the stopband attenuation that you specify in **Stopband attenuation of cascade response (dB)**. This parameter applies when you select the **Minimum order filter design**. The default is `60`.

Type of oscillator

Oscillator type, specified as one of the following:

- `Sine wave` (default) — The block performs frequency down conversion on input signal using a complex exponential obtained from samples of a sinusoidal trigonometric function.
- `NCO` — The block performs frequency down conversion on input signal with a complex exponential obtained using a numerically controlled oscillator (NCO).
- `Input port` — The block performs frequency down conversion on input signal using the complex signal that you provide through the input port of the block.
- `None` — The mixer stage in the block is not present and the block acts as three stage cascaded decimator.

Center frequency of input signal (Hz)

Center frequency of the input signal, specified as a double-precision positive scalar that is less than or equal to half the sample rate. The block downconverts the input signal from the passband center frequency, which you specify in **Center frequency of input signal (Hz)**, to `0` Hz. This parameter applies when you set **Type of oscillator** to `Sine wave` or `NCO`. The default is `14e6`.

Number of NCO accumulator bits

Number of NCO accumulator bits, specified as an integer scalar in the range `[1 128]`. This parameter applies when you set **Type of oscillator** to `NCO`. The default is `16`.

Number of NCO quantized accumulator

Number of NCO quantized accumulator bits, specified as an integer scalar in the range `[1 128]`. This value must be less than the value you specify in **Number of NCO accumulator bits**. This parameter applies when you set **Type of oscillator** to `NCO`. The default is `12`.

Dither control for NCO

When you select this check box, a number of dither bits specified in **Number of NCO dither bits** applies dither to the NCO signal. This parameter applies when you set **Type of oscillator** to `NCO`. By default, this check box is selected.

Number of NCO dither bits

Number of NCO dither bits, specified as an integer scalar smaller than the number of accumulator bits that you specify in **Number of NCO accumulator bits**. This parameter applies when you set **Type of oscillator** to NCO and select the **Dither control for NCO**. The default is 4.

Inherit sample rate from input

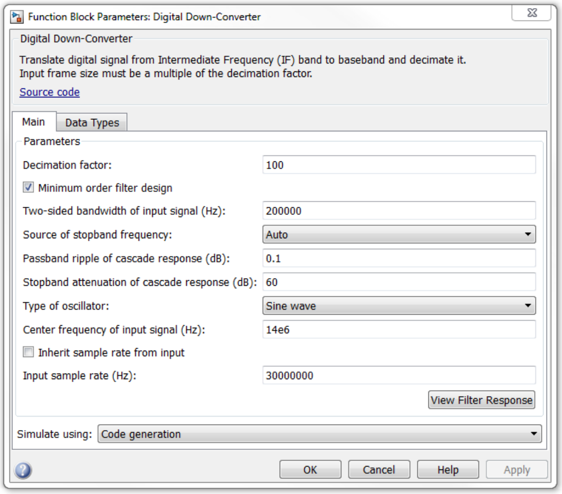
When you select this check box, sample rate is computed as N/ *Ts*, where *N* is the frame size of the input signal, and *Ts* is the sample time of the input signal. When you clear this check box, the block's sample rate is the value specified in **Input sample rate (Hz)**. By default, this check box is selected.

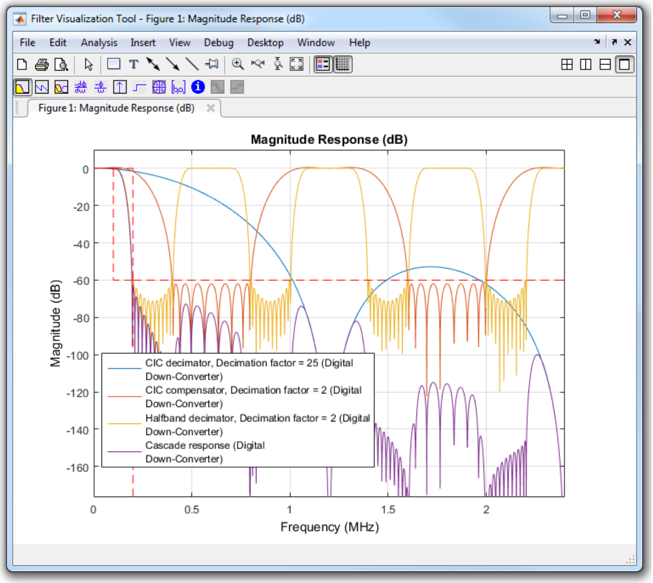
Input sample rate

Input sample rate, specified as a positive scalar value, greater than or equal to twice the value of the **Center frequency of input signal (Hz)**. The default is 30 MHz. This parameter applies when you clear the **Inherit sample rate from input** check box.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of each stage as well as the cascade of stages in the Digital Down-Converter. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.





To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

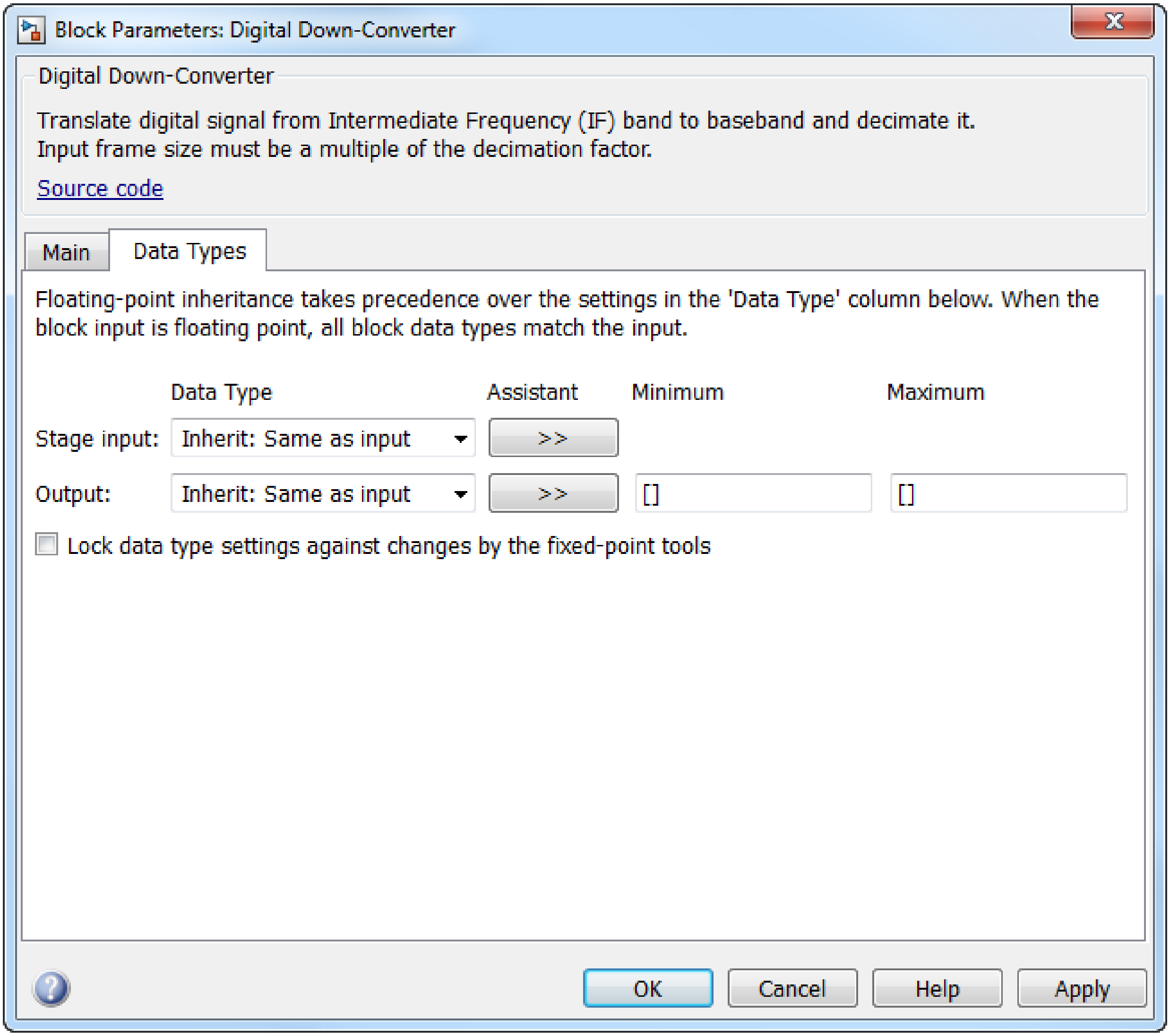
- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Data Types tab



Stage input

Data type of the input of the first, second, and third filter stages. You can set this parameter to:

- `Inherit: Same as input` (default) — The block inherits the **Stage input** from the input signal.
- `fixdt([],16,0)` — Fixed-point data type with binary point scaling. Specify the sign mode of this data type as `[]` or `true`.
- An expression that evaluates to a data type, for example, `numerictype([],16,15)`. Specify the sign mode of this data type as `[]` or `true`.

Click the **Show data type assistant** button 

to display the data type assistant, which helps you set the stage input parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output

Data type of the block output. You can set this parameter to:

- `Inherit: Same as input` (default) — The block Inherits the output datatype from the input.
- `fixdt([],16,0)` — Fixed-point data type with binary point scaling. Specify the sign mode of this data type as `[]` or `true`.
- An expression that evaluates to a data type, for example, `numerictype([],16,15)`. Specify the sign mode of this data type as `[]` or `true`.

Click the **Show data type assistant** button 

to display the data type assistant, which helps you set the **Output** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Minimum

Minimum value of the block output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Maximum

Maximum value of the block output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, 32- and 64-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, 32- and 64-bit signed integers

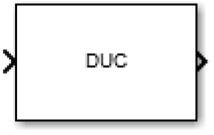
See Also

`Digital Up-Converter`	DSP System Toolbox
dsp.DigitalDownConverter	DSP System Toolbox
dsp.DigitalUpConverter	DSP System Toolbox

Introduced in R2015a

Digital Up-Converter

Interpolate digital signal and translate it from baseband to Intermediate Frequency (IF) band



Library

Signal Operations

`dspsigops`

Description

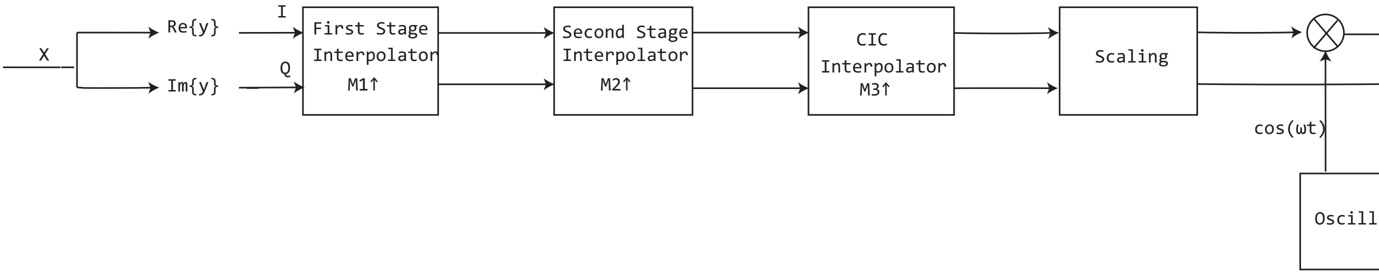
The Digital Up-Converter (DUC) block converts a complex digital baseband signal to a real passband signal.

The DUC block upsamples the input signal using a cascade of three interpolation filters. The block frequency upconverts the upsampled signal by multiplying it by the specified center frequency of the output signal. This block designs the interpolation filters according to the filter parameters that you set in the block dialog.

Structure

This block brings the capabilities of dsp.DigitalUpConverter System object to the Simulink environment.

The DUC block consists of a FIR interpolator, a CIC compensator, and a CIC interpolator. You can bypass the FIR interpolator, depending on how you set the DUC block parameters.



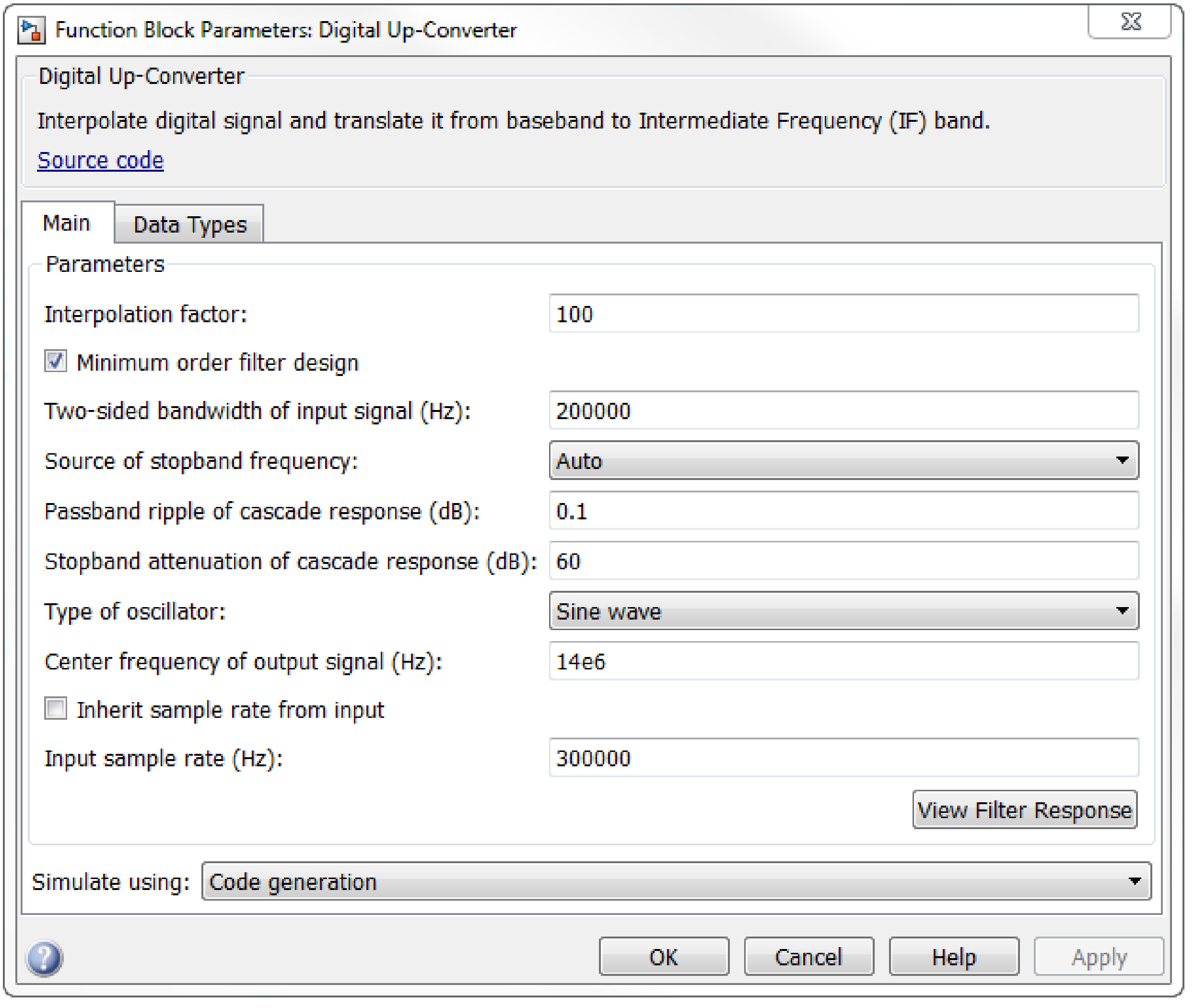
For more information on the structure that the DUC block uses, including the flow of fixed-point input, see the Construction on page 4-597 section in `dsp.DigitalUpConverter`.

Examples

- "Digital Up and Down Conversion for Family Radio Service"

Dialog Box

Main Tab



Interpolation factor

Interpolation factor, specified as a positive integer scalar, or as a 1-by-2 or 1-by-3 vector of positive integers. The default is 100.

When you set this parameter to a scalar, the block chooses the interpolation factors for each of the three filtering stages.

When you set this parameter to a 1-by-2 vector, the block bypasses the first filter stage and sets the interpolation factor of the second and third filtering stages to the values in the first and second vector elements, respectively. Both elements of the **Interpolation factor** must be greater than 1.

When you set this parameter to a 1-by-3 vector, the *i*th element of the vector specifies the interpolation factor for the *i*th filtering stage. The second and third elements of **Interpolation factor** must be greater than 1, and the first element must be 1 or 2.

Minimum order filter design

When you select this check box, the block designs filters with the minimum order that meets the requirements specified in these parameters:

- **Passband ripple of cascade response (dB)**
- **Stopband attenuation of cascade response (dB)**
- **Two sided bandwidth of input signal (Hz)**
- **Source of stopband frequency**
- **Stopband frequency (Hz)**

When you clear this check box, the block designs filters with orders that you specify in **Order of first filter stage**, **Order of CIC compensation filter stage**, and **Number of sections of CIC interpolator**. The filter designs meet the passband and stopband frequency specifications that you set in **Two sided bandwidth of input signal (Hz)**, **Source of stopband frequency**, and **Stopband frequency (Hz)**. By default, this check box is selected.

Order of first filter stage

Order of the first filter stage, specified as an even positive integer scalar. When you specify **Interpolation factor** as a 1-by-2 vector, the block ignores the value of **Order of first filter stage** because the block bypasses the first filter stage. This parameter applies when you clear the **Minimum order filter design** check box. The default is 10.

Order of CIC compensation filter stage

Order of the CIC compensation filter stage, specified as a positive integer scalar. This parameter applies when you clear the **Minimum order filter design** check box. The default is `12`.

Number of sections of CIC interpolator

Number of sections in the CIC interpolator, specified as a positive integer scalar. This parameter applies when you clear the **Minimum order filter design** check box. The default is `3`.

Two sided bandwidth of input signal (Hz)

Two sided bandwidth of the input signal, specified as a positive integer scalar. The block sets the passband frequency of the cascade of filters to half the value that you specify in this parameter. The default is 200 kHz.

Source of stopband frequency

Source of the stopband frequency, specified as `Auto` or `Property`. The default is `Auto`.

When you set this parameter to `Auto`, the block places the cutoff frequency of the cascade filter response at approximately $F_c = SampleRate/2$ Hz, and computes the stopband frequency as $F_{stop} = F_c + TW/2$. *SampleRate* is computed as `1/` *Ts*, where *Ts* is the sample time of the input signal. *TW* is the transition bandwidth of the cascade response, computed as $2\times(F_c - F_p)$, and the passband frequency, F_p, equals *Bandwidth*/2.

When you set this parameter to `Property`, specify the source in **Stopband frequency (Hz)**.

Stopband frequency (Hz)

Stopband frequency, specified as a double-precision positive scalar. This parameter applies when you set the **Source of stopband frequency** to `Property`. The default is `150` kHz.

Passband ripple of cascade response (dB)

Passband ripple of the cascade response, specified as a double-precision positive scalar. When you select the **Minimum order filter design**, the block designs the filters so that the cascade response meets the passband ripple that you specify in **Passband ripple of cascade response (dB)**. This parameter applies when you select the **Minimum order filter design** check box. The default is `0.1` dB.

Stopband attenuation of cascade response (dB)

Stopband attenuation of the cascade response, specified as a double-precision positive scalar. When you select the **Minimum order filter design** check box, the block designs the filters so that the cascade response meets the stopband attenuation that you specify in **Stopband attenuation of cascade response (dB)**. This parameter applies when you select the **Minimum order filter design** check box. The default is `60` dB.

Type of oscillator

Oscillator type, specified as one of the following:

- `Sine wave` (default) — The block frequency upconverts the output of the interpolation filter cascade using a complex exponential signal obtained from samples of a sinusoidal trigonometric function.
- `NCO` — The block performs frequency up conversion with a complex exponential obtained using a numerically controlled oscillator (NCO).

Center frequency of output signal (Hz)

Center frequency of the output signal, specified as a double-precision positive scalar. The value of this parameter must be less than or equal to half the product of the *SampleRate* times the total interpolation factor. *SampleRate* is computed as `1/` *Ts*, where *Ts* is the sample time of the input signal. The block up converts the input signal so that the output spectrum centers at the frequency you specify in **Center frequency of output signal (Hz)**. The default is `14` MHz.

Number of NCO accumulator bits

Number of NCO accumulator bits, specified as an integer scalar in the range `[1 128]`. This parameter applies when you set **Type of oscillator** to `NCO`. The default is `16`.

Number of NCO quantized accumulator

Number of NCO quantized accumulator bits, specified as an integer scalar in the range `[1 128]`. This value must be less than the value you specify in **Number of NCO accumulator bits**. This parameter applies when you set **Type of oscillator** to `NCO`. The default is `12`.

Dither control for NCO

When you select this check box, a number of dither bits specified in **Number of NCO dither bits** applies dither to the NCO signal. This parameter applies when you set **Type of oscillator** to `NCO`. By default, this check box is selected.

Number of NCO dither bits

Number of NCO dither bits, specified as an integer scalar smaller than the number of accumulator bits that you specify in **Number of NCO accumulator bits**. This parameter applies when you set **Type of oscillator** to NCO and select the **Dither control for NCO**. The default is 4.

Inherit sample rate from input

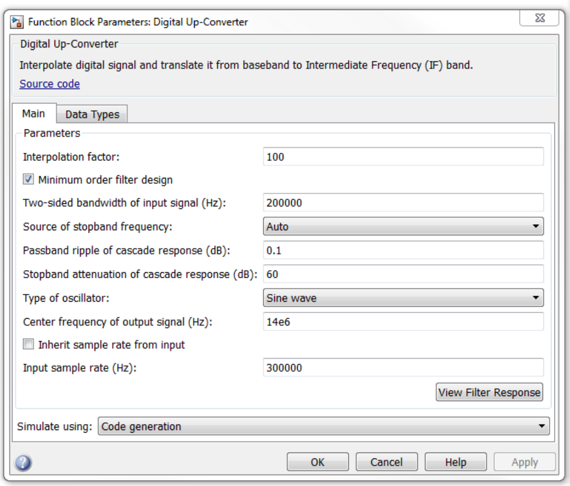
When you select this check box, sample rate is computed as N/ *Ts*, where *N* is the frame size of the input signal, and *Ts* is the sample time of the input signal. When you clear this check box, the block's sample rate is the value specified in **Input sample rate (Hz)**. By default, this check box is selected.

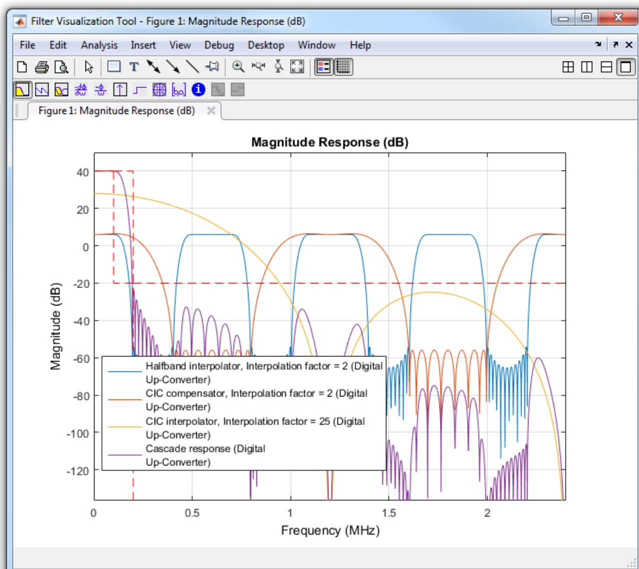
Input sample rate

Input sample rate, specified as a positive scalar. The value of this parameter multiplied by the total interpolation factor must be greater than or equal to twice the value of the **Center frequency of output signal (Hz)**. The default is 30 MHz. This parameter applies when you clear the **Inherit sample rate from input** check box.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of each stage as well as the cascade of stages in the Digital Up-Converter. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.





To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

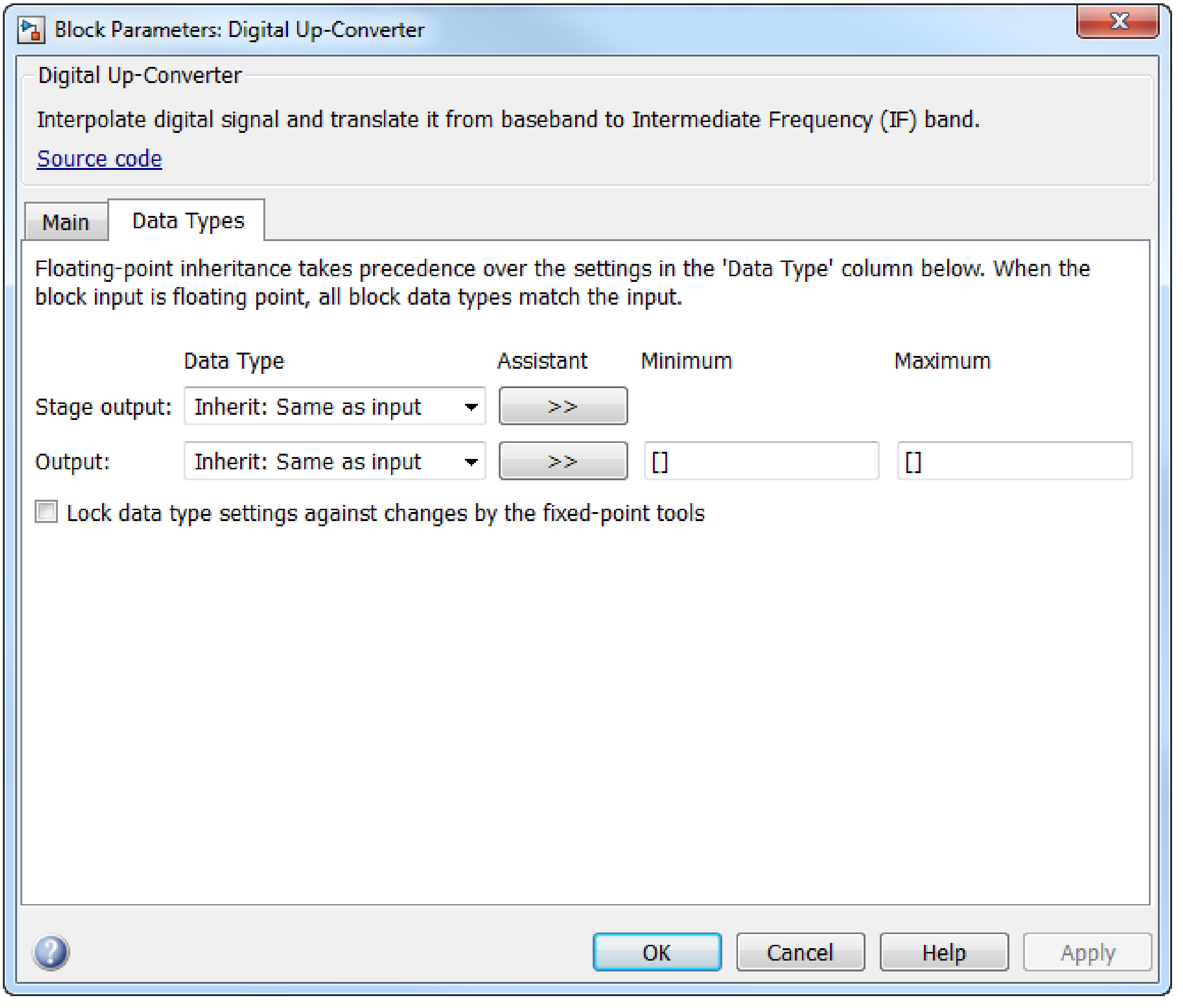
Type of simulation to run. You can set this parameter to:

- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Data Types Tab



Stage output

Data type of the output of the first, second, and third filter stages. You can set this parameter to:

- `Inherit: Same as input` (default) — The block inherits the **Stage output** from the input signal.
- `fixdt([],16,0)` — Fixed-point data type with binary point scaling. Specify the sign mode of this data type as `[]` or `true`.
- An expression that evaluates to a data type, for example, `numerictype([],16,15)`. Specify the sign mode of this data type as `[]` or `true`.

The block casts the data at the output of each filter stage according to the value you set in this parameter. For the CIC stage, the casting is done after the signal has been scaled by the normalization factor.

Click the **Show data type assistant** button  to display the data type assistant, which helps you set the stage output parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output

Data type of the block output. You can set this parameter to:

- `Inherit: Same as input` (default) — The block Inherits the output datatype from the input.
- `fixdt([],16,0)` — Fixed-point data type with binary point scaling. Specify the sign mode of this data type as `[]` or `true`.
- An expression that evaluates to a data type, for example, `numerictype([],16,15)`. Specify the sign mode of this data type as `[]` or `true`.

Click the **Show data type assistant** button  to display the data type assistant, which helps you set the **Output** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Minimum

Minimum value of the block output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Maximum

Maximum value of the block output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, 32- and 64-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, 32- and 64-bit signed integers

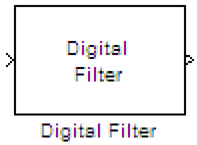
See Also

`Digital Down-Converter`	DSP System Toolbox
dsp.DigitalDownConverter	DSP System Toolbox
dsp.DigitalUpConverter	DSP System Toolbox

Introduced in R2015a

Digital Filter (Obsolete)

Filter each channel of input over time using static or time-varying digital filter implementations



Library

Filtering / Filter Implementations

`dsparch4`

Description

Note: Use of Digital Filter block in future releases is not recommended. Existing instances will continue to operate, but certain functionality will be disabled. See "Functionality being removed or replaced for blocks and System objects". We strongly recommend using one of `Discrete FIR Filter`, Discrete Filter, `Biquad Filter`, or `Allpole Filter` in new designs.

You can use the Digital Filter block to efficiently implement a floating-point or fixed-point filter for which you know the coefficients, or that is already defined in a `dfilt` object. The block independently filters each channel of the input signal with a specified digital IIR or FIR filter. The block can implement *static filters* with fixed coefficients, as well as *time-varying filters* with coefficients that change over time. You can tune the coefficients of a static filter during simulation.

This block filters each channel of the input signal independently over time. You must set the **Input processing** parameter to specify how the block interprets the input signal. You can select one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as an independent channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as an individual channel.

The output dimensions always match those of the input signal. The outputs of this block numerically match the outputs of the `Digital Filter Design` block and of the `dfilt` object.

Note: The Digital Filter block has direct feedthrough, so if you connect the output of this block back to its input you get an algebraic loop. For more information on direct feedthrough and algebraic loops, see "Algebraic Loops" (Simulink).

Sections of This Reference Page

- "Coefficient Source" on page 2-527
- "Supported Filter Structures" on page 2-528
- "Specifying Initial Conditions" on page 2-530
- "State Logging" on page 2-534
- "Fixed-Point Data Types" on page 2-534
- "Dialog Box" on page 2-535
- "Filter Structure Diagrams" on page 2-549
- "Supported Data Types" on page 2-582
- "See Also" on page 2-582

Coefficient Source

The Digital Filter block can operate in three different modes. Select the mode in the **Coefficient source** group box.

- **Dialog parameters** Enter information about the filter such as structure and coefficients in the block mask.
- **Input port(s)** Enter the filter structure in the block mask, and the filter coefficients come in through one or more block ports. This mode is useful for specifying time-varying filters.
- **Discrete-time filter object (DFILT)** Specify the filter using a `dfilt` object.

Supported Filter Structures

When you select **Discrete-time filter object (DFILT)**, the following `dfilt` structures are supported:

- `dfilt.df1`
- `dfilt.df1t`
- `dfilt.df2`
- `dfilt.df2t`
- `dfilt.df1sos`
- `dfilt.df1tsos`
- `dfilt.df2sos`
- `dfilt.df2tsos`
- `dfilt.dffir`
- `dfilt.dffirt`
- `dfilt.dfsymfir`
- `dfilt.dfasymfir`
- `dfilt.latticear`
- `dfilt.latticemamin`

When you select **Dialog parameters** or **Input port(s)**, the list of filter structures offered in the **Filter structure** parameter depends on whether you set the **Transfer function type** to `IIR (poles & zeros)`, `IIR (all poles)`, or `FIR (all zeros)`, as summarized in the following table.

Note Each structure listed in the table below supports both fixed-point and floating-point signals.

The table also shows the vector or matrix of filter coefficients you must provide for each filter structure.

Filter Structures and Filter Coefficients

Transfer Function Type	Supported Filter Structures	Filter Coefficient Specification

IIR (poles & zeros)	`Direct form I`
	`Direct form I transposed`
	`Direct form II`
	`Direct form II transposed`
		• Numerator coefficients vector [b0, b1, b2, ..., bn]
		• Denominator coefficients vector [a0, a1, a2, ..., am]
		See Special Consideration for the Leading Denominator Coefficient on page 2-530.

	`Biquadratic direct form I (SOS)`
	`Biquadratic direct form I transposed (SOS)`
	`Biquadratic direct form II (SOS)`
	`Biquadratic direct form II transposed (SOS)`
		• *M*-by-6 second-order section (SOS) matrix.
		• Scale values

IIR (all poles)	`Direct form` `Direct form transposed`	Denominator coefficients vector [a0, a1, a2, ..., am] See Special Consideration for the Leading Denominator Coefficient on page 2-530.
	`Lattice AR`	Reflection coefficients vector [k1, k2, ..., kn]

FIR (all zeros)	`Direct form`
	`Direct form symmetric`
	`Direct form antisymmetric`
	`Direct form transposed`
		Numerator coefficients vector [b0, b1, b2, ..., bn]

	`Lattice MA`	Reflection coefficients vector [k1, k2, ..., kn]

Special Considerations for the Leading Denominator Coefficient

In some cases, the Digital Filter block requires the leading denominator coefficient (a_0) to be 1. This requirement applies under the following conditions:

- The Digital Filter block is operating in a fixed-point mode. The block operates in a fixed-point mode when at least one of the following statements is true:
  - The input to the Digital Filter block has a fixed-point or integer data type.
  - The **Fixed-point instrumentation mode** parameter under **Analysis > Fixed Point Tool** has a setting of `Minimums, maximums and overflows`.
- The **Coefficient source** has a setting of `Dialog` or `Input port(s)`.

Note: If you are working in one of the fixed-point situations described in the previous bullet, and the **Coefficient source** is set to `Input port(s)`, you must select the **First denominator coefficient = 1, remove a0 term in the structure** check box.

- The **Transfer function type** and **Filter structure** parameters are set to one of the combinations described in the following table.

Transfer function type	Filter structure
IIR (poles & zeros)	`Direct form I`
	`Direct form I transposed`
	`Direct form II`
	`Direct form II transposed`
IIR (all poles)	`Direct form`
	`Direct form transposed`

The Digital Filter block produces an error if you use it in one of the these configurations and your leading denominator coefficient (a_0) does not equal 1. To resolve the error, set your leading denominator coefficient to 1 by scaling all numerator and denominator coefficients by a factor of a_0.

Specifying Initial Conditions

In **Dialog parameters** and **Input port(s)** modes, the block initializes the internal filter states to zero by default, which is equivalent to assuming past inputs and outputs are zero. You can optionally use the **Initial conditions** parameter to specify nonzero initial conditions for the filter delays.

To determine the number of initial condition values you must specify, and how to specify them, see the following table on Valid Initial Conditions and Number of Delay Elements (Filter States). The **Initial conditions** parameter can take one of four forms as described in the following table.

Valid Initial Conditions

Initial Condition	Examples	Description
Scalar	`5` Each delay element for each channel is set to `5`.	The block initializes all delay elements in the filter to the scalar value.
Vector (for applying the same delay elements to each channel)	For a filter with two delay elements: [d_1 d_2] The delay elements for all channels are d1 and d2.	Each vector element specifies a unique initial condition for a corresponding delay element. The block applies the same vector of initial conditions to each channel of the input signal. The vector length must equal the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States)).
Vector or matrix (for applying different delay elements to each channel)	For a 3-channel input signal and a filter with two delay elements: [d_1 d_2 D_1 D_2 d_1 d_2] or $\begin{bmatrix} d_1 & D_1 & d_1 \\ d_2 & D_2 & d_2 \end{bmatrix}$ • The delay elements for channel 1 are d_1 and d_2. • The delay elements for channel 2 are D_1 and D_2. • The delay elements for channel 3 are $d_1$and d_2.	Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel: • The vector length must be equal to the product of the number of input channels and the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States)). • The matrix must have the same number of rows as the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States)), and must have one column for each channel of the input signal.

Initial Condition	Examples	Description
Empty matrix	`[ ]` Each delay element for each channel is set to `0`.	The empty matrix, `[ ]`, is equivalent to setting the **Initial conditions** parameter to the scalar value `0`.

The number of delay elements (filter states) per input channel depends on the filter structure, as indicated in the following table.

Number of Delay Elements (Filter States)

Filter Structure	Number of Delay Elements per Channel

`Direct form`
`Direct form transposed`
`Direct form symmetric`
`Direct form antisymmetric`
	`#_of_filter_coeffs-1`

`Direct form I` `Direct form I transposed`	• `#_of_zeros-1` • `#_of_poles-1`
`Direct form II` `Direct form II transposed`	`max(#_of_zeros, #_of_poles)-1`

`Biquadratic direct form I (SOS)`
`Biquadratic direct form I transposed (SOS)`
`Biquadratic direct form II (SOS)`
`Biquadratic direct form II transposed (SOS)`
	`2 * #_of_filter_sections`

`Lattice AR` `Lattice MA`	`#_of_reflection_coeffs`

State Logging

Simulink enables you to log the states in your model to the MATLAB workspace. The following table indicates which filter structures of the Digital Filter block support the Simulink state logging feature. See "States" (Simulink) for more information.

Transfer Function Type	Filter Structure	State Logging Supported
IIR (poles & zeros)	`Direct form I`	No
	`Direct form I transposed`	Yes
	`Direct form II`	No
	`Direct form II transposed`	Yes
	`Biquadratic direct form I (SOS)`	Yes
	`Biquadratic direct form I transposed (SOS)`	Yes
	`Biquadratic direct form II (SOS)`	Yes
	`Biquadratic direct form II transposed (SOS)`	Yes
IIR (all poles)	`Direct form`	No
	`Direct form transposed`	Yes
	`Lattice AR`	Yes
FIR (all zeros)	`Direct form`	No
	`Direct form symmetric`	No
	`Direct form antisymmetric`	No
	`Direct form transposed`	Yes
	`Lattice MA`	Yes

Fixed-Point Data Types

All structures supported by the Digital Filter block support fixed-point data types. You can specify intermediate fixed-point data types for quantities such as the coefficients, accumulator, and product output for each filter structure. See "Filter Structure

Diagrams" on page 2-549 for diagrams depicting the use of these intermediate fixed-point data types in each filter structure.

Dialog Box

Coefficient Source

The Digital Filter block can operate in three different modes. Select the mode in the **Coefficient source** group box.

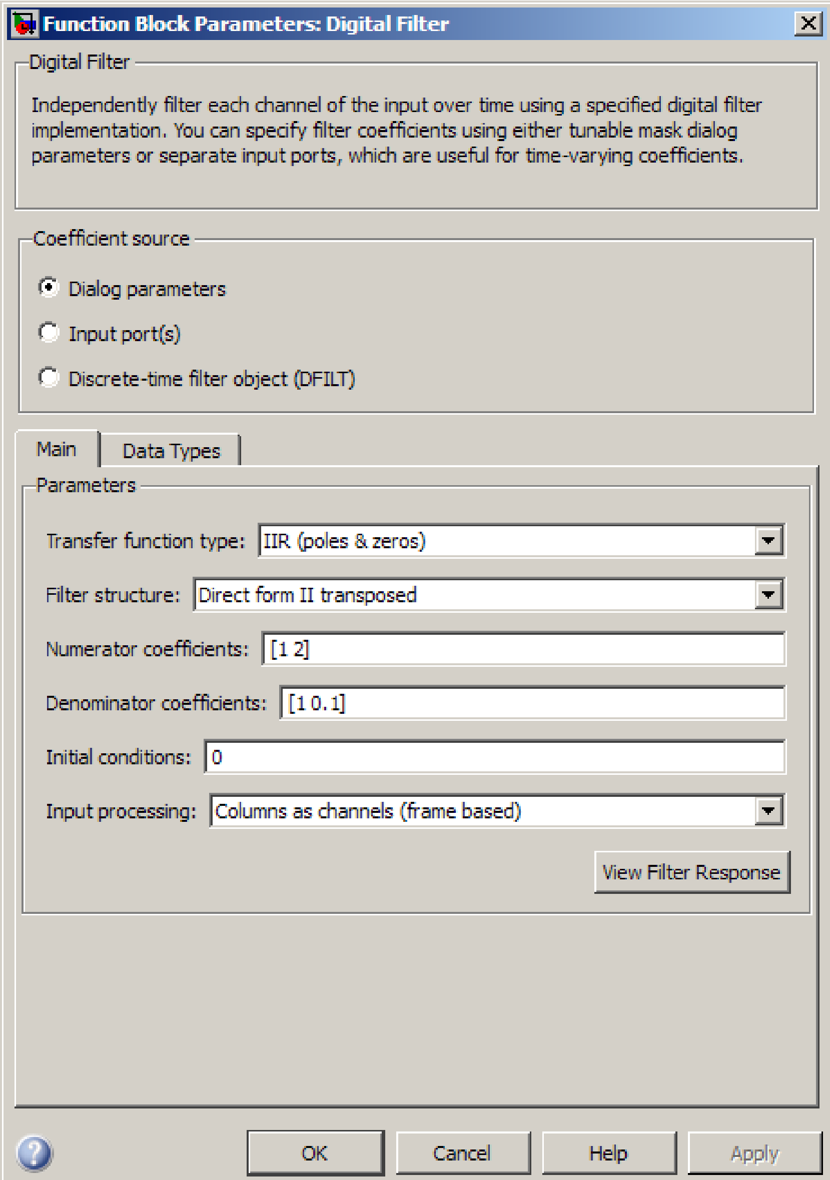
- **Dialog parameters** Enter information about the filter such as structure and coefficients in the block mask.
- **Input port(s)** Enter the filter structure in the block mask, and the filter coefficients come in through one or more block ports. This mode is useful for specifying time-varying filters.
- **Discrete-time filter object (DFILT)** Specify the filter using a `dfilt` object.

Different items appear on the Digital Filter block dialog depending on whether you select **Dialog parameters**, **Input port(s)**, or **Discrete-time filter object (DFILT)** in the **Coefficient source** group box. See the following sections for details:

- "Specify Filter Characteristics in Dialog and/or Through Input Ports" on page 2-535
- "Specify Discrete-Time Filter Object" on page 2-545

Specify Filter Characteristics in Dialog and/or Through Input Ports

The **Main** pane of the Digital Filter block dialog appears as follows when **Dialog parameters** is specified in the **Coefficient source** group box. The parameters below can appear when **Dialog parameters** or **Input port(s)** is selected, as noted.



Transfer function type

Select the type of transfer function of the filter; `IIR (poles & zeros)`, `IIR (all poles)`, or `FIR (all zeros)`. See "Supported Filter Structures" on page 2-528 for more information.

Filter structure

Select the filter structure. The selection of available structures varies depending the setting of the **Transfer function type** parameter. See "Supported Filter Structures" on page 2-528 for more information.

Numerator coefficients

Specify the vector of numerator coefficients of the filter's transfer function.

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure lends itself to specification with numerator coefficients. Tunable (Simulink).

Denominator coefficients

Specify the vector of denominator coefficients of the filter's transfer function.

In some cases, the leading denominator coefficient (a0) must be 1. See Special Consideration for the Leading Denominator Coefficient on page 2-530 for more information.

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure lends itself to specification with denominator coefficients. Tunable (Simulink).

Reflection coefficients

Specify the vector of reflection coefficients of the filter's transfer function.

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure lends itself to specification with reflection coefficients. Tunable (Simulink).

SOS matrix (Mx6)

Specify an *M*-by-6 *SOS matrix* containing coefficients of a second-order section (SOS) filter, where *M* is the number of sections. You can use the `ss2sos` and `tf2sos` functions from Signal Processing Toolbox software to check whether your SOS matrix is valid.

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure is biquadratic. Tunable (Simulink).

Scale values

Specify the scale values to be applied before and after each section of a biquadratic filter.

- If you specify a scalar, that value is applied before the first filter section. The rest of the scale values are set to 1.
- You can also specify a vector with $M + 1$ elements, assigning a different value to each scale. See “Filter Structure Diagrams” on page 2-549 for diagrams depicting the use of scale values in biquadratic filter structures.

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure is biquadratic. Tunable (Simulink).

First denominator coefficient = 1, remove a0 term in the structure

Select this parameter to reduce the number of computations the block must make to produce the output by omitting the `1 / a`$_0$ term in the filter structure. The block output is invalid if you select this parameter when the first denominator filter coefficient is *not* always 1 for your time-varying filter.

This parameter is only enabled when the **Input port(s)** is selected *and* when the selected filter structure lends itself to this specification.

Coefficient update rate

Specify how often the block updates time-varying filters; once per sample or once per frame.

This parameter appears only when the following conditions are met:

- You specify **Input port(s)** in the Coefficient source group box.
- You set the **Input processing** parameter to `Columns as channels (frame based)`.

Initial conditions

Specify the initial conditions of the filter states. To learn how to specify initial conditions, see “Specifying Initial Conditions” on page 2-530.

Initial conditions on zeros side

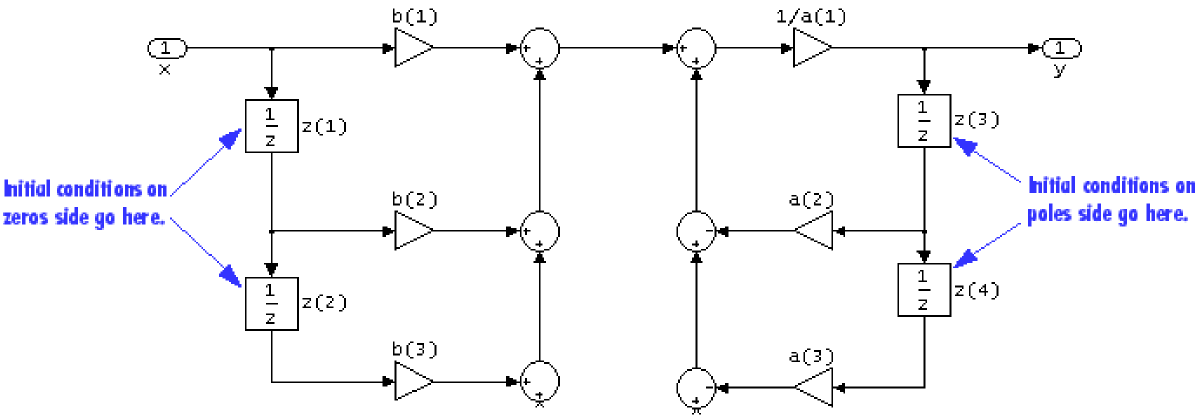
(Not shown in dialog above.) Specify the initial conditions for the filter states on the side of the filter structure with the zeros (b_0, b_1,b_2, ...); see the diagram below.

This parameter is enabled only when the filter has both poles and zeros, *and* when you select a structure such as direct form I, which has separate filter states corresponding to the poles (a_k) and zeros (b_k). To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 2-530.

Initial conditions on poles side

(Not shown in dialog above). Specify the initial conditions for the filter states on the side of the filter structure with the poles (a_0, a_1,a_2, ...); see the diagram below.

This parameter is enabled only when the filter has both poles and zeros, *and* when you select a structure such as direct form I, which has separate filter states corresponding to the poles (a_k) and zeros (b_k). To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 2-530.



Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

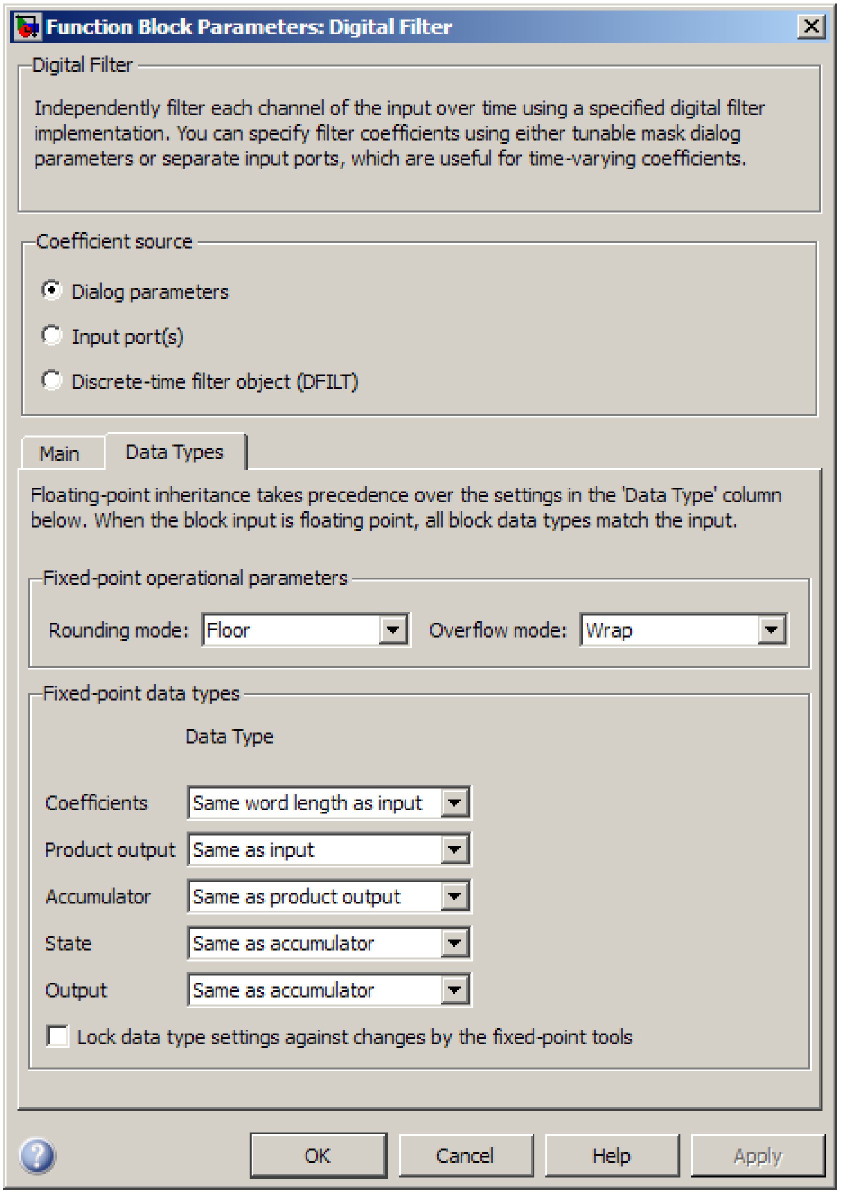
Note: The `Inherited (this choice will be removed - see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the filter defined by the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note: If you specify a filter in the **Filter** parameter, you must click the **Apply** button to apply the filter before using the **View filter response** button.

The **Data Types** pane of the Digital Filter block dialog appears as follows when **Dialog parameters** is specified in the **Coefficient source** group box. The parameters below can appear when **Dialog parameters** or **Input port(s)** is selected, depending on the filter structure and whether the coefficients are being entered via ports or on the block mask.



Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Section I/O

Choose how you specify the word length and the fraction length of the fixed-point data type going into and coming out of each section of a biquadratic filter. See "Filter Structure Diagrams" on page 2-549 for illustrations depicting the use of the section I/O data type in this block.

This parameter is only visible when the selected filter structure is biquadratic:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word and fraction lengths of the section input and output, in bits.
- When you select `Slope and bias scaling`, you can enter the word lengths, in bits, and the slopes of the section input and output. This block requires power-of-two slope and a bias of zero.

Tap sum

Choose how you specify the word length and the fraction length of the tap sum data type of a direct form symmetric or direct form antisymmetric filter. See "Filter Structure Diagrams" on page 2-549 for illustrations depicting the use of the tap sum data type in this block.

This parameter is only visible when the selected filter structure is either `Direct form symmetric` or `Direct form antisymmetric`:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the tap sum accumulator, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the tap sum accumulator. This block requires power-of-two slope and a bias of zero.

Multiplicand

Choose how you specify the word length and the fraction length of the multiplicand data type of a direct form I transposed or biquadratic direct form I transposed filter. See “Filter Structure Diagrams” on page 2-549 for illustrations depicting the use of the multiplicand data type in this block.

This parameter is only visible when the selected filter structure is either `Direct form I transposed` or `Biquad direct form I transposed (SOS)`:

- When you select `Same as output`, these characteristics match those of the output to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the multiplicand data type, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the multiplicand data type. This block requires power-of-two slope and a bias of zero.

Coefficients

Choose how you specify the word length and the fraction length of the filter coefficients (numerator and/or denominator). See “Filter Structure Diagrams” on page 2-549 for illustrations depicting the use of the coefficient data types in this block:

- When you select `Same word length as input`, the word length of the filter coefficients match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Specify word length`, you can enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the coefficients, in bits. If applicable, you can enter separate fraction lengths for the numerator and denominator coefficients.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the coefficients. If applicable, you can enter separate slopes for the numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.

- The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See "Filter Structure Diagrams" on page 2-549 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the product output, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See "Filter Structure Diagrams" on page 2-549 and "Multiplication Data Types" for illustrations depicting the use of the accumulator data type in this block:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Same as product output`, these characteristics match those of the product output.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

State

Use this parameter to specify how you would like to designate the state word and fraction lengths. See "Filter Structure Diagrams" on page 2-549 for illustrations depicting the use of the state data type in this block.

This parameter is not visible for direct form and direct form I filter structures.

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Same as accumulator`, these characteristics match those of the accumulator.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you specify the output word length and fraction length:

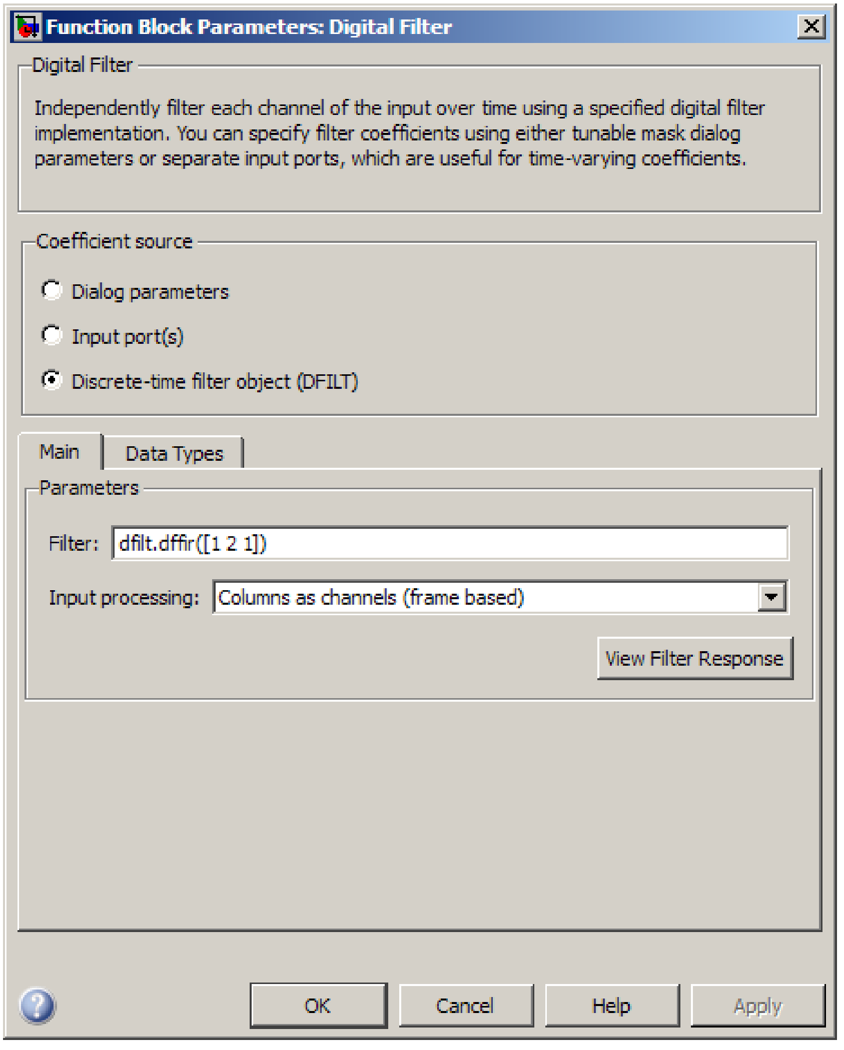
- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Same as accumulator`, these characteristics match those of the accumulator.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the output, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Specify Discrete-Time Filter Object

The **Main** pane of the Digital Filter block dialog appears as follows when **Discrete-time filter object (DFILT)** is specified in the **Coefficient source** group box.



Filter

Specify the discrete-time filter object (`dfilt`) that you would like the block to implement. You can do this in one of three ways:

- You can fully specify the `dfilt` object in the block mask, as shown in the default value.

- You can enter the variable name of a `dfilt` object that is defined in any workspace.
- You can enter a variable name for a `dfilt` object that is not yet defined.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

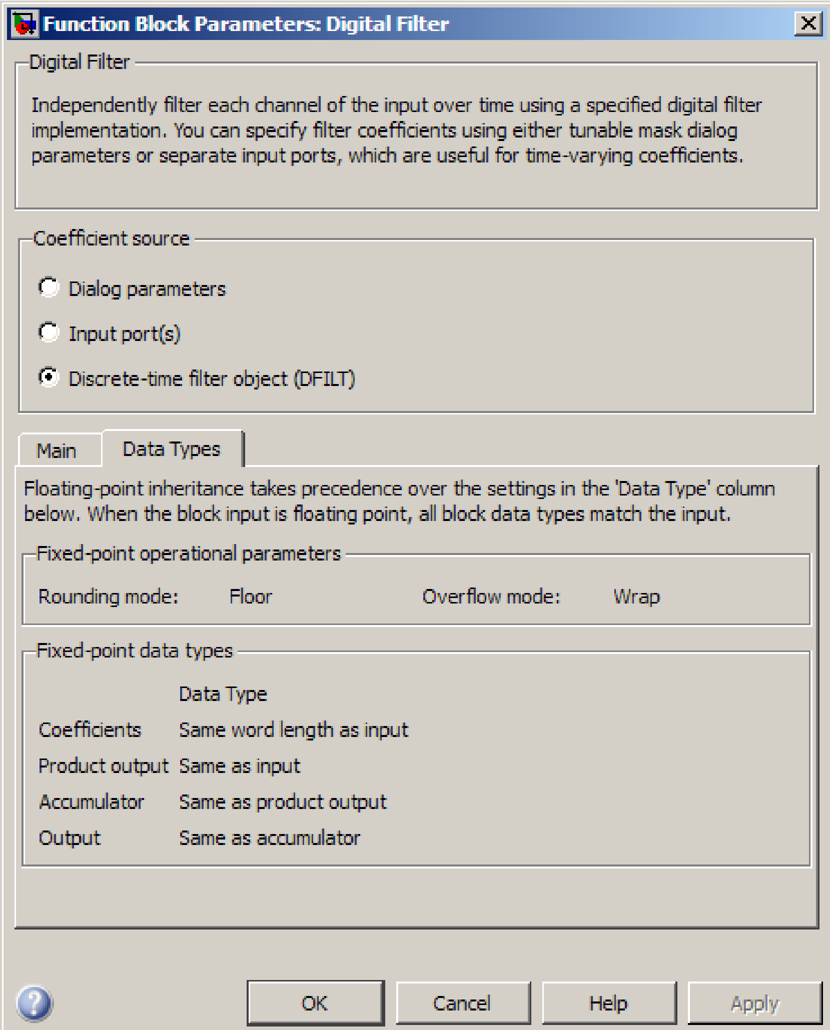
Note: The `Inherited (this choice will be removed - see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the `dfilt` object specified in the **Filter** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note: If you specify a filter in the **Filter** parameter, you must click the **Apply** button to apply the filter before using the **View filter response** button.

The **Data Types** pane of the Digital Filter block dialog appears as follows when **Discrete-time filter object (DFILT)** is specified in the **Coefficient source** group box.



The fixed-point settings of the filter object specified on the **Main** pane are displayed on the **Data Types** pane. You cannot change these settings directly on the block mask. To change the fixed-point settings you must edit the filter object directly.

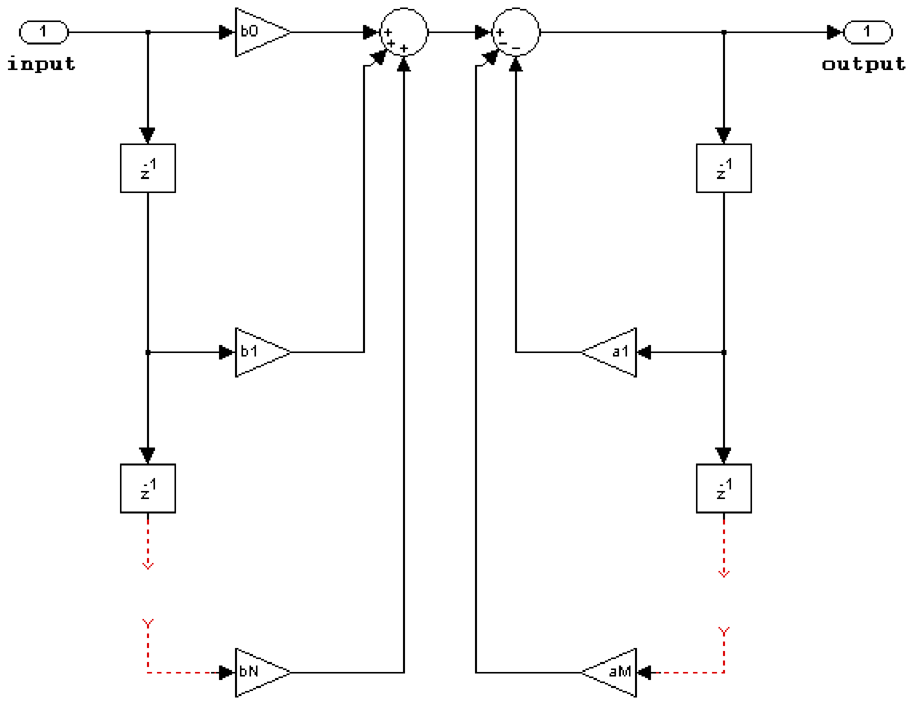
For more information on discrete-time filter objects, see the `dfilt` reference page.

Filter Structure Diagrams

The diagrams in the following sections show the filter structures supported by the Digital Filter block. They also show the data types used in the filter structures for fixed-point signals. You can set the coefficient, output, accumulator, product output, and state data types shown in these diagrams in the block dialog. This is discussed in "Dialog Box" on page 2-535.

- "IIR direct form I" on page 2-550
- "IIR direct form I transposed" on page 2-552
- "IIR direct form II" on page 2-554
- "IIR direct form II transposed" on page 2-556
- "IIR biquadratic direct form I" on page 2-558
- "IIR biquadratic direct form I transposed" on page 2-560
- "IIR biquadratic direct form II" on page 2-563
- "IIR biquadratic direct form II transposed" on page 2-565
- "IIR (all poles) direct form" on page 2-568
- "IIR (all poles) direct form transposed" on page 2-570
- "IIR (all poles) direct form lattice AR" on page 2-572
- "FIR (all zeros) direct form" on page 2-573
- "FIR (all zeros) direct form symmetric" on page 2-575
- "FIR (all zeros) direct form antisymmetric" on page 2-577
- "FIR (all zeros) direct form transposed" on page 2-579
- "FIR (all zeros) lattice MA" on page 2-581

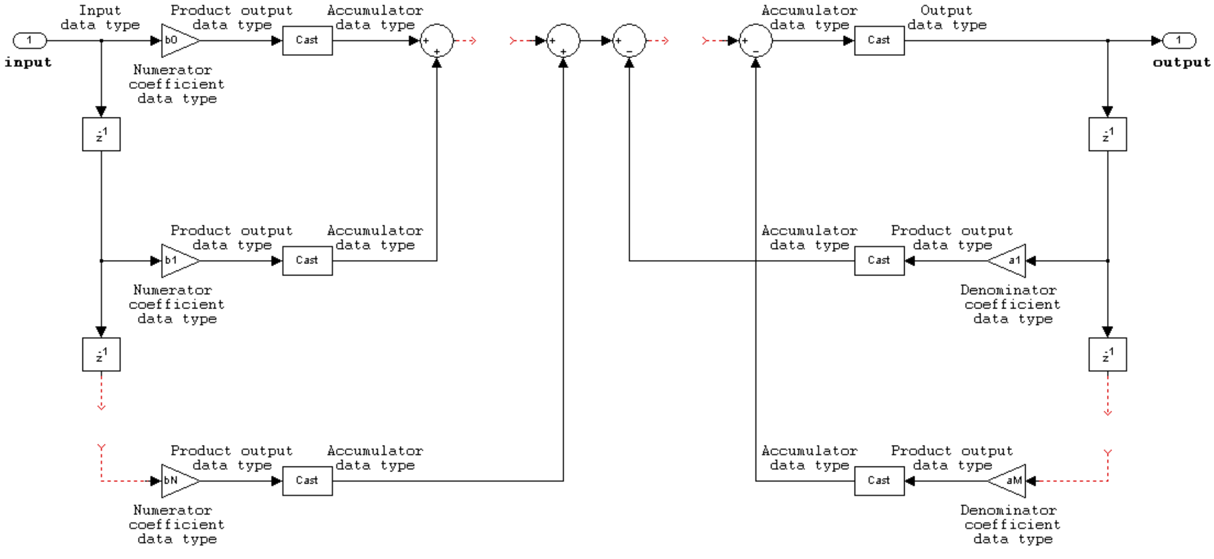
IIR direct form I



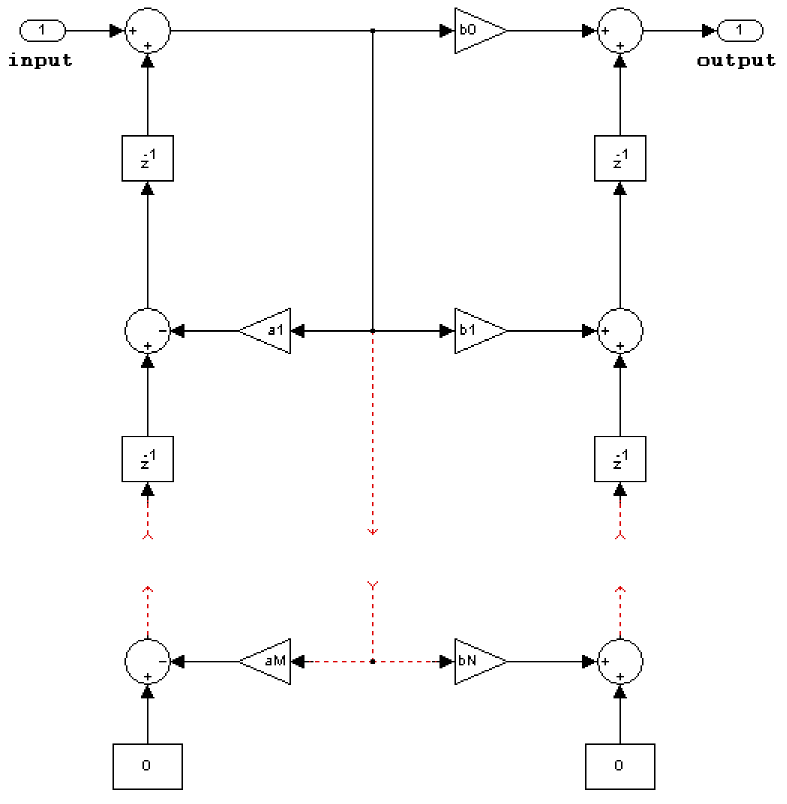
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must be the same complexity as each other.
  - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.
  - When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is real-valued is treated as if it is a complex vector with zero-valued imaginary parts.

- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.
- The State data type cannot be specified on the block mask for this structure, because the input and output states have the same data types as the input and output buffers.



IIR direct form I transposed

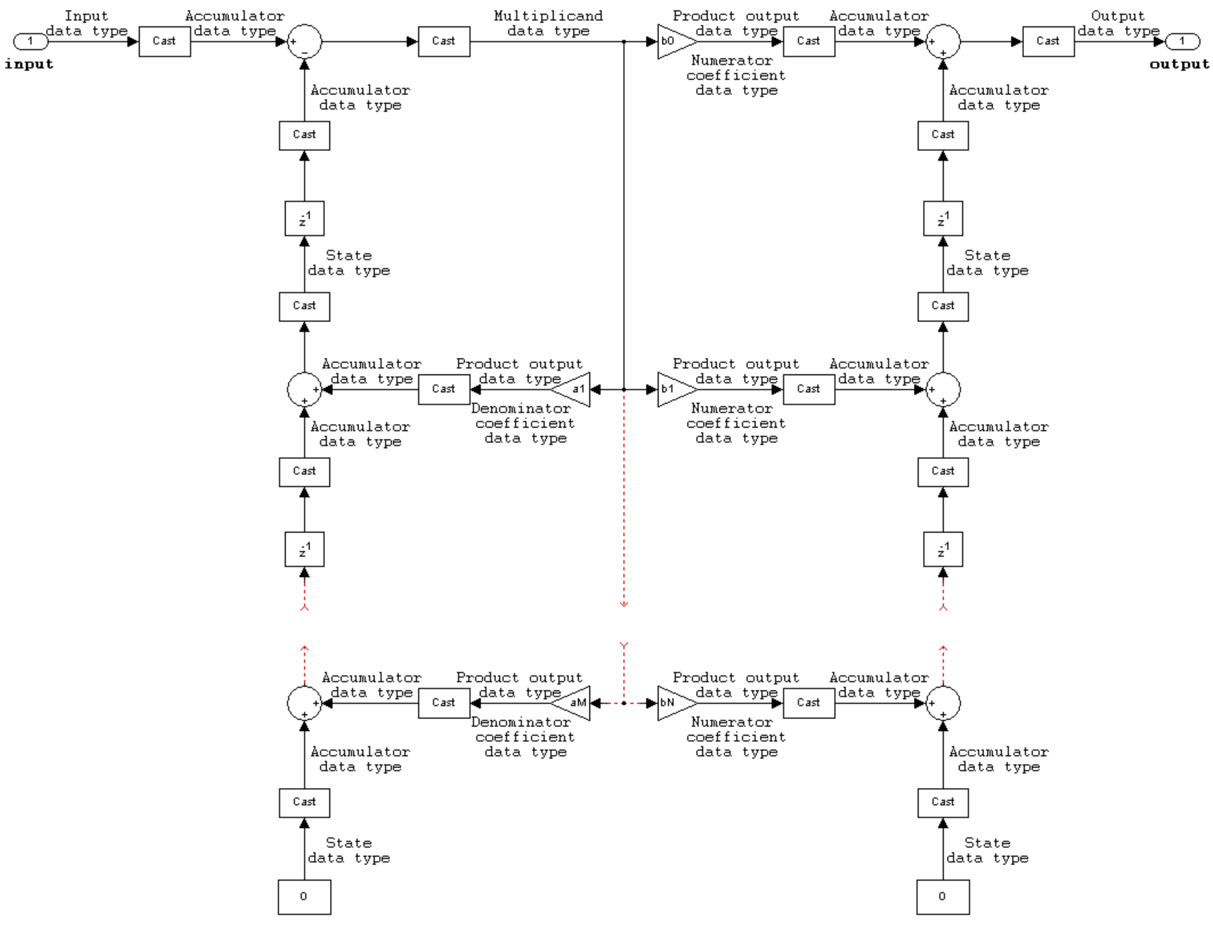


The following constraints are applicable when processing a fixed-point signal with this filter structure:

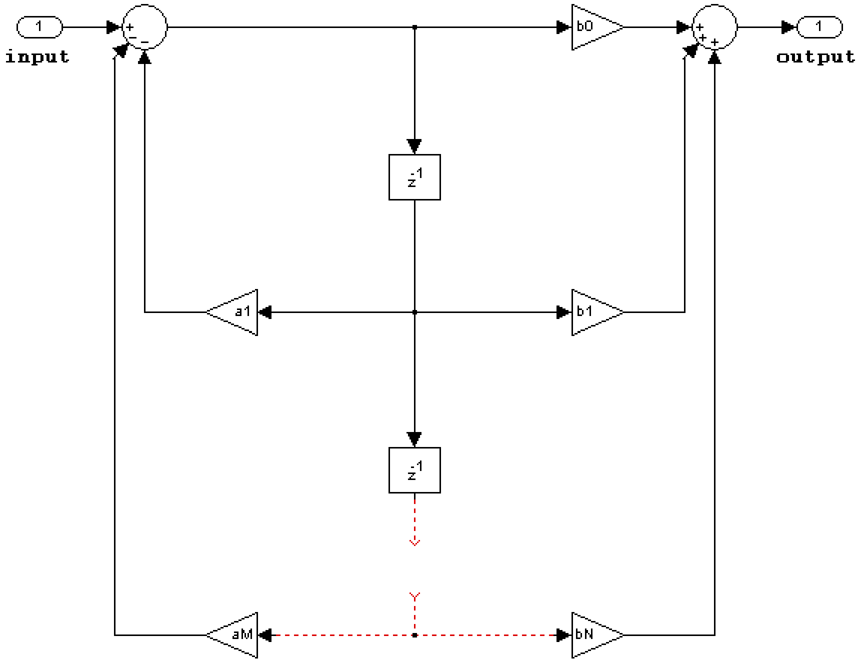
- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must be the same complexity as each other.
  - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.
  - When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but

instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is real-valued is treated as if it is a complex vector with zero-valued imaginary parts.

- States are complex when either the input or the coefficients are complex.
- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.



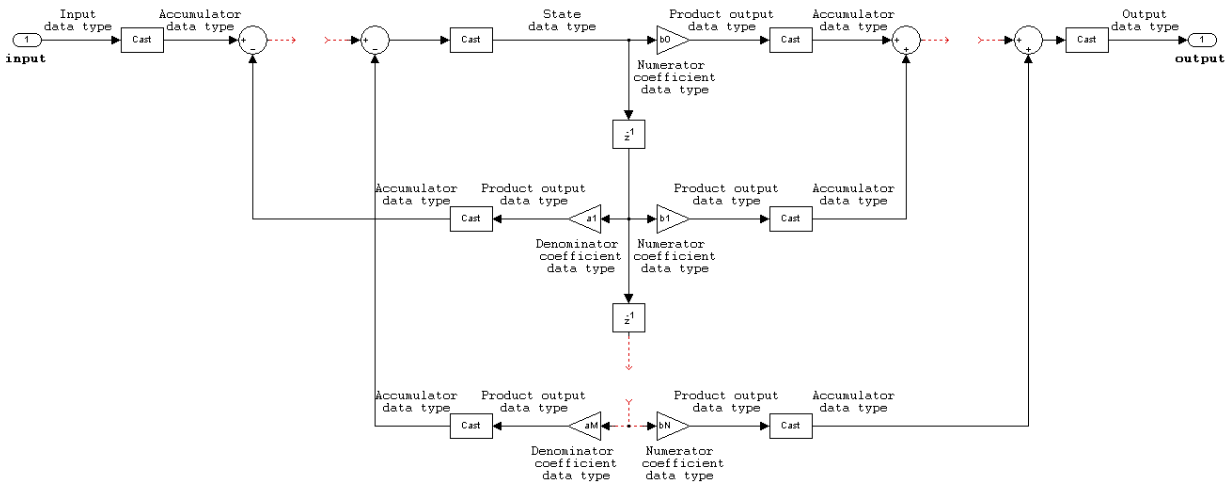
IIR direct form II



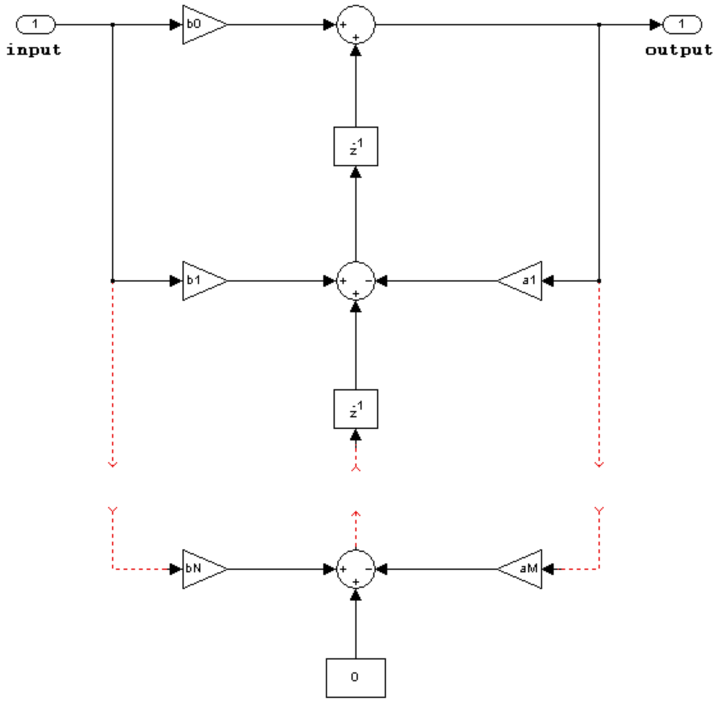
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must be the same complexity as each other.
  - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.
  - When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is real-valued is treated as if it is a complex vector with zero-valued imaginary parts.
- States are complex when either the inputs or the coefficients are complex.

- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.



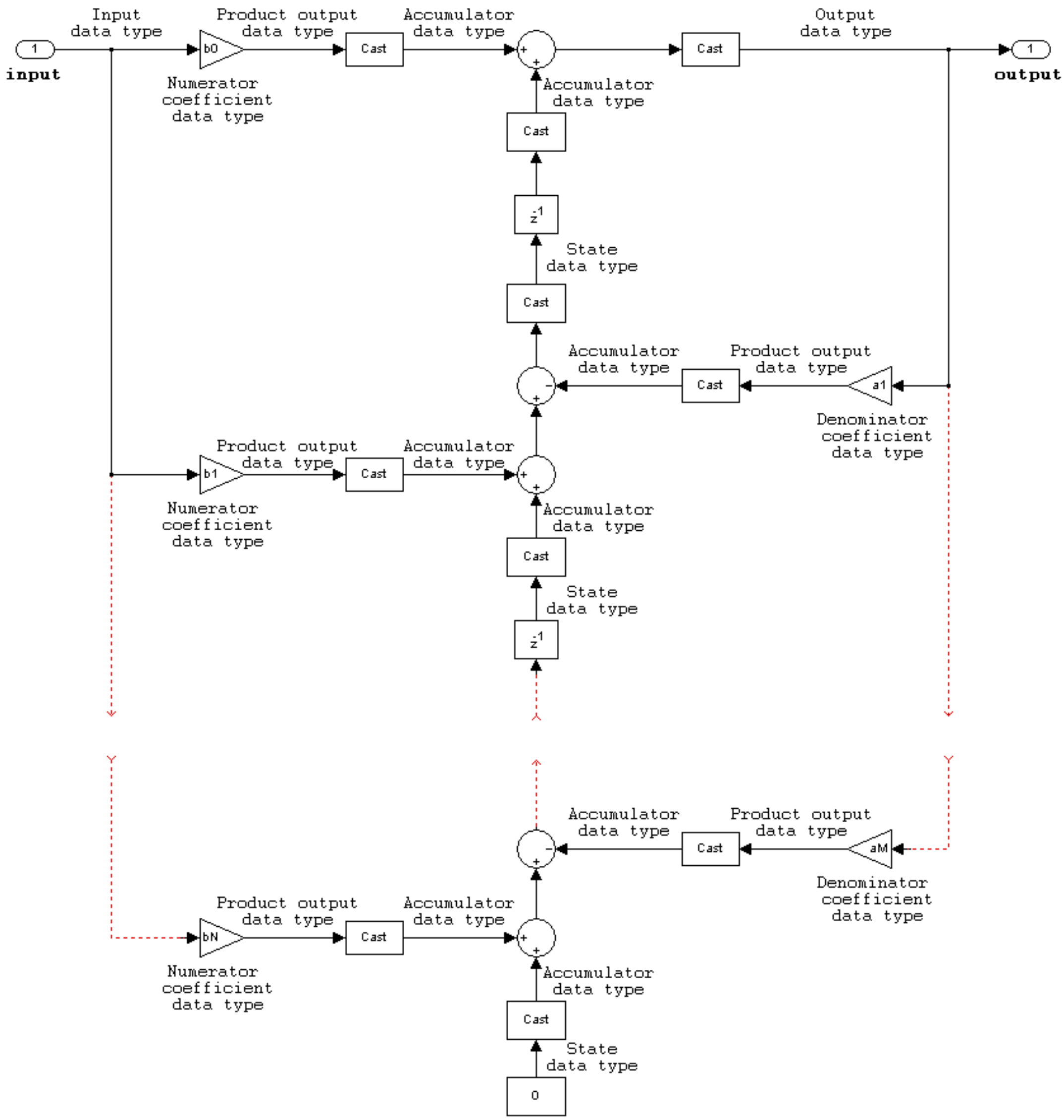
IIR direct form II transposed



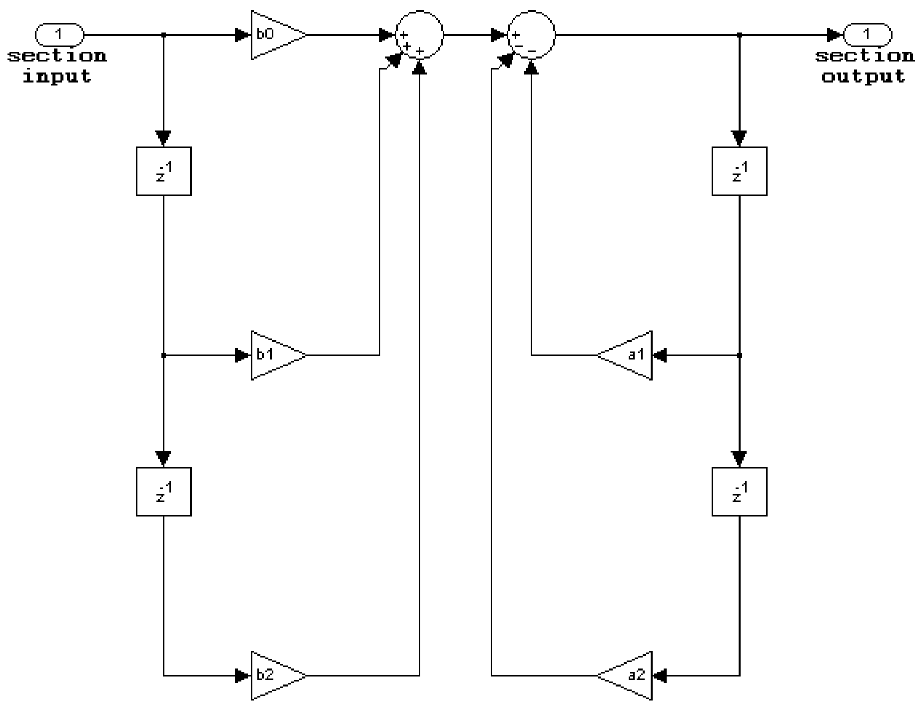
The following constraints are applicable when processing a fixed-point signal with this filter structure:

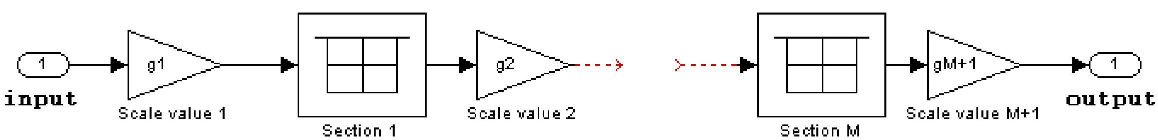
- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must be the same complexity as each other.
    - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.
    - When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is real-valued is treated as if it is a complex vector with zero-valued imaginary parts.

- States are complex when either the inputs or the coefficients are complex.
- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.



IIR biquadratic direct form I



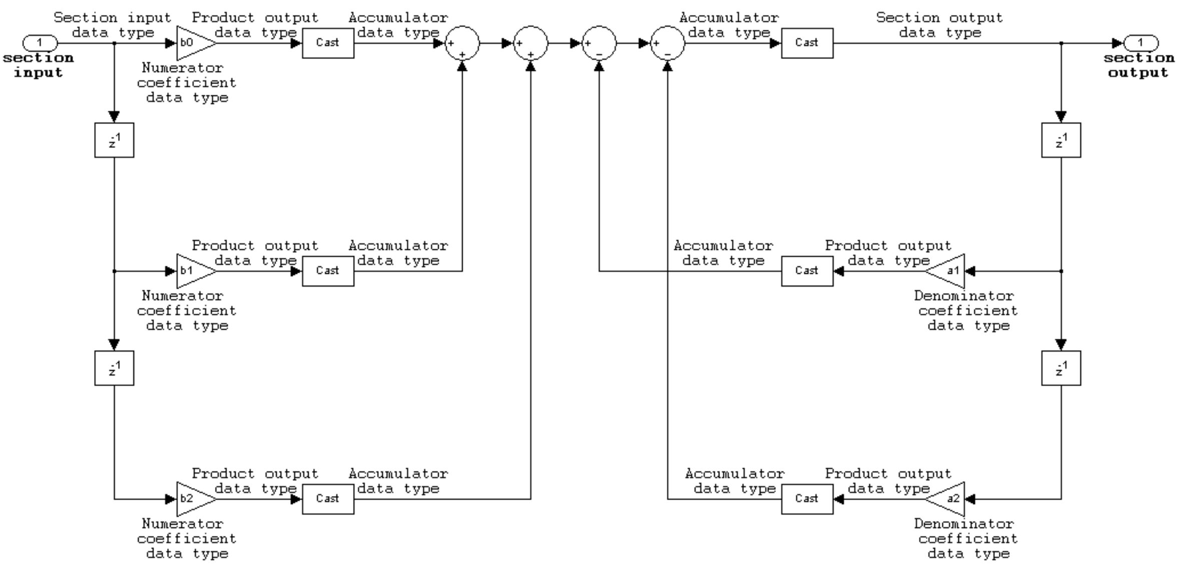


The following constraints are applicable when processing a fixed-point signal with this filter structure:

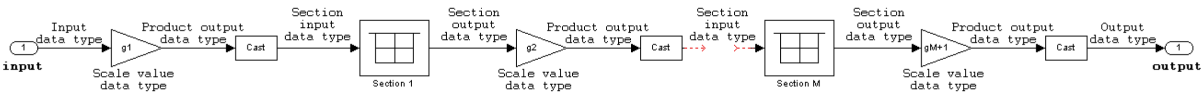
- Inputs and coefficients can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Specify the coefficients by a *M*-by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.

- When the a0 element of any row is not equal to one, that row is normalized by a0 prior to filtering.
- States are complex when either the inputs or the coefficients are complex.
- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.
- Scale values must have the same complexity as the coefficient SOS matrix.
- The scale value parameter must be a scalar or a vector of length M+1, where M is the number of sections.
- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and stage output data type must have the same word length but can have different fraction lengths.

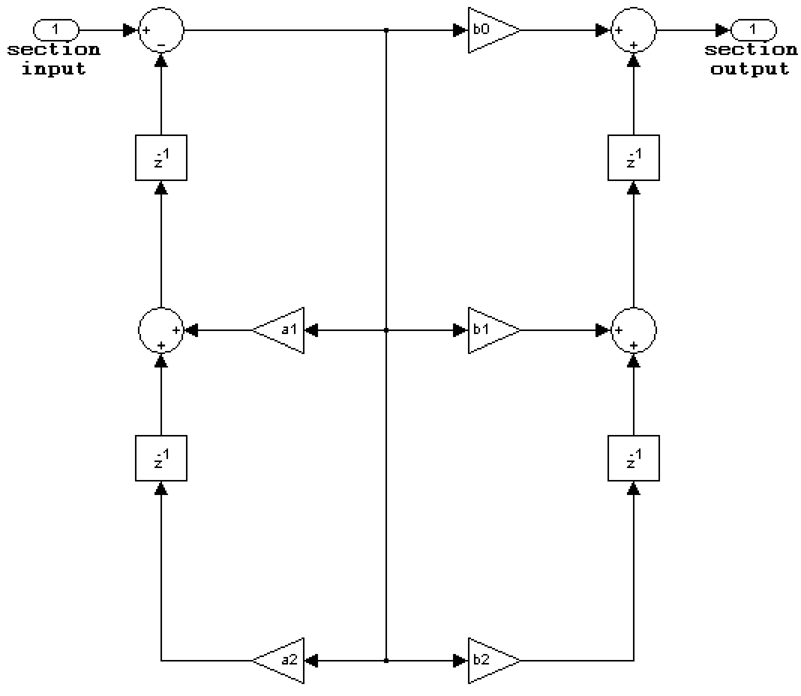
The following diagram shows the data types for one section of the filter.

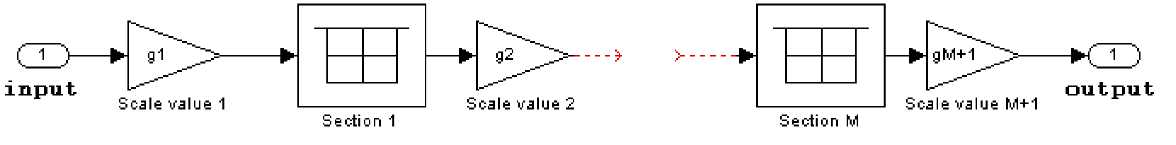


The following diagram shows the data types between filter sections.



IIR biquadratic direct form I transposed



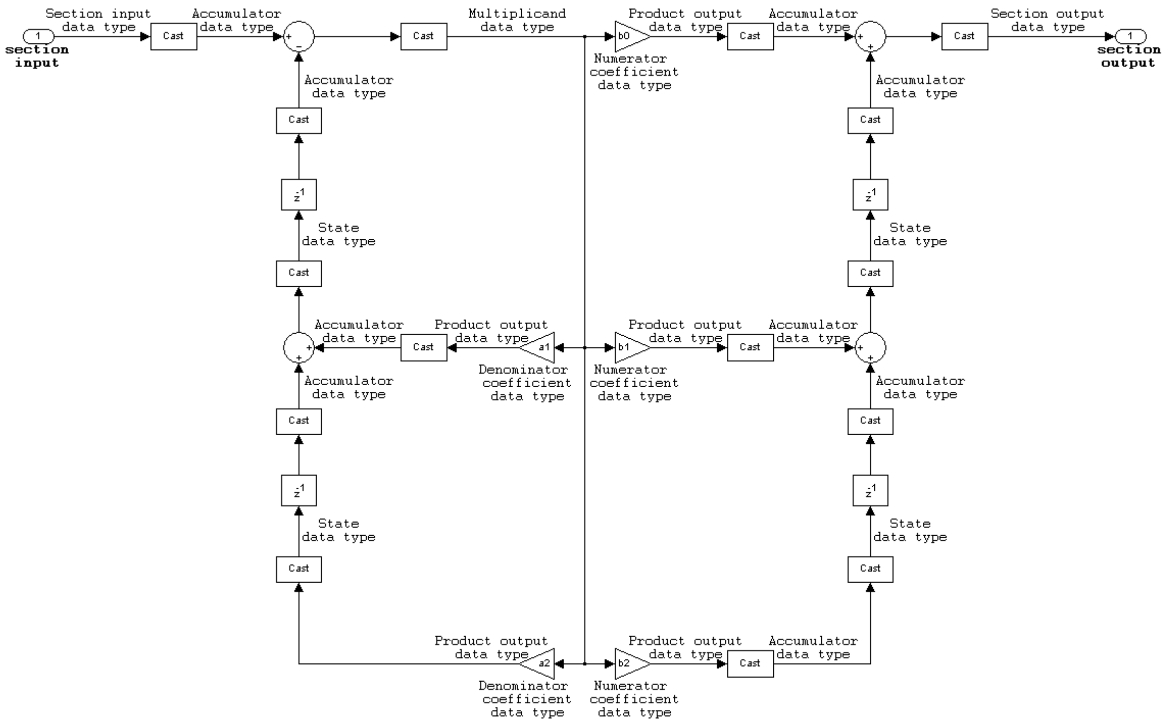


The following constraints are applicable when processing a fixed-point signal with this filter structure:

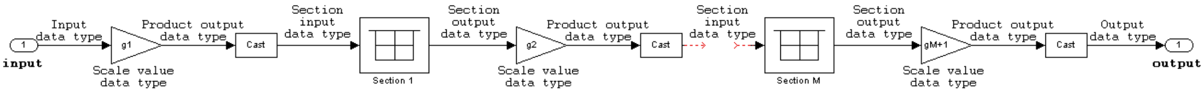
- Inputs and coefficients can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Specify the coefficients by a *M*-by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.
- When the a0 element of any row is not equal to one, that row is normalized by a0 prior to filtering.

- States are complex when either the inputs or the coefficients are complex.
- Scale values must have the same complexity as the coefficient SOS matrix.
- The scale value parameter must be a scalar or a vector of length $M+1$, where M is the number of sections.
- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and section output data type must have the same word length but can have different fraction lengths.

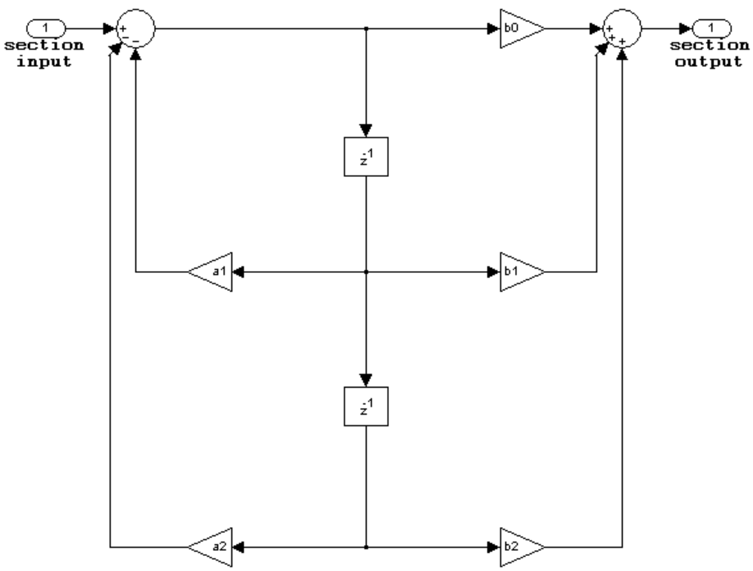
The following diagram shows the data types for one section of the filter.

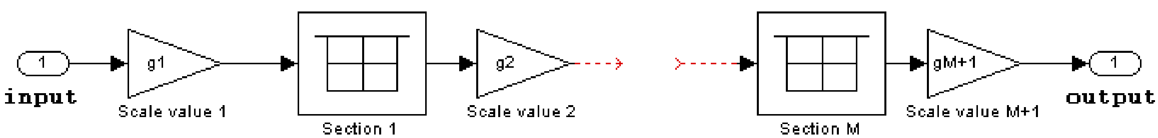


The following diagram shows the data types between filter sections.



IIR biquadratic direct form II



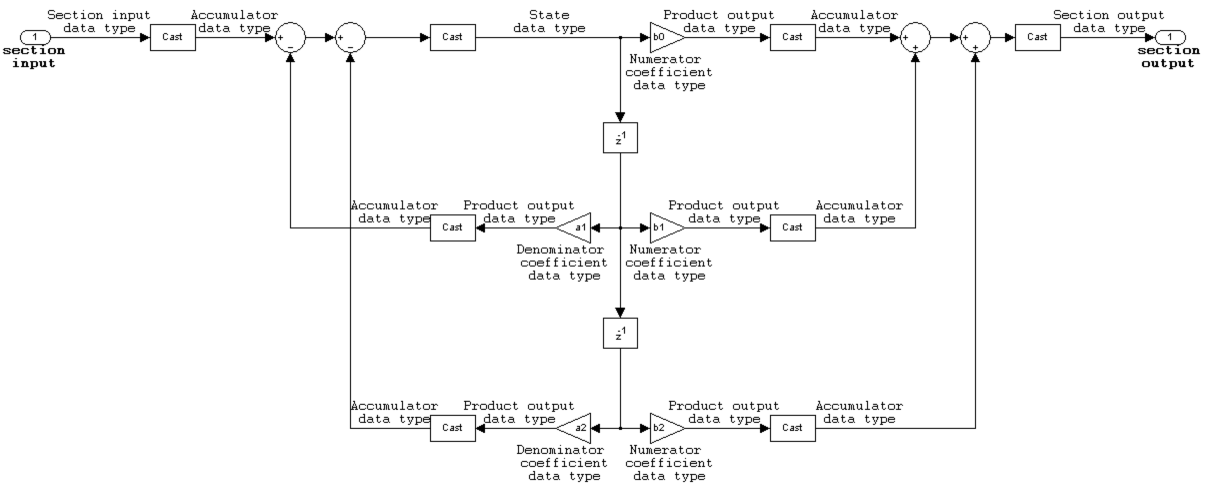


The following constraints are applicable when processing a fixed-point signal with this filter structure:

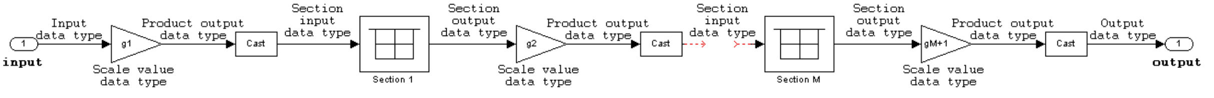
- Inputs and coefficients can be real or complex.
- Numerator and denominator coefficients can be real or complex.

- Specify the coefficients by a M-by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.
- When the a0 element of any row is not equal to one, that row is normalized by a0 prior to filtering.
- States are complex when either the inputs or the coefficients are complex.
- Scale values must have the same complexity as the coefficient SOS matrix.
- The scale value parameter must be a scalar or a vector of length M+1, where M is the number of sections.
- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and section output data type must have the same word length but can have different fraction lengths.

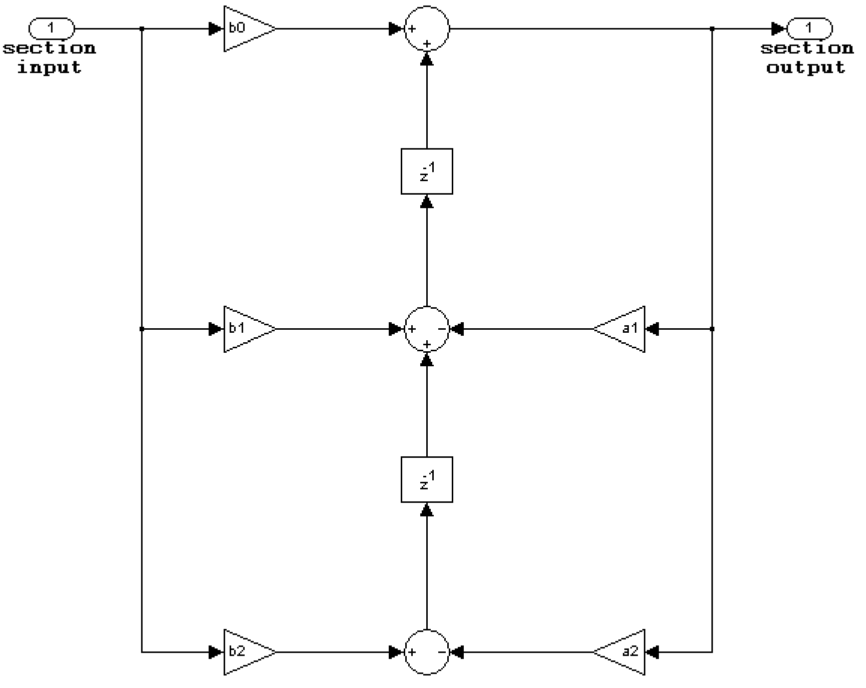
The following diagram shows the data types for one section of the filter.

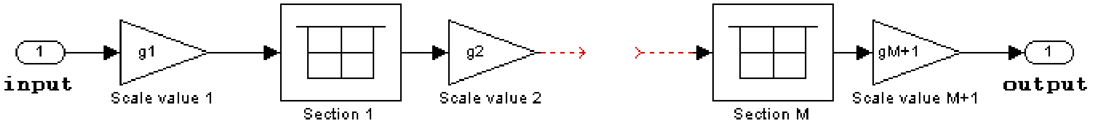


The following diagram shows the data types between filter sections.



IIR biquadratic direct form II transposed

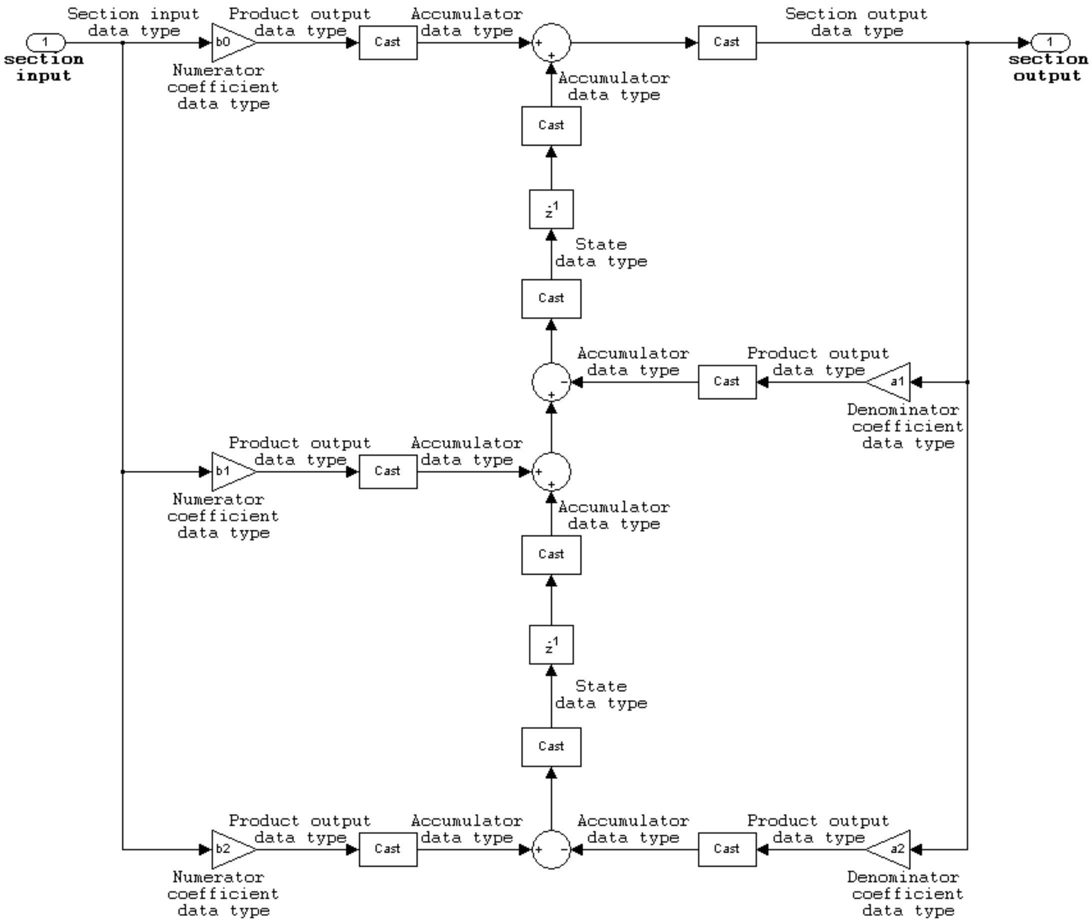




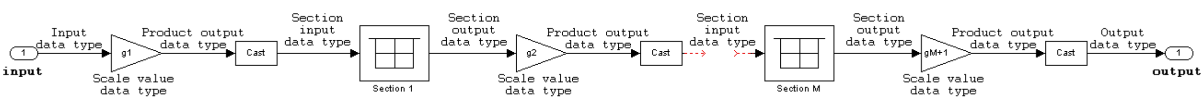
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Specify the coefficients by a *M*-by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.
- When the a0 element of any row is not equal to one, that row is normalized by a0 prior to filtering.
- States are complex when either the inputs or the coefficients are complex.
- Scale values must have the same complexity as the coefficient SOS matrix.
- The scale value parameter must be a scalar or a vector of length *M*+1, where *M* is the number of sections.
- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and section output data type must have the same word length but can have different fraction lengths.

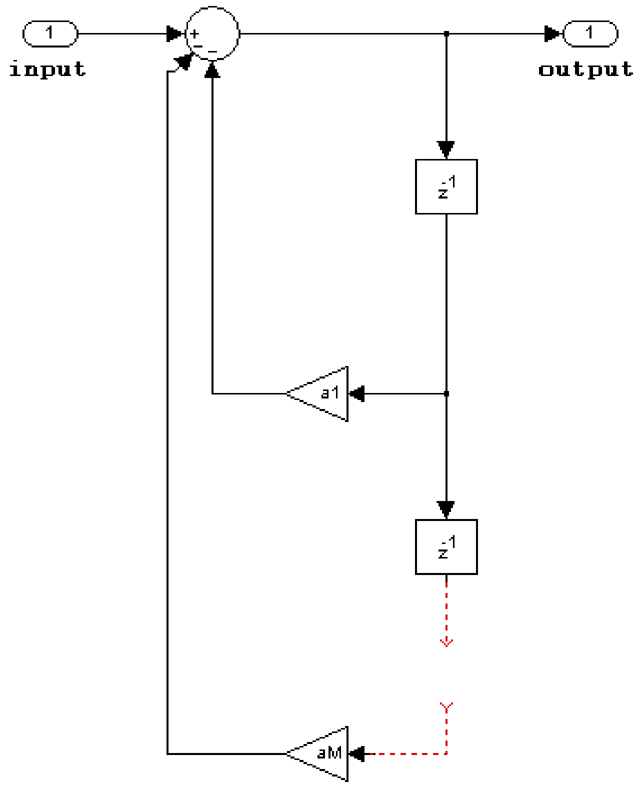
The following diagram shows the data types for one section of the filter.



The following diagram shows the data types between filter sections.

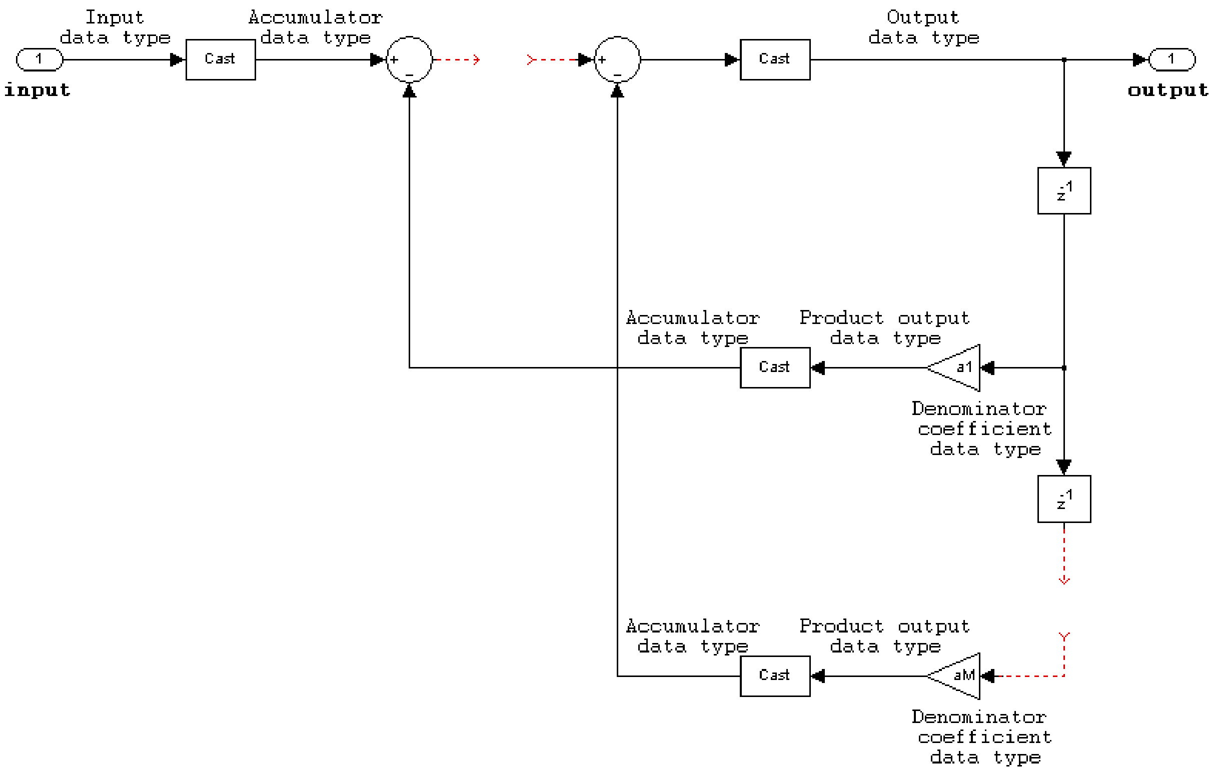


IIR (all poles) direct form

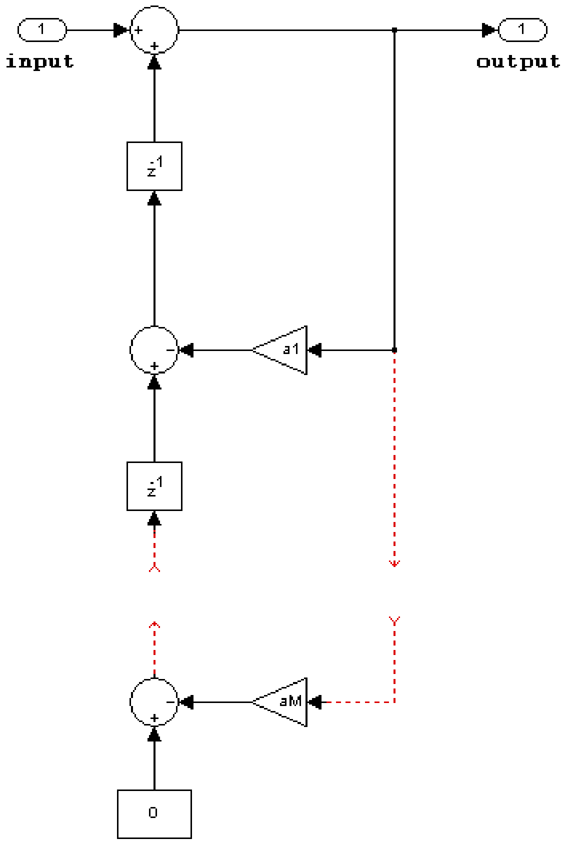


The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.
- Denominator coefficients can be real or complex.
- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.

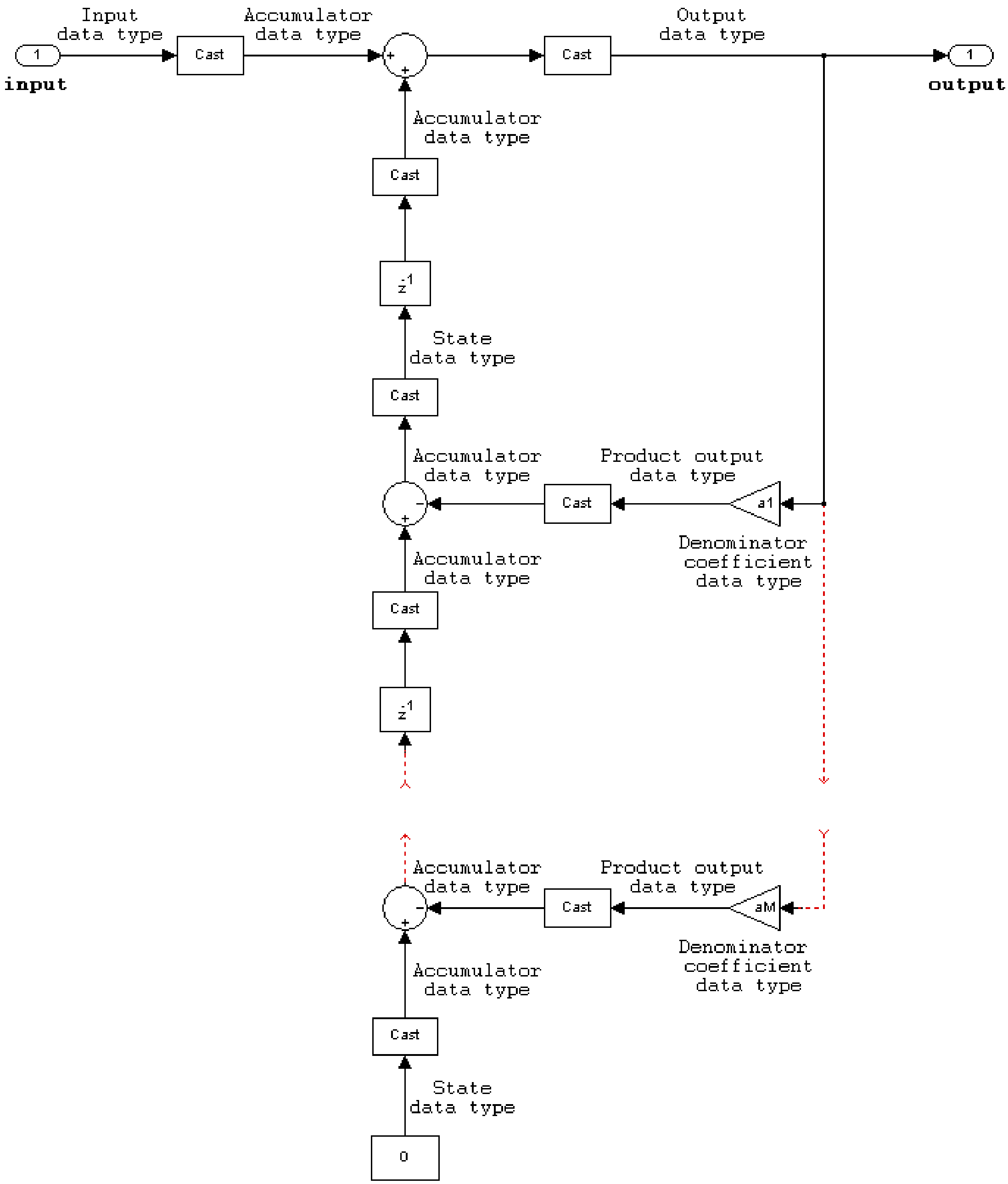

Input
data type
Accumulator
data type
Output
data type
1
input
Cast
Cast
1
output
z-1
Accumulator
data type
Product output
data type
Cast
a1
Denominator
coefficient
data type
z-1
Accumulator
data type
Product output
data type
Cast
aM
Denominator
coefficient
data type

IIR (all poles) direct form transposed

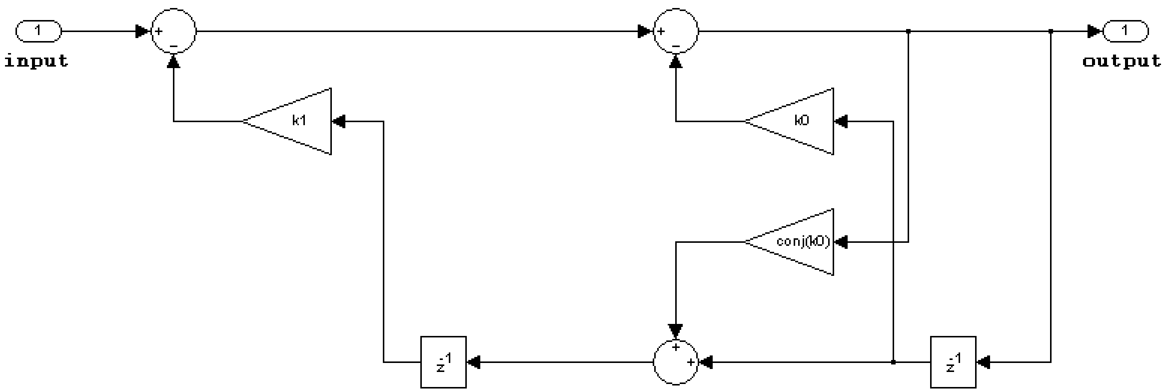


The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.
- Denominator coefficients can be real or complex.

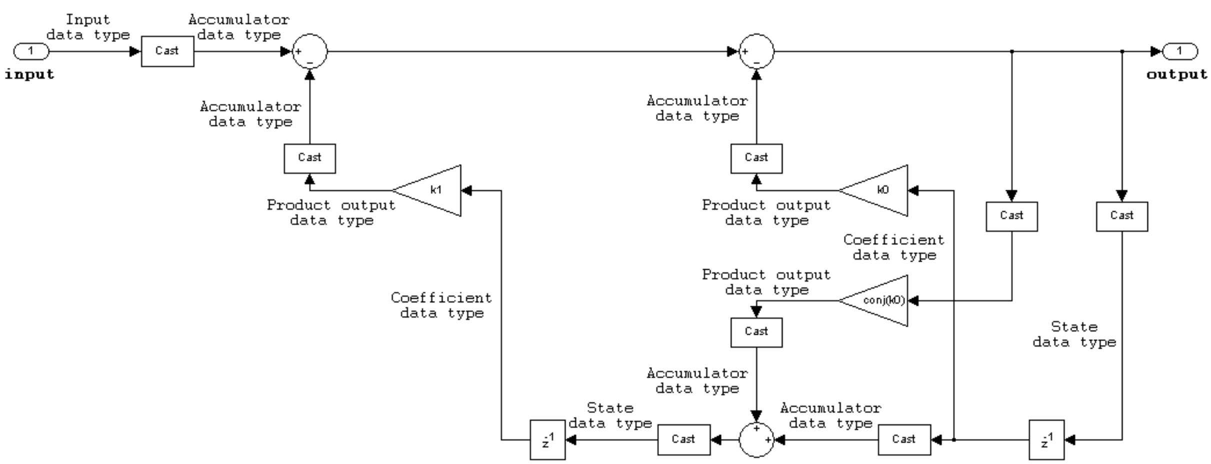

Input
data type
Accumulator
data type
Output
data type
1
input
Cast
+
+
Cast
1
output
Accumulator
data type
Cast
z-1
State
data type
Cast
Accumulator
data type
Product output
data type
Cast
a1
Denominator
coefficient
data type
Accumulator
data type
Cast
z-1
Accumulator
data type
Product output
data type
Cast
aM
Denominator
coefficient
data type
Accumulator
data type
Cast
State
data type
0

IIR (all poles) direct form lattice AR

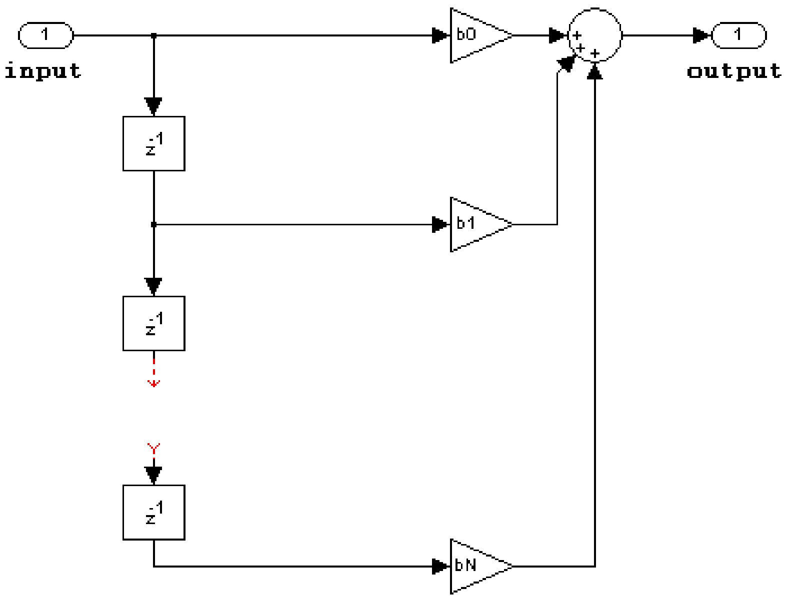


The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.
- Coefficients can be real or complex.

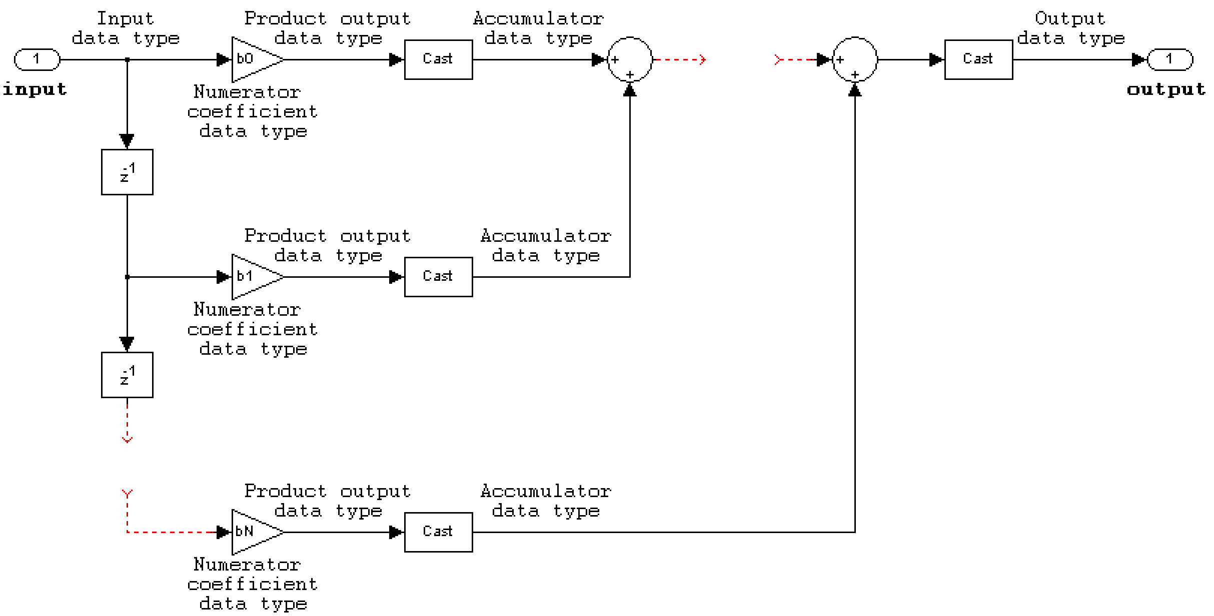


FIR (all zeros) direct form

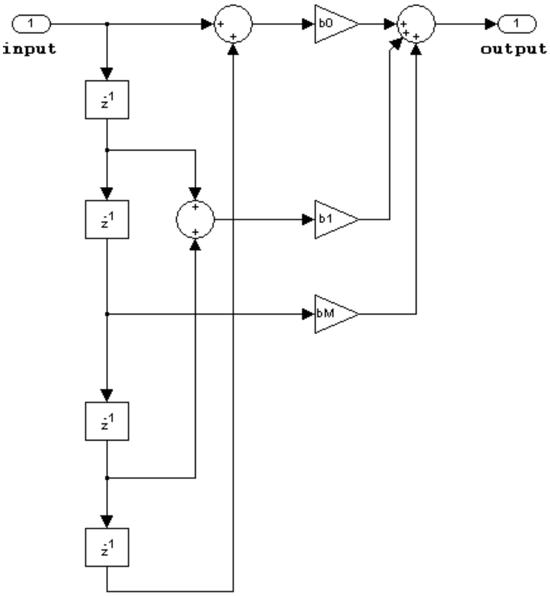


The following constraints are applicable when processing a fixed-point signal with this filter structure:

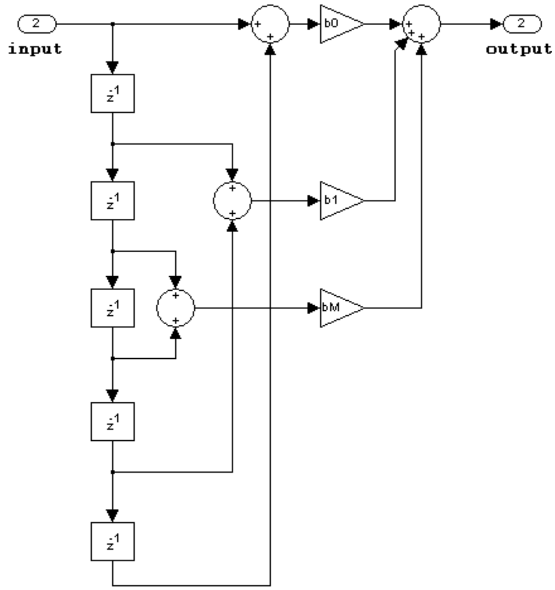
- Inputs can be real or complex.
- Numerator coefficients can be real or complex.
- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.


Input
data type
1
input
Product output
data type
b0
Numerator
coefficient
data type
Cast
Accumulator
data type
+
+
z-1
Product output
data type
b1
Numerator
coefficient
data type
Cast
Accumulator
data type
z-1
Product output
data type
bN
Numerator
coefficient
data type
Cast
Accumulator
data type
Cast
Output
data type
1
output

FIR (all zeros) direct form symmetric



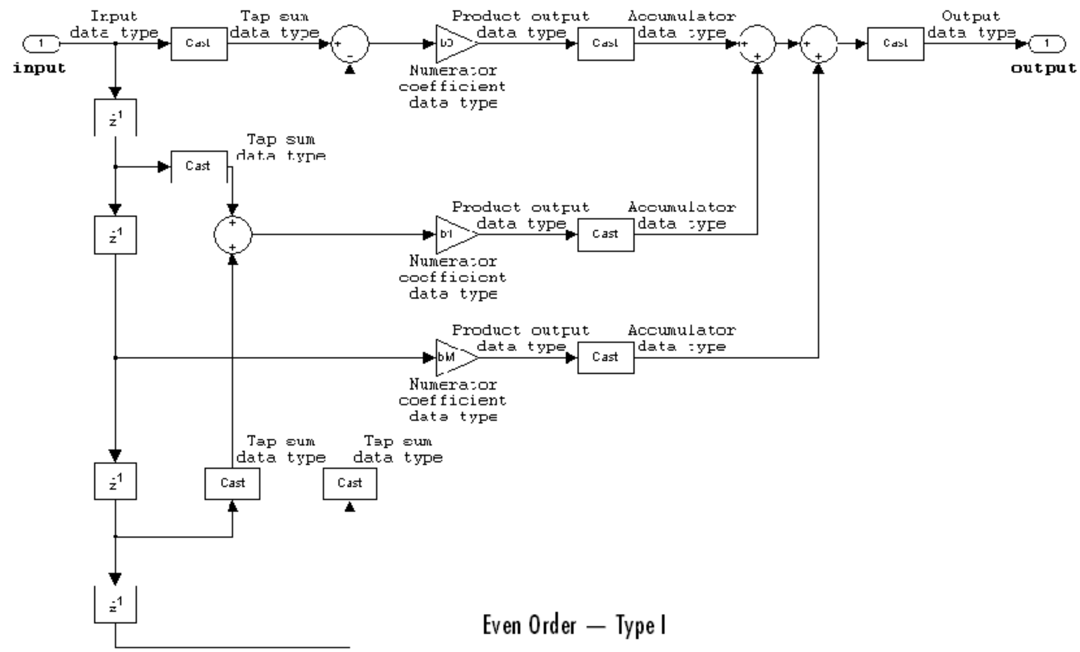
Even Order — Type I



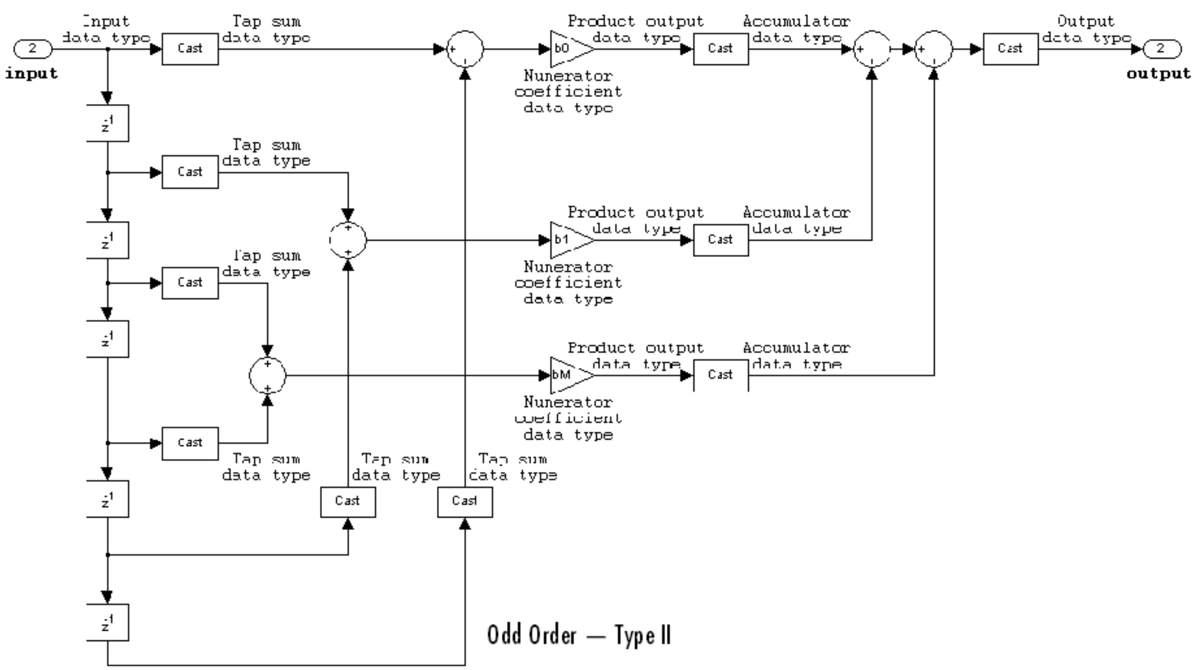
Odd Order — Type II

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator coefficients can be real or complex.
- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.
- It is assumed that the filter coefficients are symmetric. Only the first half of the coefficients are used for filtering.
- The **Tap Sum** parameter determines the data type the filter uses when it sums the inputs prior to multiplication by the coefficients.

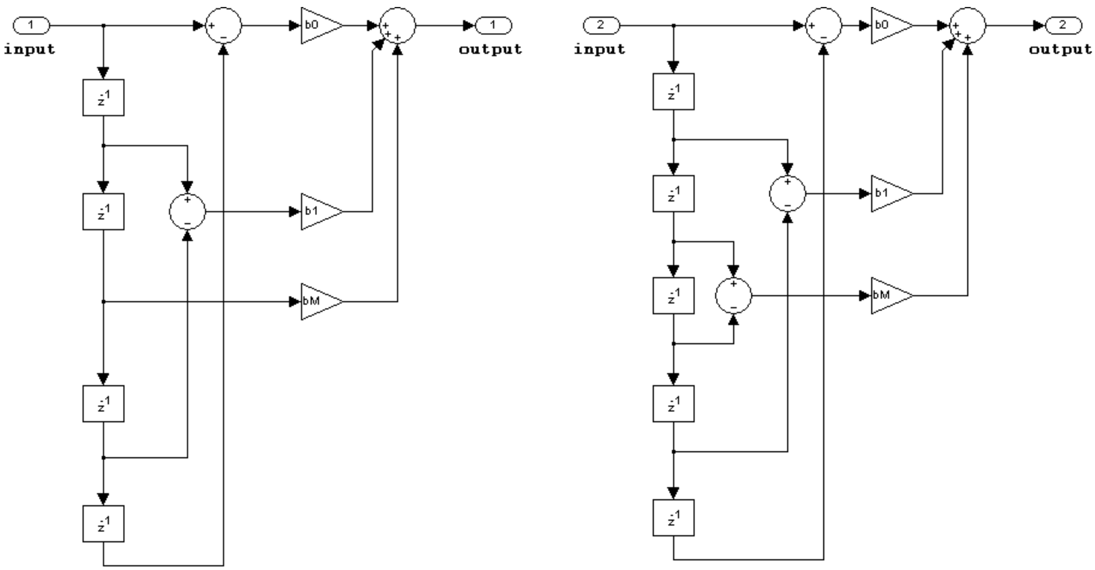


Even Order — Type I



Odd Order — Type II

FIR (all zeros) direct form antisymmetric

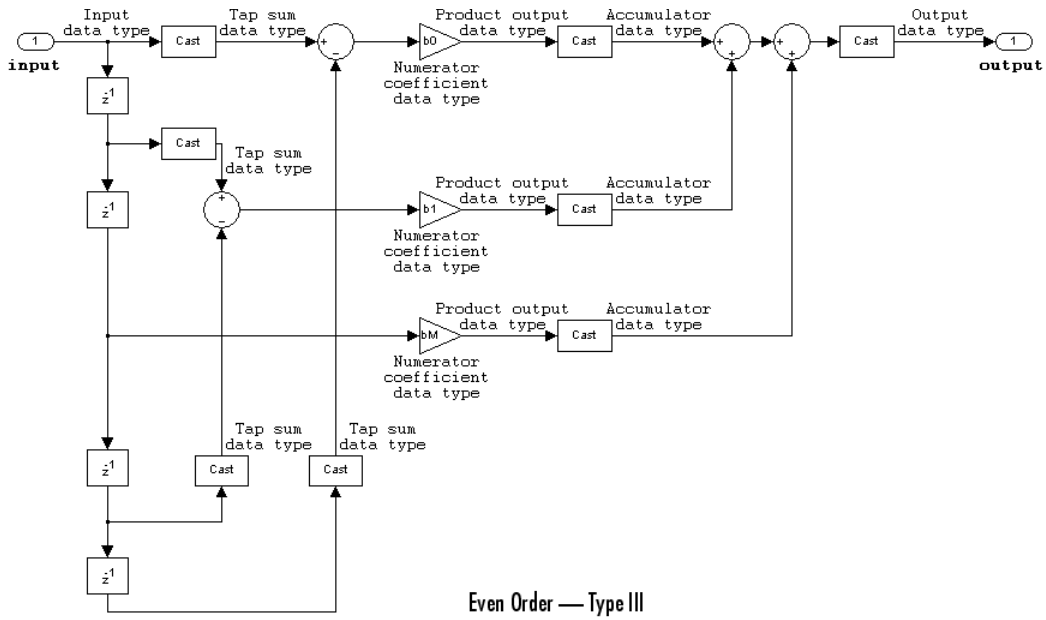


Even Order — Type III

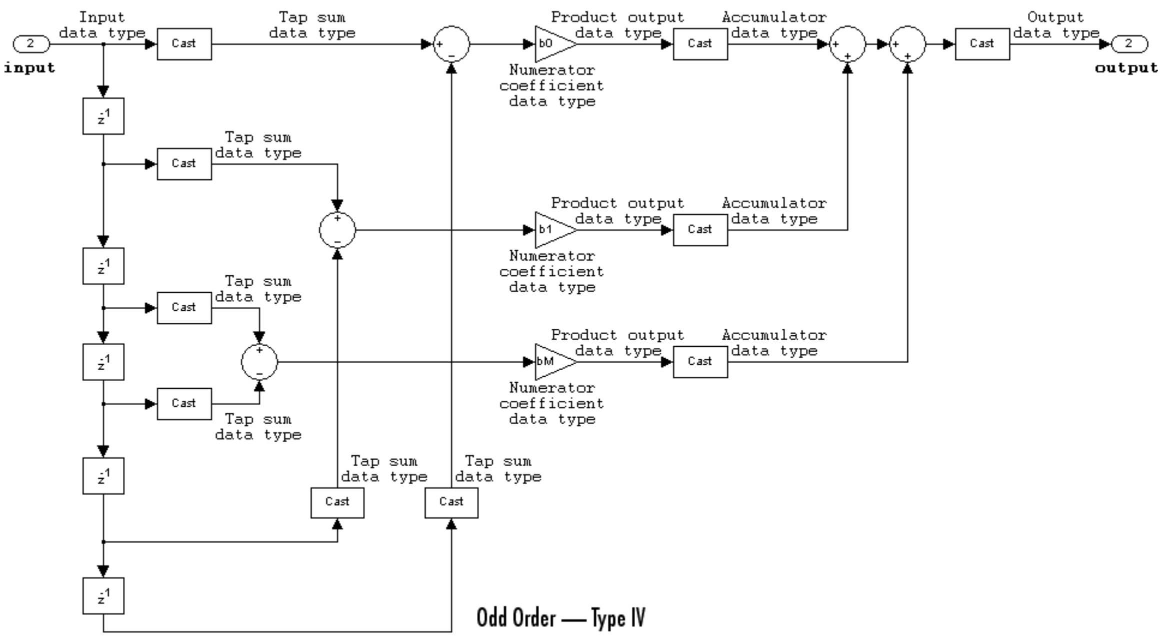
Odd Order — Type IV

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator coefficients can be real or complex.
- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.
- It is assumed that the filter coefficients are antisymmetric. Only the first half of the coefficients are used for filtering.
- The **Tap Sum** parameter determines the data type the filter uses when it sums the inputs prior to multiplication by the coefficients.

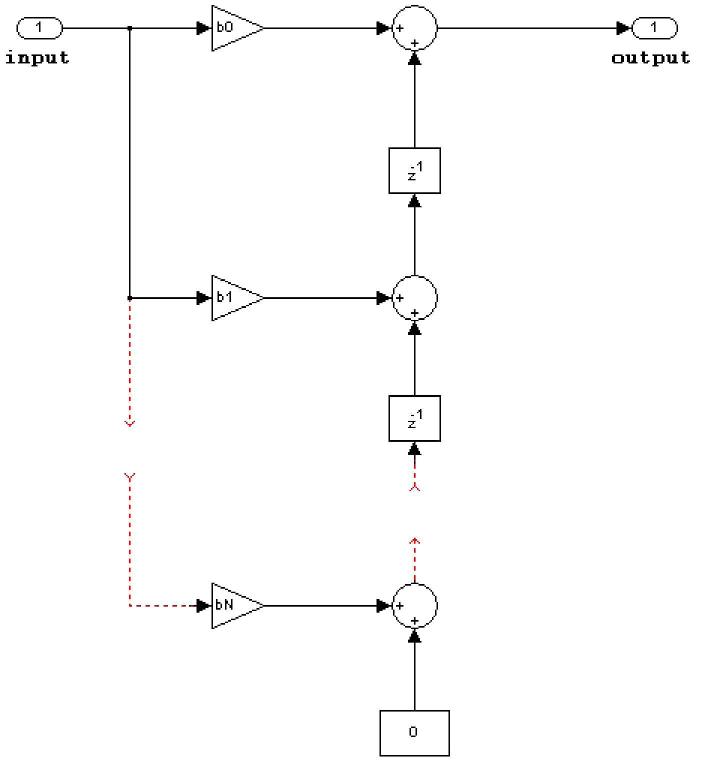


Even Order — Type III



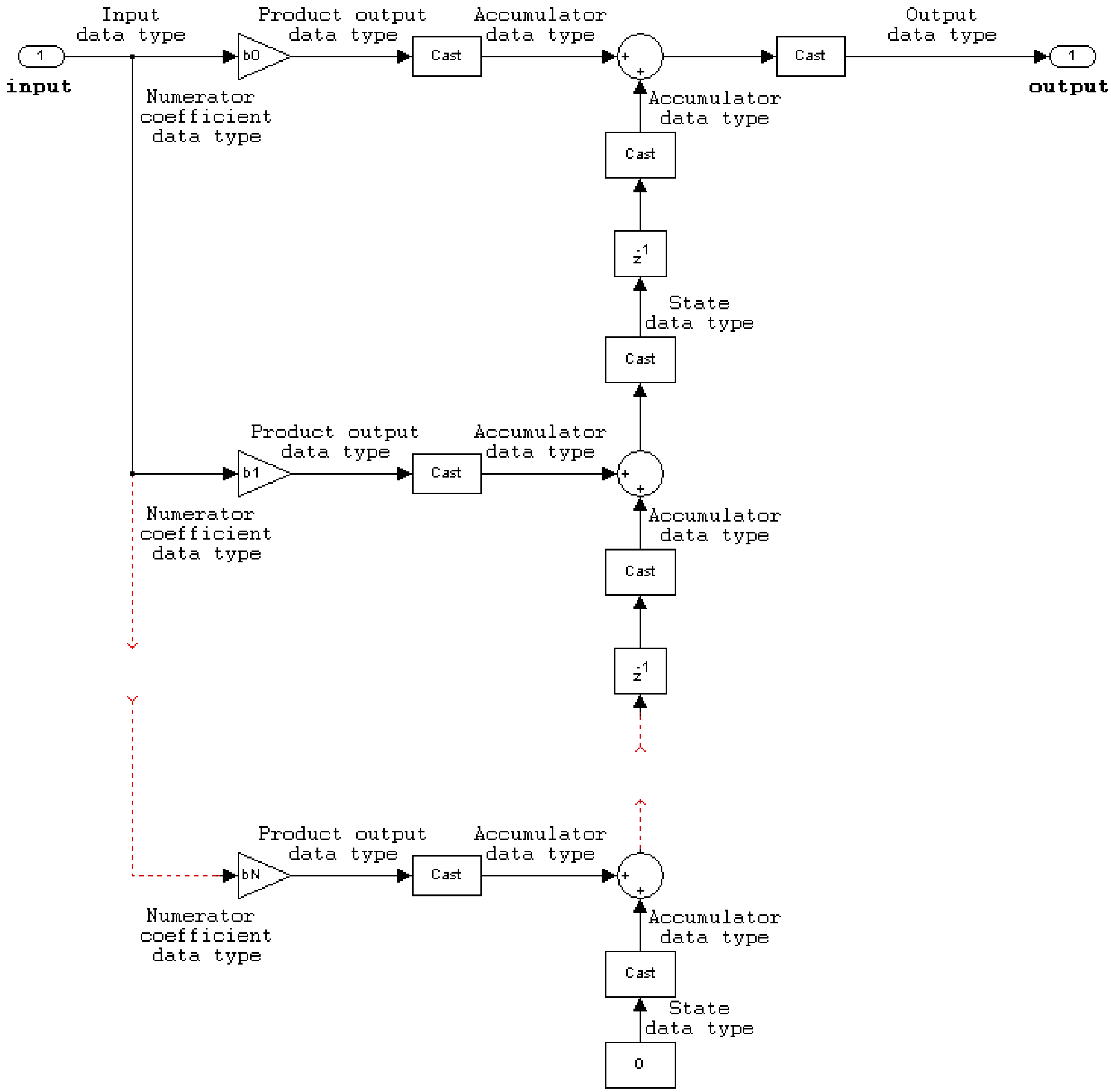
Odd Order — Type IV

FIR (all zeros) direct form transposed

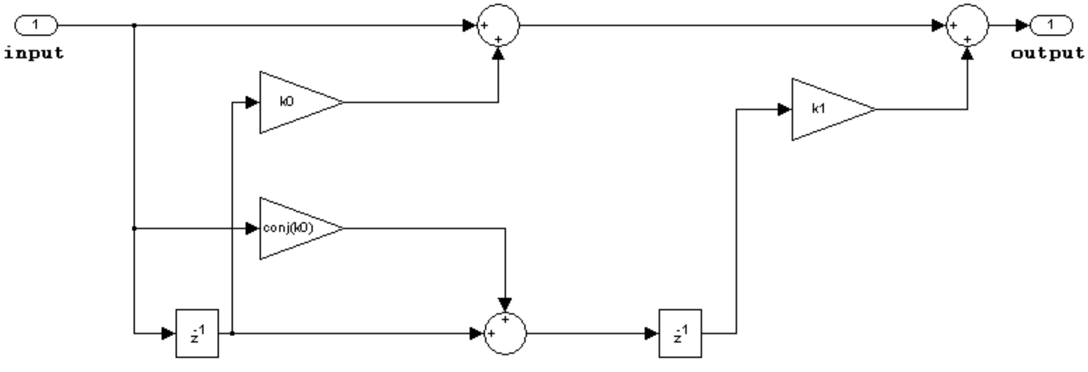


The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Coefficients can be real or complex.
- States are complex when either the inputs or the coefficients are complex.

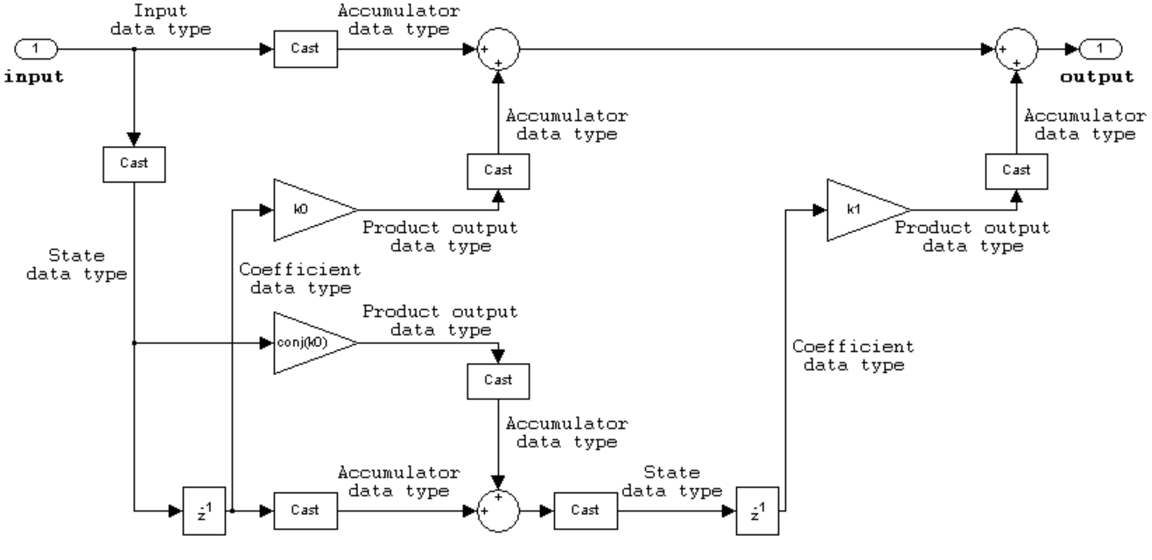

Input
data type
Product output
data type
Accumulator
data type
Output
data type
1
input
b0
Cast
1
output
Numerator
coefficient
data type
Accumulator
data type
Cast
z-1
State
data type
Cast
Product output
data type
Accumulator
data type
b1
Cast
Numerator
coefficient
data type
Accumulator
data type
Cast
z-1
Product output
data type
Accumulator
data type
bN
Cast
Numerator
coefficient
data type
Accumulator
data type
Cast
State
data type
0

FIR (all zeros) lattice MA



The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.
- Coefficients can be real or complex.



Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

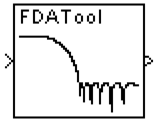
See Also

`Allpole Filter`	DSP System Toolbox
`Digital Filter Design`	DSP System Toolbox
`Biquad Filter`	DSP System Toolbox
Discrete Filter	Simulink
Discrete FIR Filter	Simulink
`Filter Realization Wizard`	DSP System Toolbox
`dfilt`	DSP System Toolbox
`filterDesigner`	DSP System Toolbox
`fvtool`	Signal Processing Toolbox
`sptool`	Signal Processing Toolbox

Introduced in R2014b

Digital Filter Design

Design and implement digital FIR and IIR filters



Library

Filtering / Filter Implementations

`dsparch4`

Description

Note: Use this block to design, analyze, and efficiently implement floating-point filters. The following blocks also implement digital filters, but serve slightly different purposes:

- `Discrete FIR Filter` and `Biquad Filter`— Use to efficiently implement floating-point or fixed-point filters that you have already designed. These blocks provide the same exact filter implementation as the Digital Filter Design block.
- `Filter Realization Wizard` — Use to implement floating-point or fixed-point filters built from Sum, Gain, and Unit Delay blocks. You can either design the filter within this block, or import the coefficients of a filter that you designed elsewhere.

The Digital Filter Design block implements a digital FIR or IIR filter that you design using the filter designer (`filterDesigner`) app. This block provides the same filter implementation as the Discrete FIR Filter or Biquad Filter blocks.

You must specify whether the block performs frame-based or sample-based processing on the input by setting the **Input processing** parameter. The block applies the specified filter to each channel of a discrete-time input signal, and outputs the result. The outputs of the block numerically match the outputs of the Discrete FIR Filter or Biquad Filter block, the MATLAB `filter` function, and the DSP System Toolbox `filter` function.

The sampling frequency, Fs, that you specify in the filter designer app should be identical to the sampling frequency of the input to the Digital Filter Design block. When the sampling frequencies do not match, the Digital Filter Design block returns a warning message and inherits the sampling frequency of the input block.

Designing the Filter

Double-click the Digital Filter Design block to open filter designer. Use filter designer to design or import a digital FIR or IIR filter. To learn how to design filters with this block and filter designer, see the following topics:

- “Digital Filter Design Block”
- `filterDesigner` reference page

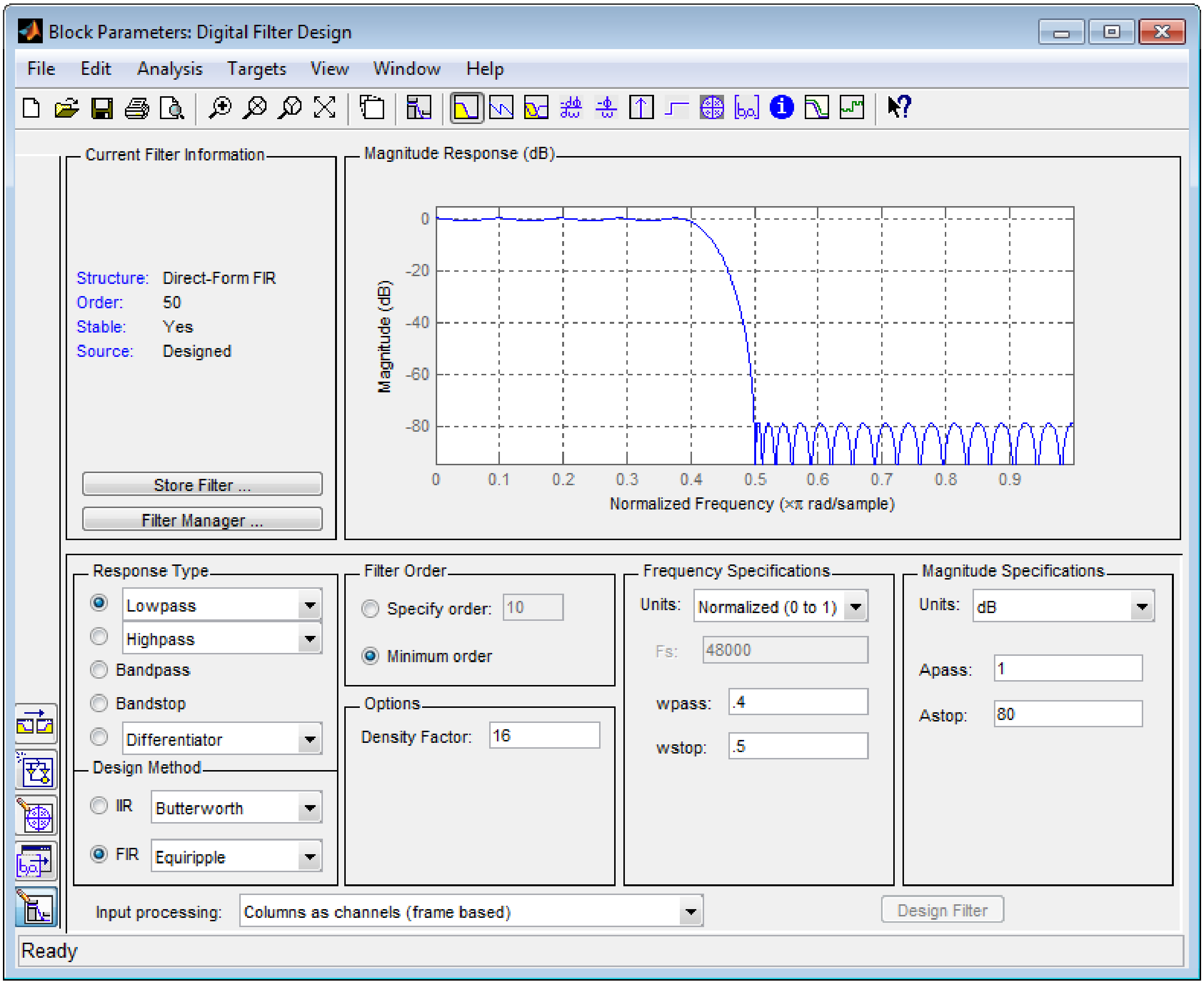
Tuning the Filter During Simulation

You can tune the filter specifications in filter designer during simulations as long as your changes do not modify the filter length or filter order. The filter updates as soon as you apply any filter changes in filter designer.

Examples

See “Digital Filter Design Block”.

Dialog Box



For more information about the parameters on this dialog box, see "Getting Started with Filter Designer" (Signal Processing Toolbox).

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

`Discrete FIR Filter`	DSP System Toolbox
`Biquad Filter`	DSP System Toolbox
Analog Filter Design	DSP System Toolbox
Window Function	DSP System Toolbox
`filterDesigner`	DSP System Toolbox
`filter`	DSP System Toolbox
`fvtool`	Signal Processing Toolbox
`sptool`	Signal Processing Toolbox

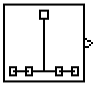
To learn how to use this block and filter designer, see the following:

- "Filter Design"
- "Filter Analysis"
- "Digital Filter Design Block"

Introduced before R2006a

Discrete Impulse

Generate discrete impulse



Library

Sources

`dspsrcs4`

Description

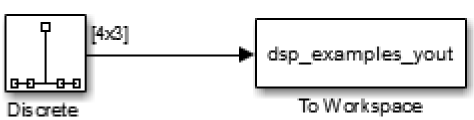
The Discrete Impulse block generates an impulse (the value `1`) at output sample `D+1`, where `D` is specified by the **Delay** parameter (`D` ≥`0`). All output samples preceding and following sample `D+1` are zero.

When `D` is a length-N vector, the block generates an M-by-N matrix output representing N distinct channels, where frame size M is specified by the **Samples per frame** parameter. The impulse for the `i`th channel appears at sample `D(i)+1`.

The **Sample time** parameter value, T_s, specifies the output signal sample period. The resulting frame period is $M*T_s$.

Examples

Construct the model below.



Configure the Discrete Impulse block to generate a three-channel output of type `double`, with impulses at samples 1, 4, and 6 of channels 1, 2, and 3, respectively. Use a sample period of 0.25 and a frame size of 4. The corresponding settings should be as follows:

- **Delay** = `[0 3 5]`
- **Sample time** = `0.25`
- **Samples per frame** = 4
- **Output data type** = `double`

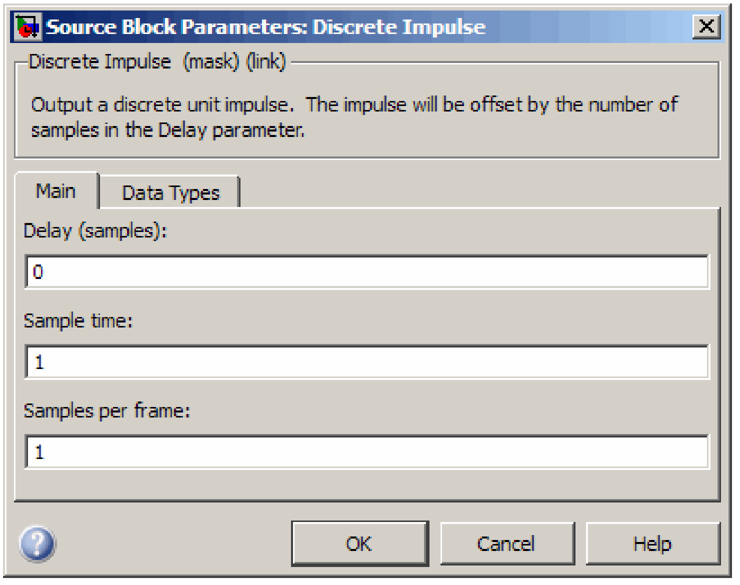
Run the model and look at the output, `dsp_examples_yout`. The first few samples of each channel are shown below.

```
dsp_examples_yout(1:10,:)

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

The block generates an impulse at sample 1 of channel 1 (first column), at sample 4 of channel 2 (second column), and at sample 6 of channel 3 (third column).

Dialog Box

The **Main** pane of the Discrete Impulse block dialog appears as follows.



Delay

The number of zero-valued output samples, `D`, preceding the impulse. A length-N vector specifies an N-channel output.

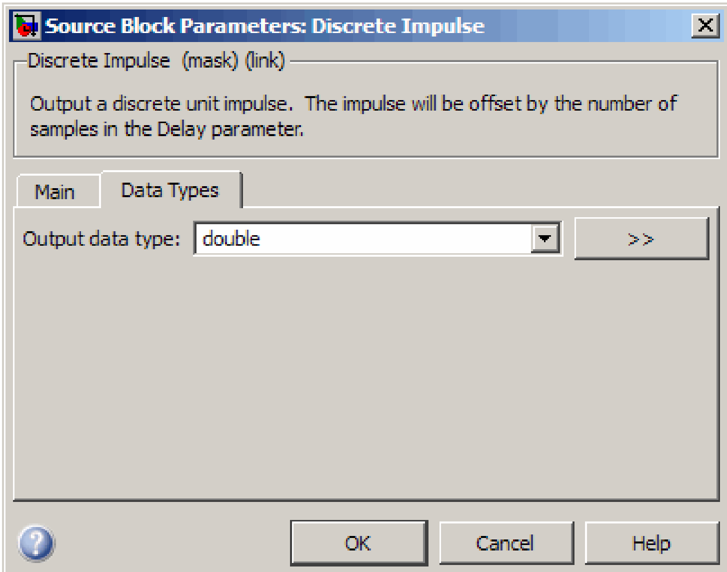
Sample time

The sample period, T_s, of the output signal. The output frame period is $M*T_s$.

Samples per frame

The number of samples, M, in each output frame.

The **Data Types** pane of the Discrete Impulse block dialog appears as follows.



Output data type

Specify the output data type for this block. You can select one of the following:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`. When you select this option, the output data type and scaling matches that of the next downstream block.
- A built in data type, such as `double`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Supported Data Types

- Double-precision floating point

- Single-precision floating point
- Fixed point
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

See Also

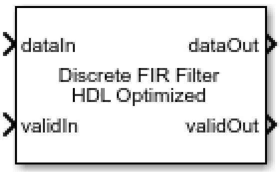
Data Type Conversion	Simulink
Constant	Simulink
`Multiphase Clock`	DSP System Toolbox
`N-Sample Enable`	DSP System Toolbox
`Signal From Workspace`	DSP System Toolbox
`impz`	Signal Processing Toolbox

Introduced before R2006a

Discrete FIR Filter HDL Optimized

Finite impulse response filter—optimized for HDL code generation

Library: DSP System Toolbox / Filtering / Filter Implementations
DSP System Toolbox HDL Support / Filtering



Description

The Discrete FIR Filter HDL Optimized block models FIR filter architectures optimized for HDL code generation. The block is sample based, accepting one scalar at a time. It provides a hardware-friendly interface with optional input and output flow control signals.

The block implements a direct-form systolic FIR filter architecture based on multiply-accumulate operations with pipeline registers. The choice of architecture makes the block well-suited for high-throughput HDL code generation targeted for FPGAs with dedicated DSP blocks. The generated HDL code runs at a high clock rate and can filter new input data on every cycle.

The block provides flexible and efficient resource sharing options. This property results in tradeoffs between throughput and resource utilization. A sharing factor of *N* indicates that the block reduces overall DSP resource utilization by a factor of *N*. In this case, the block can process only input samples that are at least *N* cycles apart. To determine when the block is ready for new input data, use the optional **ready** output port.

To provide a cycle-accurate simulation of the generated HDL code, the block models pipeline registers and resource sharing. These operations cause a delay between valid input data and the corresponding valid output data. The actual latency depends on the number of coefficients and the sharing factor.

Ports

Input

`dataIn` — Input data

real or complex scalar

Input data, specified as a real or complex scalar. When the input data type is integer or fixed point, the block uses fixed-point arithmetic for internal calculations. `double` and `single` are accepted for simulation but not for HDL code generation.

Data Types: `fixed_point` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`
Complex Number Support: Yes

`validIn` — Indicates valid input data (optional)

logical scalar

When **validIn** is `true`, the block captures the input data on **dataIn**. The port is enabled by default and is required when you select DSP resource sharing.

Dependencies

To enable this port, on the **Main** tab, select **Share DSP resources**. Alternatively, if you clear **Share DSP resources**, you can enable this port by selecting **Enable valid input port** on the **Control Ports** tab.

Data Types: `Boolean`

Output

`dataOut` — Filtered output data

real or complex scalar

Filtered output data, returned as a real or complex scalar. When the input data type is floating point, the output data inherits the data type of the input data. When the input data is integer or fixed point, the **Output** parameter on the **Data Types** tab controls the output data type.

Data Types: `fixed_point` | `single` | `double`

Complex Number Support: Yes

validOut — Indicates valid output data
logical scalar

The block sets **validOut** to `true` with each valid output data returned by **dataOut**.

Data Types: `Boolean`

ready — Indicates that the block is ready for new input data (optional)
logical scalar

The block sets **ready** to `true` to indicate that it is ready for new input data on the next cycle.

Dependencies

To enable this port, on the **Control Port** tab, select **Enable ready output port**.

Data Types: `Boolean`

Parameters

Main

Coefficients — Discrete FIR filter coefficients
`[ 0.5, 0.5 ]` (default) | vector of numeric values

Discrete FIR filter coefficients, specified as a vector of numeric values. You can also specify the vector as a workspace variable, or as a call to a filter design function. Complex coefficients are not supported. When the input data type is floating point, the block casts the coefficients to the same data type as the input. When the input data type is integer or fixed point, you can modify coefficient type casting details by setting the **Coefficients** on the **Data Types** tab.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Share DSP resources — Enable resource sharing
off (default) | on

Enable resource sharing to reduce overall DSP resource utilization on the FPGA. When this parameter is selected, specify the **Sharing factor** by which you want to reduce the number of DSPs.

`Sharing factor` — Factor by which the number of DSPs is reduced

`2` (default) | positive integer

Factor by which the number of DSPs is reduced, specified as a positive integer. A sharing factor of *N* indicates that the block reduces DSP resource utilization by a factor of *N*. In this case, the block can process only input samples that are at least *N* cycles apart.

Dependencies

To enable this parameter, on the **Main** tab, select **Share DSP resources**.

Data Types

`Rounding mode` — Rounding mode for type casting the output

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

Rounding mode for type casting the output to the data type specified by **Output**. When the input data type is floating point, the value of **Rounding mode** has no effect. See "Rounding Modes" for more details.

`Overflow mode` — Overflow handling for type casting the output

`Wrap` (default) | `Saturate`

Overflow handling for type casting the output to the data type specified by **Output**. When the input data type is floating point, the value of **Overflow mode** has no effect. See "Overflow Handling" for more details.

`Coefficients` — Data type of discrete FIR filter coefficients

`Inherit: Same word length as input` (default) | `<data type expression>`

The block casts the filter coefficients of the discrete FIR filter to this data type. The quantization uses nearest rounding and saturate overflow modes. When the input data type is floating point, the value of **Coefficients** has no effect.

`Output` — Data type of discrete FIR filter output

`Inherit: Same word length as input` (default) | `Inherit: Inherit via internal rule` | `<data type expression>`

The block casts the output of the discrete FIR filter to this data type. The quantization uses the settings of **Rounding mode** and **Overflow mode**. When the input data type is floating point, the value of the **Output** data type has no effect.

Control Ports

`Enable valid input port` — Enable valid input control signal
on (default) | off

Select this parameter to enable the **validIn** port.

Dependencies

To enable this parameter, on the **Main** tab, clear **Share DSP resources**. When DSP resources are shared, the **validIn** port always shows.

`Enable ready output port` — Enable optional ready control signal
off (default) | on

Select this parameter to enable the **ready** port.

Model Examples

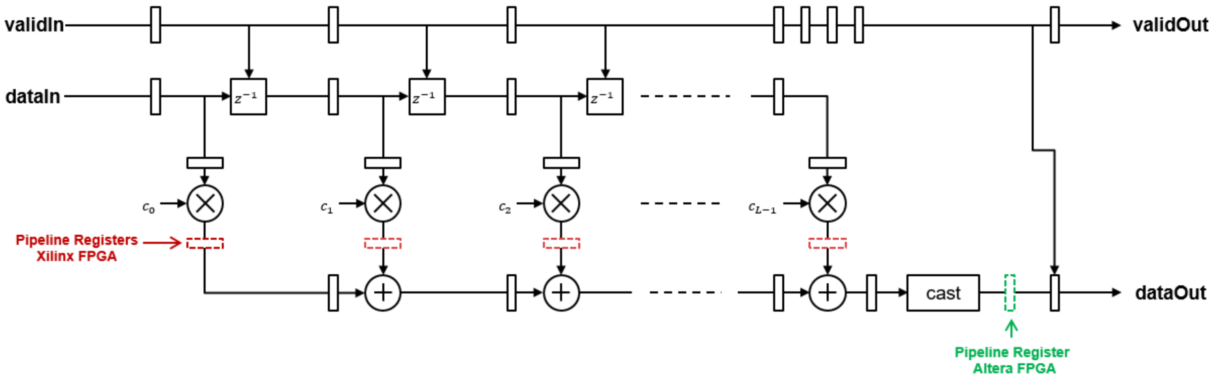
Algorithm

The block implements a direct-form systolic FIR filter architecture based on multiply-accumulate operations with pipeline registers. The simulation models the latency incurred by the pipeline registers of the generated HDL code, so that the latency of the block matches the hardware.

The transfer function has L coefficients. The sharing factor, N, and the target device determine the specifics of the architecture used for simulation and HDL code generation. The implementation takes into account vendor-specific hardware details of the DSP blocks when adding pipeline registers to the architecture. The following diagrams depict the differences between the architecture choices made for FPGA targets from two different hardware vendors.

Fully Parallel FIR Systolic Architecture ($N = 1$)

With no DSP resource sharing, or when resource sharing is of factor 1, the block implements a fully parallel systolic architecture. In this case, the implementation utilizes L DSP (multiply-add) blocks and is heavily pipelined, resulting in high throughput. The filter can accept a new input sample on every cycle. The latency of the block is $L + 4$.



Resource Utilization and Performance

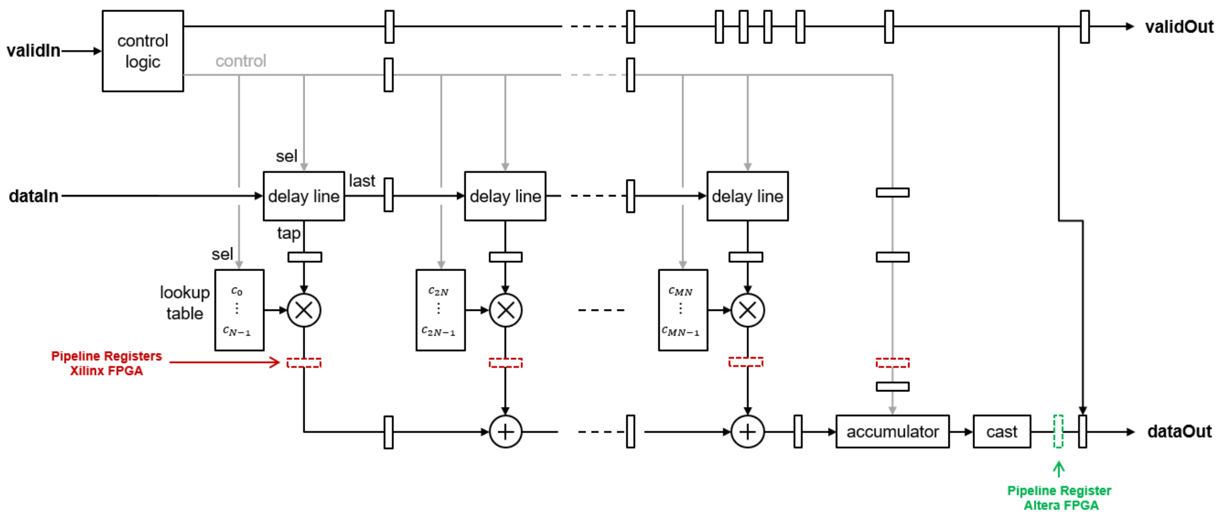
The following table shows post synthesis resource utilization for the HDL code generated from the “Fully Parallel Systolic FIR Filter Implementation” example. The implementation is for a 25-tap FIR filter with 16 bit input and 16 bit coefficients. The synthesis targets a Xilinx Virtex-6 (XC6VLX240T-1FF1156) FPGA. The reset type is synchronous and minimize clock enables is switched on.

Resource	Uses
FFS	830
Xilinx LogiCORE DSP48	25

After place and route, the maximum clock frequency of the design is 421 MHz.

Partly Serial FIR Systolic Architecture ($1 < N < L$)

When DSP resource sharing is of factor N, where $1 < N < L$, the block implements a partly serial systolic architecture. In this case, the implementation uses `M = ceil(L/N)` systolic cells. Each cell consists of a delay line, a coefficient lookup table, and a DSP (multiply-add) block. The coefficients are spread across the M lookup tables. The computation performed by each DSP block is serialized. Input samples to the block must be at least N cycles apart. The latency of the block is M + `ceil(`L`/`M`)` + 5.



Resource Utilization and Performance

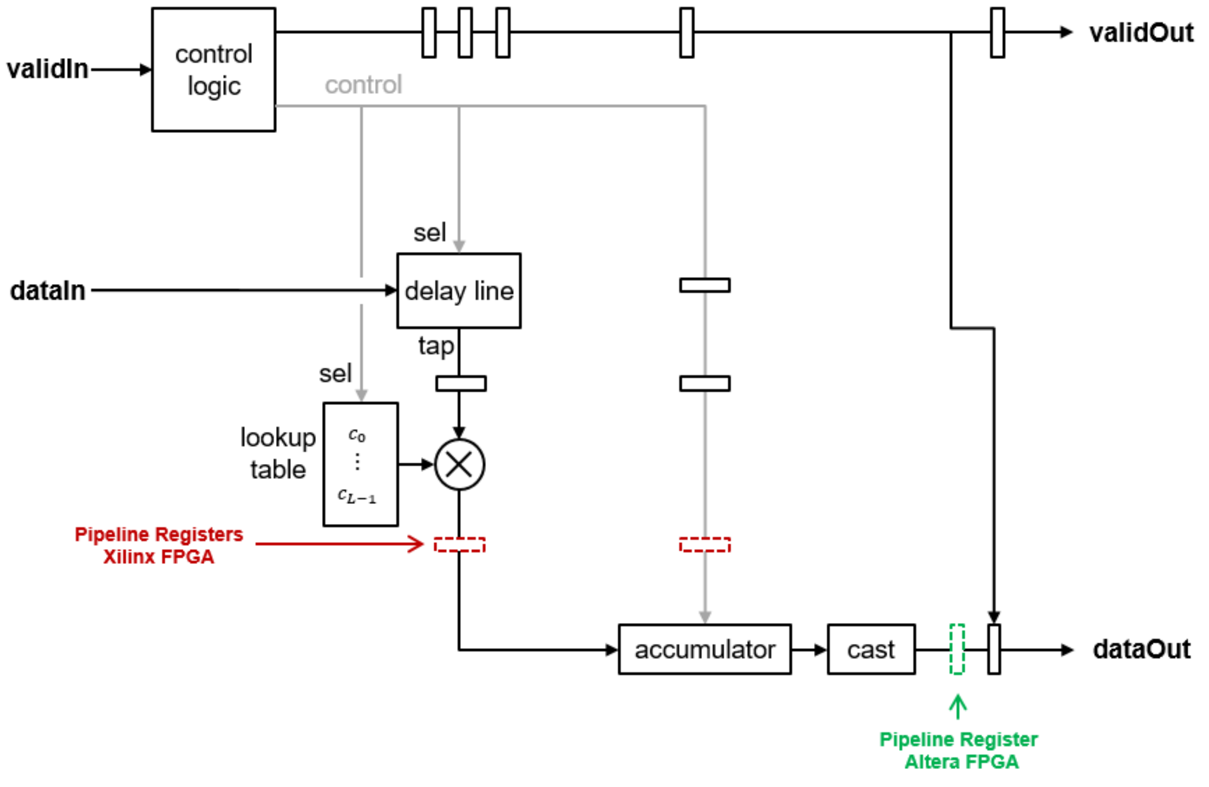
The following table shows post synthesis resource utilization for the HDL code generated from the "Partly Serial Systolic FIR Filter Implementation" example. The implementation is for a 32-tap FIR filter with 16 bit input, 16 bit coefficients and a sharing factor of 8. The synthesis targets a Xilinx Virtex-6 (XC6VLX240T-1FF1156) FPGA. The reset type is synchronous and minimize clock enables is switched on.

Resource	Uses
LUT	192
FFS	606
Xilinx LogiCORE DSP48	5

After place and route, the maximum clock frequency of the design is 376 MHz.

Fully Serial FIR Systolic Architecture ($N \geq L$)

When DSP resource sharing is of factor N, where $N \geq L$, the block implements a fully serial systolic architecture. In this case, the implementation utilizes a single DSP (multiply-add) block with a delay line and a lookup table for all L coefficients. Input samples must be at least N cycles apart. The latency of the block is $L + 5$.



See Also

See Also

System Objects

dsp.HDLFIRFilter | dsp.FIRFilter

Blocks

`Discrete FIR Filter`

Topics

HDL Code Generation Support for DSP System Toolbox

Introduced in R2017a

Discrete Transfer Function Estimator

Compute estimate of frequency-domain transfer function of system

Library

Estimation / Power Spectrum Estimation

`dspspect3`

Description

The Discrete Transfer Function Estimator block estimates the frequency-domain transfer function of a system using the Welch's method of averaged modified periodograms.

The block takes two inputs, x and y. x is the system input signal and y is the system output signal. x and y must have the same dimensions. For 2D inputs, the block treats each column as an independent channel. The first dimension is the length of the channel. The second dimension is the number of channels. The block treats 1D inputs as one channel. The sample rate of the block is equal to $1/T$. T is the sample time of the inputs to the block.

The block buffers the input data into overlapping segments. You can set the length of the data segment and the amount of data overlap through the parameters set in the block dialog box.

The block first applies a window function to the two inputs, x and y, and then scales them by the window power. It takes the FFT of each signal, calling them X and Y. The block calculates P_{xx} which is the square magnitude of the FFT, X. The block then calculates P_{yx} which is X multiplied by the conjugate of Y. The output transfer function estimate, H, is calculated by dividing P_{yx} by P_{xx}.

Parameters

Window length source

Source of the window length value. You can set this parameter to:

- `Same as input frame length` (default) — Window length is set to the frame size of the input.
- `Specify on dialog` — Window length is the value specified in **Window length**.

This parameter is nontunable.

Window length

Length of the window, in samples, used to compute the spectrum estimate, specified as a positive integer scalar greater than `2`. This parameter applies when you set **Window length source** to `Specify on dialog`. The default is `1024`. This parameter is nontunable.

Window Overlap (%)

Percentage of overlap between successive data windows, specified as a scalar in the range [`0,100`). The default is `0`. This parameter is nontunable.

Number of spectral averages

Specify the number of spectral averages. The Transfer Function Estimator block computes the current estimate by averaging the last *N* estimates. *N* is the number of spectral averages. It can be any positive integer scalar, and the default is `1`.

FFT length source

Specify the source of the FFT length value. It can be one of `Auto` (default) or `Property`. When the source of the FFT length is set to `Auto`, the Transfer Function Estimator block sets the FFT length to the input frame size. When the source of the FFT length is set to `Property`, you specify the FFT length in the **FFT length** parameter.

FFT length

Specify the length of the FFT that the Transfer Function Estimator block uses to compute spectral estimates. It can be any positive integer scalar, and the default is 128.

Window function

Specify a window function for the Transfer Function Estimator block. Possible values are:

- `Hann` (default)
- `Rectangular`
- `Chebyshev`
- `Flat Top`
- `Hamming`
- `Kaiser`

Sidelobe attenuation of window (dB)

Specify the sidelobe attenuation of the window. It can be any real positive scalar value in decibels (dB). The default is `60`.

Note: This parameter is visible only when **Window function** is set to `Kaiser` or `Chebyshev`.

Frequency range

Specify the frequency range of the transfer function estimate.

- `centered` (default)

  When you set the frequency range to `centered`, the Transfer Function Estimator block computes the centered two-sided transfer function of the real or complex input signals, x and y.
- `onesided`

  When you set the frequency range to `onesided`, the Transfer Function Estimator block computes the one-sided transfer function of real input signals, x and y.
- `twosided`

  When you set the frequency range to `twosided`, the Transfer Function Estimator block computes the two-sided transfer function of the real or complex input signals, x and y.

Output magnitude squared coherence estimate

Select this check box to compute and output the magnitude squared coherence estimate using Welch's averaged, modified periodogram method. The magnitude squared coherence estimate indicates how well two inputs correspond to each other at each frequency.

Simulate using

Type of simulation to run. You can set this parameter to:

- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Supported Data Types

The Discrete Transfer Function Estimator block supports real and complex inputs.

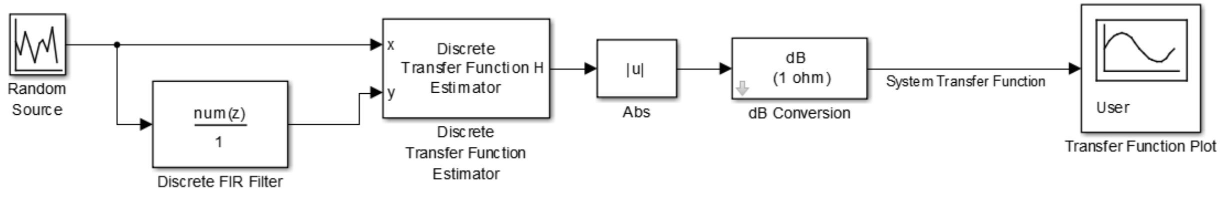
Port	Supported Data Type
x	• Double-precision floating point • Single-precision floating point
y	• Double-precision floating point • Single-precision floating point
Output, *H*	• Double-precision floating point • Single-precision floating point

Examples

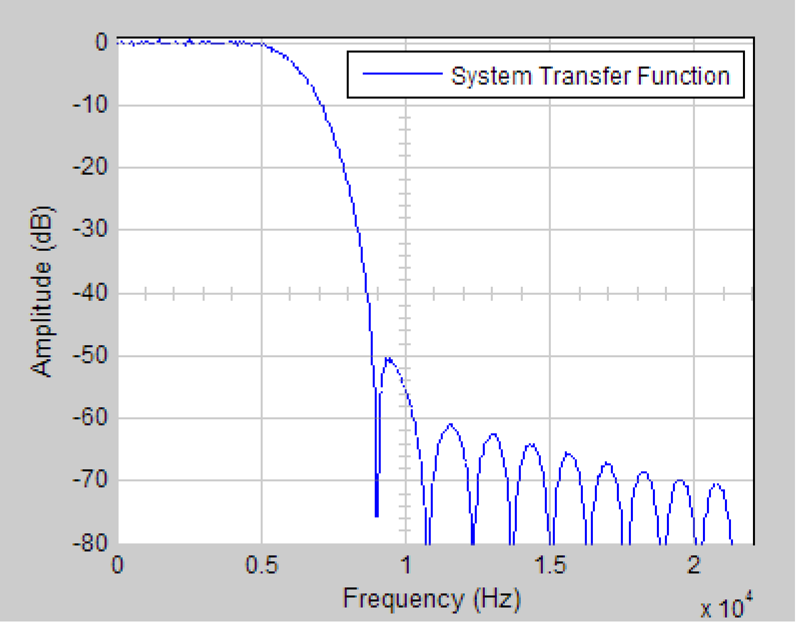
This example shows how to use the Discrete Transfer Function Estimator block to estimate the frequency-domain transfer function of a system.

The Random Source block represents the system input signal. The sample rate of the system input is 44.1 KHz. The Random Source input passes through a low-pass filter with a normalized cutoff frequency of 0.3. The filtered signal represents the system output signal. Because the Discrete Transfer Function Estimator block outputs complex values, take the magnitude of the output to see a plot of the transfer function estimate.

To view this example, execute `ex_discrete_transfer_function_estimator` in MATLAB Command prompt.



The transfer function plot displays the system transfer function, a low-pass filter that matches the frequency response of the Discrete FIR Filter block.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- When the FFT length is not a power of two, the executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- When the FFT length is a power of two, you can generate standalone C and C++ code from this block.

See Also

See Also

dsp.TransferFunctionEstimator | `Cross-Spectrum Estimator` | `Periodogram` | `Spectrum Analyzer`

Introduced in R2014a

Display

Show value of input

Library

Sinks

`dspsnks4`

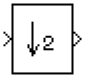
Description

The Display block is an implementation of the Simulink Display block. See Display for more information.

Introduced in R2008a

Downsample

Resample input at lower rate by deleting samples



Library

Signal Operations

`dspsigops`

Description

The Downsample block decreases the sampling rate of the input by deleting samples. When the block performs frame-based processing, it resamples the data in each column of the M_i-by-N input matrix independently. When the block performs sample-based processing, it treats each element of the input as a separate channel and resamples each channel of the input array across time. The resample rate is K times lower than the input sample rate, where K is the integer you specify for the **Downsample factor** parameter. The Downsample block resamples the input by discarding K–1 consecutive samples following each sample that is passed through to the output.

The **Sample offset** parameter delays the output samples by an integer number of sample periods, D, where $0 \leq D \leq (K–1)$, so that any of the K possible output phases can be selected. For example, when you downsample the sequence 1, 2, 3, ... by a factor of 4, you can select from the following four phases.

Input Sequence	Sample Offset, D	Output Sequence (K=4)
`1,2,3,...`	0	`1,5,9,13,17,21,25,29,...`
`1,2,3,...`	1	`0,2,6,10,14,18,22,26,...`
`1,2,3,...`	2	`0,3,7,11,15,19,23,27,...`
`1,2,3,...`	3	`0,4,8,12,16,20,24,28,...`

The initial zero in each of the latter three output sequences in the table is a result of the default zero **Initial conditions** parameter setting for this example. See "Latency" on page 2-611 for more information on the **Initial conditions** parameter.

This block supports triggered subsystems when you set the **Rate options** parameter to `Enforce single-rate processing`.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the input can be an N-D array. The Downsample block treats each element of the input as a separate channel, and resamples each channel of the input over time. The block downsamples the input array by discarding K–1 samples following each sample that it passes through to the output. The input and output sizes of the Downsample block are identical.

The **Rate options** parameter specifies how the block adjusts the rate at the output to accommodate the reduced number of samples. There are two available options:

- `Allow multirate processing`

  When you select `Allow multirate processing`, the sample period of the output is K times longer than the input sample period ($T_{so} = KT_{si}$).

- `Enforce single-rate processing`

  When you select `Enforce single-rate processing`, the block forces the output sample rate to match the input sample rate ($T_{so} = T_{si}$) by repeating every Kth input sample K times at the output. In this mode, the behavior of the block is similar to the operation of a Sample and Hold block with a repeating trigger event of period KT_{si}.

Frame-Based Processing

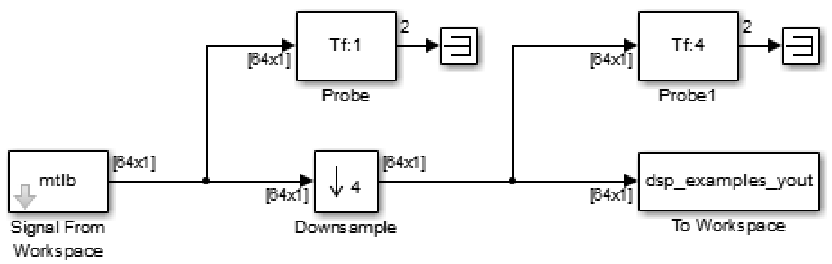
When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each of the N input columns as an individual channel containing M_i sequential time samples. The block downsamples each channel independently by discarding K–1 rows of the input matrix following each row that it passes through to the output.

The **Rate options** parameter specifies how the block adjusts the rate at the output to accommodate the reduced number of samples. There are two available options:

- `Allow multirate processing`

  The block generates the output at the slower (downsampled) rate by using a proportionally longer frame *period* at the output port than at the input port. For downsampling by a factor of K, the output frame period is K times longer than the input frame period ($T_{fo} = KT_{fi}$), but the input and output frame sizes are equal.

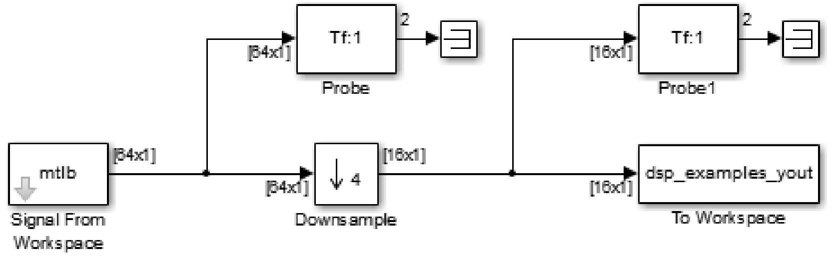
  The ex_downsample_ref1 model shows a single-channel input with a frame period of 1 second being downsampled by a factor of 4 to a frame period of 4 seconds. The input and output frame sizes are identical.



- `Enforce single rate processing`

  The block generates the output at the slower (downsampled) rate using a proportionally smaller frame *size* than the input. For downsampling by a factor of K, the output frame size is K times smaller than the input frame size ($M_o = M_i/K$), but the input and output frame rates are equal.

  The ex_downsample_ref2 model shows a single-channel input with a frame size of 64 being downsampled by a factor of 4 to a frame size of 16. The input and output frame rates are identical.



Latency

The Downsample block has *zero-tasking latency* in the following cases:

- The **Downsample factor** parameter, K, is `1`.
- The **Input processing** parameter is set to `Columns as channels (frame based)` and the **Rate options** parameter is set to `Enforce single-rate processing`.
- The **Input processing** parameter is set to `Columns as channels (frame based)` and the input frame size is equal to one.
- The **Input processing** parameter is set to `Elements as channels (sample based)`, and **Sample offset** parameter, D, is `0`.

Zero-tasking latency means that the block propagates input sample D+1 (received at t=0) as the first output sample, followed by input sample D+1+K, input sample D+1+2K, and so on. When there is zero-tasking latency, the block ignores the value of the **Initial conditions** parameter.

In all other cases, the latency is nonzero:

- When the **Input processing** parameter is set to `Elements as channels (sample based)`, the latency is one sample.
- When the **Input processing** parameter is set to `Columns as channels (frame based)` and the input frame size is greater than one, the latency is one frame.

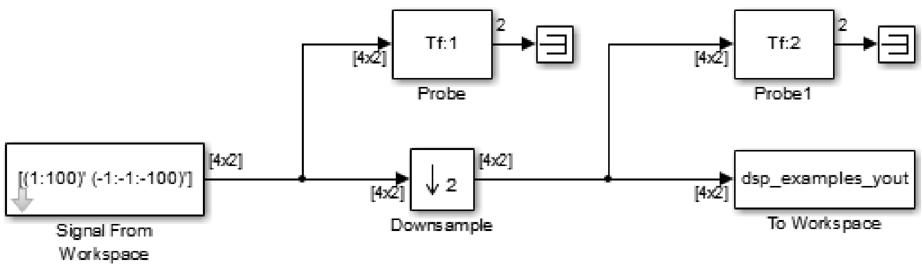
In all cases of *one-sample latency*, the initial condition for each channel appears as the first output sample. Input sample D+1 appears as the second output sample for each channel, followed by input sample D+1+K, input sample D+1+2K, and so on. The **Initial conditions** parameter can be an array of the same size as the input, or a scalar to be applied to all signal channels.

In all cases of *one-frame latency*, the M_i rows of the initial condition matrix appear in sequence as the first M_i output rows. Input sample D+1 (i.e, row D+1 of the input matrix) appears in the output as sample M_i+1, followed by input sample D+1+K, input sample D+1+2K, and so on. The **Initial conditions** value can be an M_i-by-N matrix containing one value for each channel, or a scalar to be repeated across all elements of the M_i-by-N matrix. See the following example for an illustration of this case.

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Time-Based Scheduling and Code Generation" (Simulink Coder).

Examples

Open the ex_downsample_ref3 model.



Run the model and look at the output, `yout`. The first few samples of each channel are as follows:

```
yout =
    11    -11
    12    -12
    13    -13
    14    -14
     2     -2
     4     -4
     6     -6
     8     -8
    10    -10
    12    -12
    14    -14
```

You can see from the two Probe blocks that there are at least two distinct frame rates in this model. Because you ran this model in multirate, multitasking mode, the first row of the initial condition matrix appears as the first output sample, followed by the other three initial condition rows. The second row of the first input matrix (row D+1, where D is the **Sample offset**) appears in the output as sample 5 (sample M_i+1, where M_i is the input frame size).

Parameters

Downsample factor

The integer factor, K, by which to decrease the input sample rate.

Sample offset

The sample offset, D, which must be an integer in the range [0, K–1].

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel. See “Frame-Based Processing” on page 2-609 for more information
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel. See “Sample-Based Processing” on page 2-609 for more information.

Rate options

Specify the method by which the block should downsample the input. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate.
- `Allow multirate processing` — When you select this option, the block downsamples the signal such that the output sample rate is K times slower than the input sample rate.

Initial conditions

The value with which the block is initialized for cases of nonzero latency. You can specify a scalar, or an array of the same size as the input. This parameter appears only when you set the **Rate options** parameter to `Allow multirate processing`.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

For more information on implementations, properties, and restrictions for HDL code generation, see Downsample.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

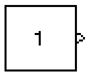
See Also

`FIR Decimation`	DSP System Toolbox
`FIR Rate Conversion`	DSP System Toolbox
`Repeat`	DSP System Toolbox
`Sample and Hold`	DSP System Toolbox
`Upsample`	DSP System Toolbox

Introduced before R2006a

DSP Constant (Obsolete)

Generate discrete- or continuous-time constant signal



Library

Sources

`dspobslib`

Description

Note The DSP Constant block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Constant block.

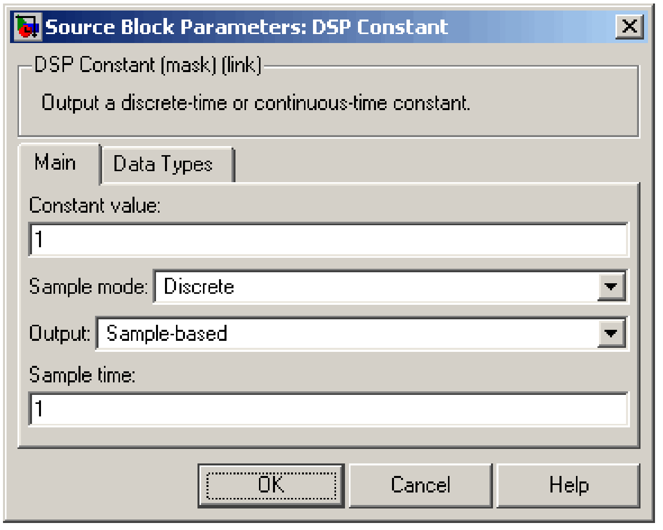
The DSP Constant block generates a signal whose value remains constant throughout the simulation. The **Constant value** parameter specifies the constant to output, and can be any valid MATLAB expression that evaluates to a scalar, vector, or matrix.

When **Sample mode** is set to `Continuous`, the output is a continuous-time signal. When **Sample mode** is set to `Discrete`, the **Sample time** parameter is visible, and the signal has the discrete output period specified by the **Sample time** parameter.

You can set the output signal to `Frame-based`, `Sample-based`, or `Sample-based (interpret vectors as 1-D)` with the **Output** parameter.

Dialog Box

The **Main** pane of the DSP Constant block dialog box appears as follows.



Constant value

Specify the constant to generate. This parameter is Tunable (Simulink); values entered here can be tuned, but their dimensions must remain fixed.

When you specify any data type information in this field, it is overridden by the value of the **Output data type** parameter in the **Data Types** pane, unless you select `Inherit from 'Constant value'`.

Sample mode

Specify the sample mode of the output, `Discrete` for a discrete-time signal or `Continuous` for a continuous-time signal.

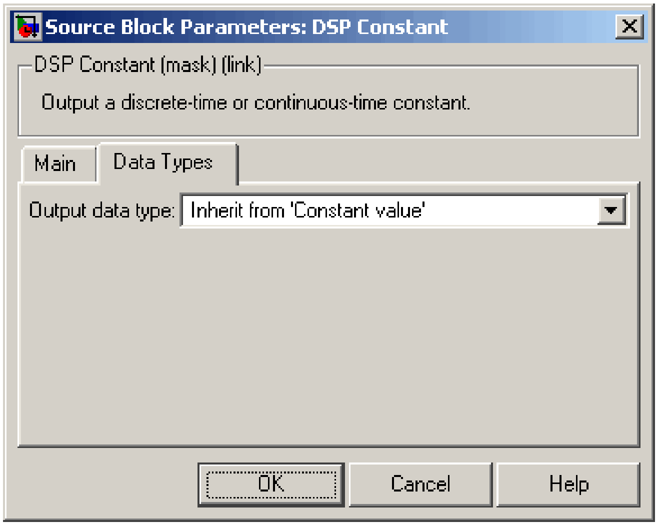
Output

Specify whether the output is `Sample-based (interpret vectors as 1-D)`, `Sample-based`, or `Frame-based`. When you select `Sample-based` and the output is a vector, its dimension is constrained to match the **Constant value** dimension (row or column). When you select `Sample-based (interpret vectors as 1-D)`, however, the output has no specified dimensionality.

Sample time

Specify the discrete sample period for sample-based outputs. When you select `Frame-based` for the **Output** parameter, this parameter is named **Frame period**, and is the discrete frame period for the frame-based output. This parameter is only visible when you select `Discrete` for the **Sample mode** parameter.

The **Data Types** pane of the DSP Constant block dialog box appears as follows.



Output data type

Specify the output data type in one of the following ways:

- Choose one of the built-in data types from the list.
- Choose `Fixed-point` to specify the output data type and scaling in the **Signed**, **Word length**, **Set fraction length in output to**, and **Fraction length** parameters.
- Choose `User-defined` to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to** , and **Fraction length** parameters.
- Choose `Inherit from 'Constant value'` to set the output data type and scaling to match the values of the **Constant value** parameter in the **Main** pane.
- Choose `Inherit via back propagation` to set the output data type and scaling to match the following block.

The value of this parameter overrides any data type information specified in the **Constant value** parameter in the **Main** pane, except when you select `Inherit from 'Constant value'`.

Signed

Select to output a signed fixed-point signal. Otherwise, the signal is unsigned. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

Word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

User-defined data type

Specify any built-in or fixed-point data type. You can specify fixed-point data types using the following Fixed-Point Designer functions: `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac`. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.
- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter, or when you select `User-defined` and the specified output data type is a fixed-point data type.

Fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Output data type** parameter and `User-defined` for the **Set fraction length in output to** parameter.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Boolean
- 8-, 16-, and 32-bit signed integers

- 8-, 16-, and 32-bit unsigned integers

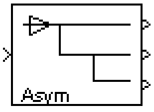
See Also

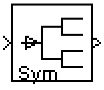
Constant	Simulink
Signal From Workspace	DSP System Toolbox

Introduced in R2008b

DWT

Discrete wavelet transform (DWT) of input or decompose signals into subbands with smaller bandwidths and slower sample rates





Library

Transforms

`dspxfrm3`

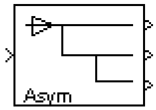
Description

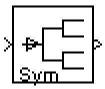
The DWT block is the same as the Dyadic Analysis Filter Bank block in the Multirate Filters library, but with different default settings. See the `Dyadic Analysis Filter Bank` block reference page for more information.

Introduced before R2006a

Dyadic Analysis Filter Bank

Decompose signals into subbands with smaller bandwidths and slower sample rates or compute discrete wavelet transform (DWT)





Library

Filtering / Multirate Filters

`dspmlti4`

Description

Note: This block always interprets input signals as frames. The frame size of the input signal must be a multiple of 2^n, where `n` is the value of the **Number of levels** parameter. The block decomposes the input signal into either n+1 or 2^n subbands. To decompose signals with a frame size that is not a multiple of 2^n, use the `Two-Channel Analysis Subband Filter` block. (You can connect multiple copies of the Two-Channel Analysis Subband Filter block to create a multilevel dyadic analysis filter bank.)

You can configure this block to compute the Discrete Wavelet Transform (DWT) or decompose a broadband signal into a collection of subbands with smaller bandwidths and slower sample rates. The block uses a series of highpass and lowpass FIR filters to repeatedly divide the input frequency range, as illustrated in “Wavelet Filter Banks” on page 2-625 (the Asymmetric one).

You can specify the filter bank's highpass and lowpass filters by providing vectors of filter coefficients. You can do so directly on the block mask, or, if you have a Wavelet

Toolbox™ license, you can specify wavelet-based filters by selecting a wavelet from the **Filter** parameter. You must set the filter bank structure to asymmetric or symmetric, and specify the number of levels in the filter bank.

For the same input, the DWT configuration of this block does not produce the same results as the Wavelet Toolbox `dwt` function. Because DSP System Toolbox is designed for real-time implementation and Wavelet Toolbox is designed for analysis, the products handle boundary conditions and filter states differently. To make the output of the `dwt` function match the DWT output of this block, complete the following steps:

1. Set the boundary condition of the `dwt` function to zero-padding. To do so, type `dwtmode('zpd')` at the MATLAB command line.
2. To match the latency of the block (implemented using FIR filters), add zeros to the input of the `dwt` function. The number of zeros you add must be equal to the half-length of the filter.

Input Requirements

- Input must be a vector or matrix.
- The input frame size must be a multiple of 2^n, where n is the number of filter bank levels. For example, a frame size of 16 would be appropriate for a three-level tree (16 is a multiple of 2^3).
- The block always treats input signals as frames and operates along the columns.

For an illustration of why the above input requirements exist, see the figure Outputs of a 3-Level Asymmetric Dyadic Analysis Filter Bank.

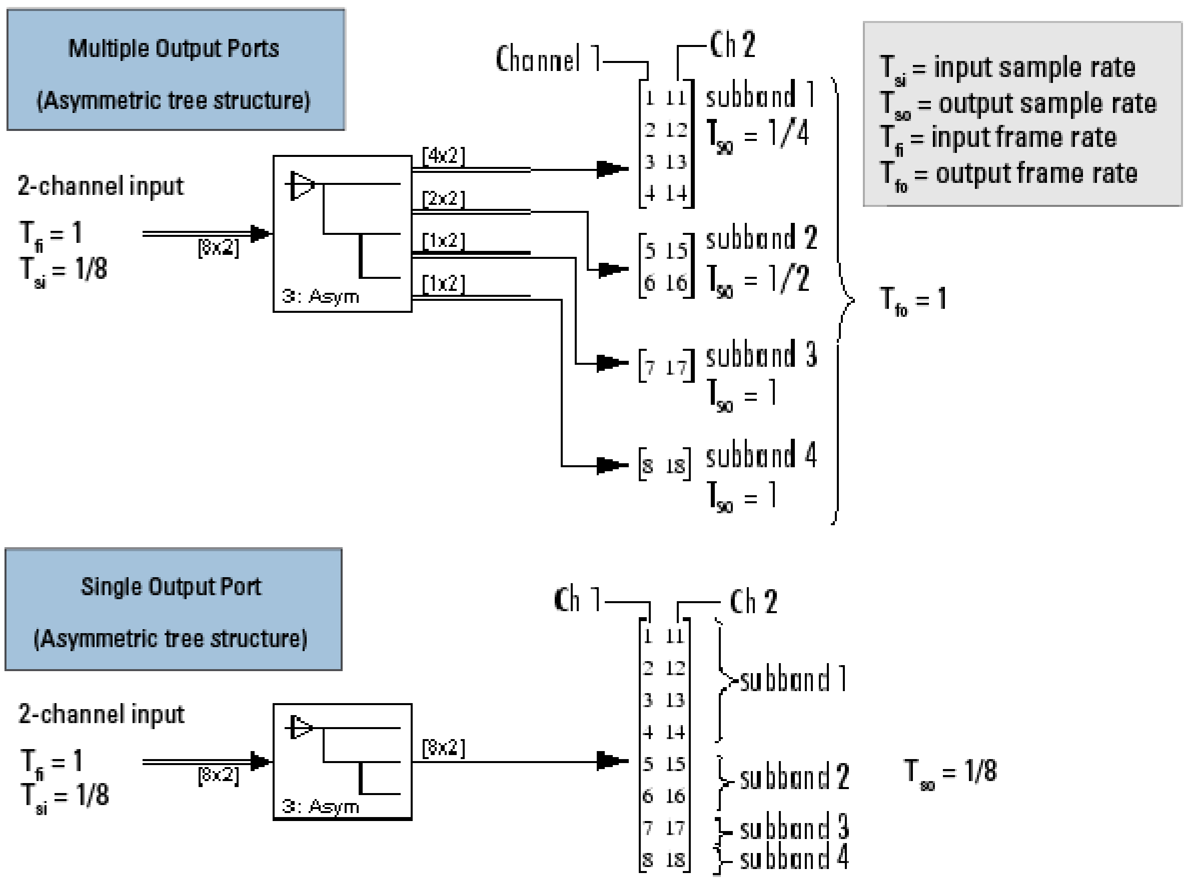
Output Characteristics

The output characteristics vary depending on the block's parameter settings, as summarized in the following list and figure:

- **Number of levels** parameter set to n
- **Tree structure** parameter setting:
  - `Asymmetric` — Block produces n+1 output subbands
  - `Symmetric` — Block produces 2^n output subbands
- **Output** parameter setting can be `Multiple ports` or `Single port`. When you set the **Output** parameter to `Single port`, the block outputs one vector or matrix of

concatenated subbands. The following figure illustrates the difference between the two settings for a 3-level asymmetric dyadic analysis filter bank. For an explanation of the illustrated output characteristics, see the table Output Characteristics for an n-Level Dyadic Analysis Filter Bank.

For more information about the filter bank levels and structures, see “Dyadic Analysis Filter Banks”.



Outputs of a 3-Level Asymmetric Dyadic Analysis Filter Bank

The following table summarizes the different output characteristics of the block when it is set to output from single or multiple ports.

Output Characteristics for an n-Level Dyadic Analysis Filter Bank

	Single Output Port	Multiple Output Ports
Output Description	Block concatenates all the subbands into one vector or matrix, and outputs the concatenated subbands from a single output port. Each output column contains subbands of the corresponding input channel.	Block outputs each subband from a separate output port. The topmost port outputs the subband with the highest frequencies. Each output column contains a subband for the corresponding input channel.
Output Frame Rate	*Not applicable*	Same as input frame rate (However, the output frame sizes can vary, so the output sample rates can vary.)
Output Dimensions (Frame Size)	Same number of rows and columns as the input.	The output has the same number of columns as the input. The number of output rows is the output frame size. For an input with frame size M_i output y_k has frame size $M_{o,k}$: • `Symmetric` — All outputs have the frame size, $M_i / 2^n$. • `Asymmetric` — The frame size of each output (except the last) is half that of the output from the previous level. The outputs from the last two output ports have the same frame size since they originate from the same level in the filter bank. $$M_{o,k} = \begin{cases} M_i / 2^k & (1 \le k \le n) \\ M_i / 2^n & (k = n+1) \end{cases}$$
Output Sample Rate	Same as input sample rate.	Though the outputs have the same frame rate as the input, they have different frame sizes than

	Single Output Port	Multiple Output Ports
		the input. Thus, the output sample rates, F_{so}, k, are different from the input sample rate, F_{si}: • `Symmetric` — All outputs have the sample rate F_{si} / 2^n. • `Asymmetric` — $F_{so,k} = \begin{cases} F_{si}/2^k & (1 \le k \le n) \\ F_{si}/2^n & (k = n+1) \end{cases}$

Wavelet Filter Banks

- "Filter Banks"

Filter Bank Filters

You must specify the highpass and lowpass filters in the filter bank by setting the **Filter** parameter to one of the following options:

- `User defined` — Allows you to explicitly specify the filters with two vectors of filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters. The block uses the same lowpass and highpass filters throughout the filter bank. The two filters should be halfband filters, where each filter passes the frequency band that the other filter stops.
- Wavelet such as `Biorthogonal` or `Daubechies` — The block uses the specified wavelet to construct the lowpass and highpass filters using the Wavelet Toolbox `wfilters` function. Depending on the wavelet, the block might enable either the **Wavelet order** or **Filter order [synthesis / analysis]** parameter. (The latter parameter allows you to specify different wavelet orders for the analysis and synthesis filter stages.) You must have a Wavelet Toolbox license to use wavelets.

Specifying Filters with the Filter Parameter and Related Parameters

Filter	Sample Setting for Related Filter Specification Parameters	Corresponding Wavelet Function Syntax
User-defined	Filters based on Daubechies wavelets with wavelet order `3`:	None

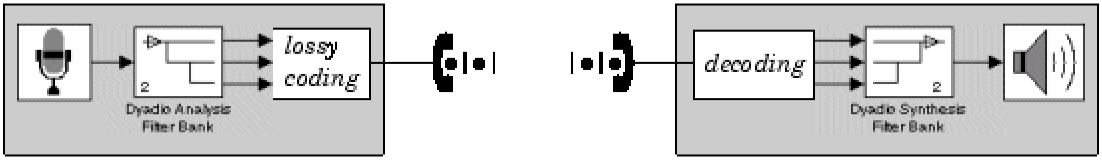
Filter	Sample Setting for Related Filter Specification Parameters	Corresponding Wavelet Function Syntax
	• **Lowpass FIR filter coefficients** = `[0.0352 -0.0854 -0.1350 0.4599 0.8069 0.3327]` • **Highpass FIR filter coefficients** = `[-0.3327 0.8069 -0.4599 -0.1350 0.0854 0.0352]`	
Haar	None	`wfilters('haar')`
Daubechies	**Wavelet order** = 4	`wfilters('db4')`
Symlets	**Wavelet order** = 3	`wfilters('sym3')`
Coiflets	**Wavelet order** = 1	`wfilters('coif1')`
Biorthogonal	**Filter order [synthesis / analysis]** = [3/1]	`wfilters('bior3.1')`
Reverse Biorthogonal	**Filter order [synthesis / analysis]** = [3/1]	`wfilters('rbio3.1')`
Discrete Meyer	None	`wfilters('dmey')`

Examples

Wavelets

The primary application for dyadic analysis filter banks and dyadic synthesis filter banks is coding for data compression using wavelets.

At the transmitting end, the output of the dyadic analysis filter bank is fed to a lossy compression scheme, which typically assigns the number of bits for each filter bank output in proportion to the relative energy in that frequency band. This represents the more powerful signal components by a greater number of bits than the less powerful signal components.



At the receiving end, the transmission is decoded and fed to a dyadic synthesis filter bank to reconstruct the original signal. The filter coefficients of the complementary analysis and synthesis stages are designed to cancel aliasing introduced by the filtering and resampling.

See “Calculate Channel Latencies Required for Wavelet Reconstruction” for an example using the Dyadic Analysis and Dyadic Synthesis Filter Bank blocks.

Examples

See the floating-point frame-based version of the DSP System Toolbox Wavelet Reconstruction and Noise Reduction example, which uses the Dyadic Analysis Filter Bank and Dyadic Synthesis Filter Bank blocks.

Parameters

The parameters displayed in the block dialog vary depending on the setting of the **Filter** parameter. Only some of the parameters described below are visible in the dialog box at any one time.

Filter

The type of filter used to determine the high- and low-pass FIR filters in the filter bank:

Select `User defined` to explicitly specify the filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.

Select a wavelet such as `Biorthogonal` or `Daubechies` to specify a wavelet-based filter. The block uses the Wavelet Toolbox `wfilters` function to construct the filters. Extra parameters such as **Wavelet order** or **Filter order [synthesis / analysis]** might become enabled. For a list of the supported wavelets, see Specifying Filters with the Filter Parameter and Related Parameters.

Lowpass FIR filter coefficients

A vector of filter coefficients (descending powers of z) that specifies coefficients used by all the lowpass filters in the filter bank. This parameter is enabled when you set **Filter** to `User defined`. The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. The default values of this parameter specify a filter based on a Daubechies wavelet with wavelet order `3`.

Highpass FIR filter coefficients

A vector of filter coefficients (descending powers of z) that specifies coefficients used by all the highpass filters in the filter bank. This parameter is enabled when you set **Filter** to `User defined`. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. The default values of this parameter specify a filter based on a Daubechies wavelet with wavelet order `3`.

Wavelet order

The order of the wavelet selected in the **Filter** parameter. This parameter is enabled only when you set **Filter** to certain types of wavelets, as shown in the Specifying Filters with the Filter Parameter and Related Parameters table.

Filter order [synthesis / analysis]

The order of the wavelet for the synthesis and analysis filter stages. For example, when you set the **Filter** parameter to `Biorthogonal` and set the **Filter order [synthesis / analysis]** parameter to `[2 / 6]`, the block calls the `wfilters` function with input argument `'bior2.6'`. This parameter is enabled only when you set **Filter** to certain types of wavelets, as shown in Specifying Filters with the Filter Parameter and Related Parameters.

Number of levels

The number of filter bank levels. An n-level asymmetric structure has n+1 outputs, and an n-level symmetric structure has 2^n outputs, as shown in “Wavelet Filter Banks” on page 2-625. The block's icon changes depending on the value of this parameter.

The default setting of this parameter is `2`.

Tree structure

The structure of the filter bank: `Asymmetric`, or `Symmetric`. See “Wavelet Filter Banks” on page 2-625.

The default setting of this parameter is `Asymmetric` for the Dyadic Analysis Filter Bank block, and `Symmetric` for the DWT block.

Output

Set to `Multiple ports` to output each output subband on a separate port (the topmost port outputs the subband with the highest frequency band). Set to `Single port` to concatenate the subbands into one vector or matrix and output the concatenated subbands on a single port. For more information, see "Output Characteristics" on page 2-622.

The default setting of this parameter is `Multiple ports` for the Dyadic Analysis Filter Bank block, and `Single port` for the DWT block.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

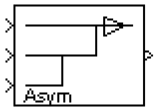
See Also

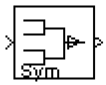
`DWT`	DSP System Toolbox
`Dyadic Synthesis Filter Bank`	DSP System Toolbox
`Two-Channel Analysis Subband Filter`	DSP System Toolbox

Introduced before R2006a

Dyadic Synthesis Filter Bank

Reconstruct signals from subbands with smaller bandwidths and slower sample rates or compute inverse discrete wavelet transform (IDWT)





Library

Filtering / Multirate Filters

`dspmlti4`

Description

Note This block always does frame-based processing, and its inputs must be of certain sizes. To use input subbands that do not fit the criteria of this block, use the `Two-Channel Synthesis Subband Filter` block. (You can connect multiple copies of the Two-Channel Synthesis Subband Filter block to create a multilevel dyadic synthesis filter bank.)

You can configure this block to compute the inverse discrete wavelet transform (IDWT) or reconstruct a signal from subbands with smaller bandwidths and slower sample rates. When the block computes the inverse discrete wavelet transform (IDWT) of the input, the output has the same dimensions as the input. Each column of the output is the IDWT of the corresponding input column. When reconstructing a signal, the block uses a series of highpass and lowpass FIR filters to reconstruct the signal from the input subbands, as illustrated in “Wavelet Filter Banks” on page 2-635 (the Asymmetric one). The reconstructed signal has a wider bandwidth and faster sample rate than the input subbands.

You can specify the filter bank's highpass and lowpass filters by providing vectors of filter coefficients. You can do so directly on the block mask, or, if you have a Wavelet Toolbox license, you can specify wavelet-based filters by selecting a wavelet from the **Filter** parameter. You must set the filter bank structure to asymmetric or symmetric, and specify the number of levels in the filter bank.

When you set the **Input** parameter to `Multiple ports`, you must provide each subband to the block through a different input port as a vector or matrix. You should input the highest frequency band through the topmost port. When you set the **Input** parameter to `Single port`, the block input must be a vector or matrix of concatenated subbands.

Note To use a dyadic synthesis filter bank to perfectly reconstruct the output of a dyadic analysis filter bank, the number of levels and tree structures of both filter banks *must* be the same. In addition, the filters in the synthesis filter bank *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction is not perfect.

This block automatically computes wavelet-based perfect reconstruction filters when the wavelet selection in the **Filter** parameter of this block is the *same* as the **Filter** parameter setting of the corresponding `Dyadic Analysis Filter Bank` block. The use of wavelets requires a Wavelet Toolbox license. To learn how to design your own perfect reconstruction filters, see “References” on page 2-639.

Input Requirements

The inputs to this block are usually the outputs of a `Dyadic Analysis Filter Bank` block. Since the Dyadic Analysis Filter Bank block can output from either a single port or multiple ports, the Dyadic Synthesis Filter Bank block accepts inputs to either a single port or multiple ports.

The **Input** parameter sets whether the block accepts inputs from a single port or multiple ports, and thus determines the input requirements, as summarized in the following lists and figure.

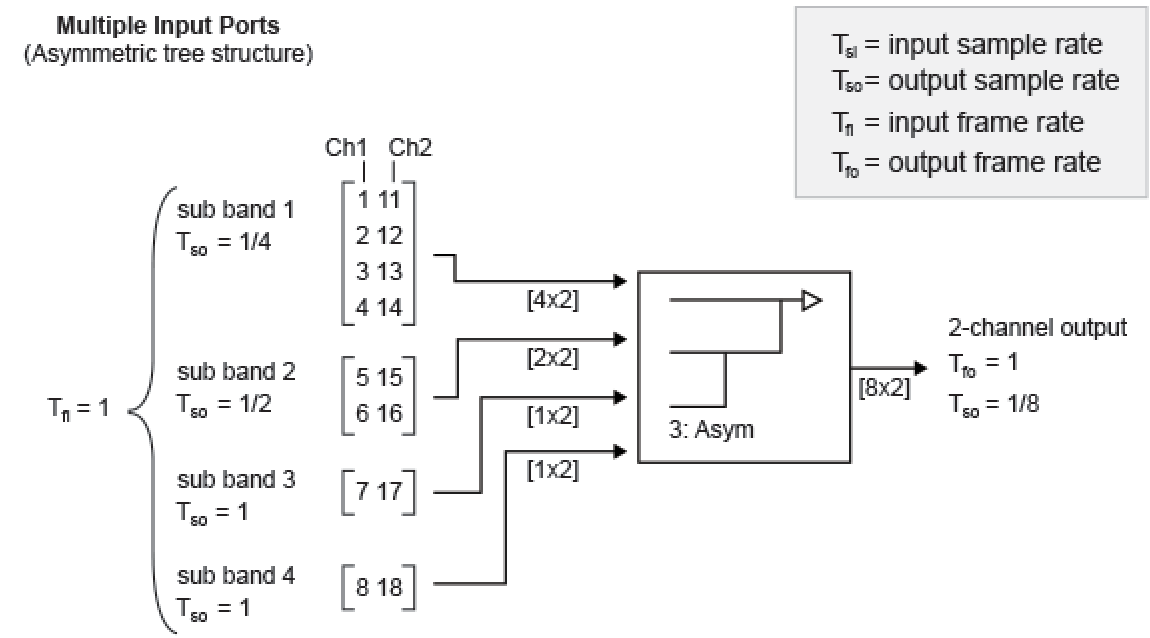
Note: Any output of a Dyadic Analysis Filter Bank block whose parameter settings match the corresponding settings of this block is a valid input to this block. For example, the setting of the Dyadic Analysis Filter Bank block parameter, **Output**, must be the same as this block's **Input** parameter (`Single port` or `Multiple ports`).

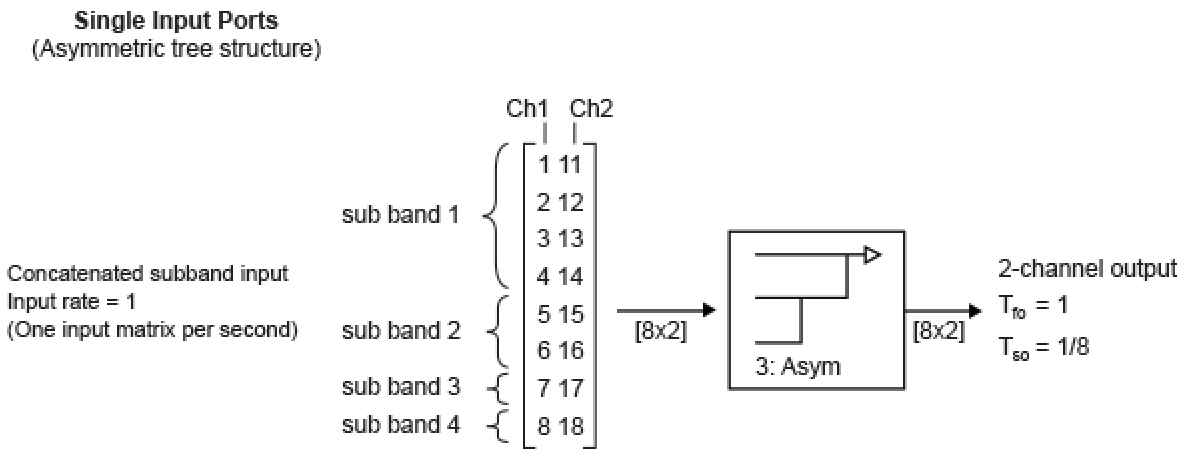
Valid Inputs for Input Set to Single Port

- Inputs must be vectors or matrices of concatenated subbands.
- Each input column contains the subbands for an independent signal.
- Upper input rows contain the high-frequency subbands, and the lower rows contain the low-frequency subbands.

Valid Inputs for Input Set to Multiple Ports

- Each subband must be provided as a vector or matrix to separate block input ports.
- The columns of each input contains a subband for an independent signal.
- The input to the topmost input port is the subband containing the highest frequencies, and the input to the bottommost port is the subband containing the lowest frequencies.


Multiple Input Ports
(Asymmetric tree structure)
T_{si} = input sample rate
T_{so}= output sample rate
T_{fi} = input frame rate
T_{fo} = output frame rate
Ch1 Ch2
sub band 1
T_{so} = 1/4
1 11
2 12
3 13
4 14
sub band 2
T_{so} = 1/2
5 15
6 16
sub band 3
T_{so} = 1
7 17
sub band 4
T_{so} = 1
8 18
T_{fi} = 1
[4x2]
[2x2]
[1x2]
[1x2]
3: Asym
[8x2]
2-channel output
T_{fo} = 1
T_{so} = 1/8


Single Input Ports
(Asymmetric tree structure)
Ch1 Ch2
Concatenated subband input
Input rate = 1
(One input matrix per second)
sub band 1
sub band 2
sub band 3
sub band 4
1 11
2 12
3 13
4 14
5 15
6 16
7 17
8 18
[8x2]
3: Asym
[8x2]
2-channel output
T_{fo} = 1
T_{so} = 1/8

For general information about the filter banks, see “Dyadic Synthesis Filter Banks”.

Output Characteristics

The following table summarizes the output characteristics for both types of inputs. For an illustration of why the output characteristics exist, see the figure Valid Inputs to a 3-Level Asymmetric Dyadic Synthesis Filter Bank.

	Input = Multiple ports	Input = Single port (Concatenated Subband Inputs)
Output Frame Rate	Same as the input frame rate.	Same as the input rate (the rate of the concatenated subband inputs).

Output Frame Dimensions	• The output has the same number of columns as the inputs.
	• The number of output rows depends on the tree structure of the filter bank:
	  • **Asymmetric** — The number of output rows is twice the number of rows in the input to the topmost input port.
	  • **Symmetric** — The number of output rows is the product of the number of input ports and the number of rows in an input to any input port.
		The output has the same number of rows and columns as the input.

For general information about the filter banks, see “Dyadic Synthesis Filter Banks”.

Wavelet Filter Banks

- “Filter Banks”

Filter Bank Filters

You must specify the highpass and lowpass filters in the filter bank by setting the **Filter** parameter to one of the following options:

- `User defined` — Allows you to explicitly specify the filters with two vectors of filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters. The block uses the same lowpass and highpass filters throughout the filter bank. The two filters should be halfband filters, where each filter passes the frequency band that the other filter stops. To use this block to perfectly reconstruct a signal decomposed by a `Dyadic Analysis Filter Bank` block, the filters in this block *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. To learn how to design your own perfect reconstruction filters, see "References" on page 2-639.
- Wavelet such as `Biorthogonal` or `Daubechies` — The block uses the specified wavelet to construct the lowpass and highpass filters using the Wavelet Toolbox function `wfilters`. Depending on the wavelet, the block might enable either the **Wavelet order** or **Filter order [synthesis / analysis]** parameter. (The latter parameter allows you to specify different wavelet orders for the analysis and synthesis filter stages.) To use this block to reconstruct a signal decomposed by a `Dyadic Analysis Filter Bank` block, you must set both blocks to use the same wavelets with the same order. You must have a Wavelet Toolbox license to use wavelets.

Specifying Filters with the Filter Parameter and Related Parameters

Filter	Sample Setting for Related Filter Specification Parameters	Corresponding Wavelet Function Syntax
User-defined	Filters based on Daubechies wavelets with wavelet order `3`: • **Lowpass FIR filter coefficients** = `[0.0352 -0.0854 -0.1350 0.4599 0.8069 0.3327]` • **Highpass FIR filter coefficients** = `[-0.3327 0.8069 -0.4599 -0.1350 0.0854 0.0352]`	None
Haar	None	`wfilters('haar')`
Daubechies	**Wavelet order** = `4`	`wfilters('db4')`
Symlets	**Wavelet order** = `3`	`wfilters('sym3')`
Coiflets	**Wavelet order** = `1`	`wfilters('coif1')`

Filter	Sample Setting for Related Filter Specification Parameters	Corresponding Wavelet Function Syntax
Biorthogonal	**Filter order [synthesis / analysis]** = `[3/1]`	`wfilters('bior3.1')`
Reverse Biorthogonal	**Filter order [synthesis / analysis]** = `[3/1]`	`wfilters('rbio3.1')`
Discrete Meyer	None	`wfilters('dmey')`

Examples

See "Examples" on page 2-626 on the `Dyadic Analysis Filter Bank` block reference page.

Parameters

The parameters displayed in the block dialog vary depending on the setting of the **Filter** parameter. Only some of the parameters described below are visible in the dialog box at any one time.

Note To use this block to reconstruct a signal decomposed by a `Dyadic Analysis Filter Bank` block, all the parameters in this block must be the same as the corresponding parameters in the Dyadic Analysis Filter Bank block (except the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients**; see the descriptions of these parameters).

Filter

The type of filter used to determine the high- and low-pass FIR filters in the filter bank:

- Select `User defined` to explicitly specify the filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.
- Select a wavelet such as `Biorthogonal` or `Daubechies` to specify a wavelet-based filter. The block uses the Wavelet Toolbox `wfilters` function to construct the filters. Extra parameters such as **Wavelet order** or **Filter order**

[synthesis / analysis] might become enabled. For a list of the supported wavelets, see the table Specifying Filters with the Filter Parameter and Related Parameters.

Lowpass FIR filter coefficients

A vector of filter coefficients (descending powers of *z*) that specifies coefficients used by all the lowpass filters in the filter bank. This parameter is enabled when you set **Filter** to `User defined`. The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. To perfectly reconstruct a signal decomposed by the `Dyadic Analysis Filter Bank`, the filters in this block *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction is not perfect. The default values of this parameter specify a perfect reconstruction filter for the default settings of the Dyadic Analysis Filter Bank (based on a Daubechies wavelet with wavelet order `3`).

Highpass FIR filter coefficients

A vector of filter coefficients (descending powers of *z*) that specifies coefficients used by all the highpass filters in the filter bank. This parameter is enabled when you set **Filter** to `User defined`. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. To perfectly reconstruct a signal decomposed by the `Dyadic Analysis Filter Bank`, the filters in this block *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction is not perfect. The default values of this parameter specify a perfect reconstruction filter for the default settings of the Dyadic Analysis Filter Bank (based on a Daubechies wavelet with wavelet order `3`).

Wavelet order

The order of the wavelet selected in the **Filter** parameter. This parameter is enabled only when you set **Filter** to certain types of wavelets, as shown in the table Specifying Filters with the Filter Parameter and Related Parameters.

Filter order [synthesis / analysis]

The order of the wavelet for the synthesis and analysis filter stages. For example, when you set the **Filter** parameter to **Biorthogonal** and set the **Filter order [synthesis / analysis]** parameter to `[2 / 6]`, the block calls the `wfilters` function with input argument `'bior2.6'`. This parameter is enabled only when you set **Filter** to certain types of wavelets, as shown in Specifying Filters with the Filter Parameter and Related Parameters.

Number of levels

The number of filter bank levels. An n-level asymmetric structure has n+1 inputs, and an n-level symmetric structure has 2^n inputs, as shown in "Wavelet Filter Banks" on page 2-635.

The default setting of this parameter is `2`.

Tree structure

The structure of the filter bank: `Asymmetric`, or `Symmetric`. See "Wavelet Filter Banks" on page 2-635.

The default setting of this parameter is `Asymmetric` for the Dyadic Synthesis Filter Bank block, and `Symmetric` for the IDWT block.

Input

Set to `Multiple ports` to accept each input subband at a separate port (the topmost port accepts the subband with the highest frequency band). Set to `Single port` to accept one vector or matrix of concatenated subbands at a single port. For more information, see "Input Requirements" on page 2-632.

The default setting of this parameter is `Multiple ports` for the Dyadic Synthesis Filter Bank block, and `Single port` for the IDWT block.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
Output	• Double-precision floating point • Single-precision floating point

See Also

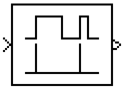
`Dyadic Analysis Filter Bank`	DSP System Toolbox
`IDWT`	DSP System Toolbox
`Two-Channel Synthesis Subband Filter`	DSP System Toolbox

See “Multirate and Multistage Filters” for related information.

Introduced before R2006a

Edge Detector

Detect transition from zero to nonzero value



Library

Signal Management / Switches and Counters

`dspswit3`

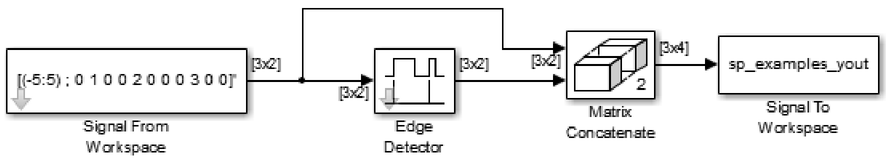
Description

The Edge Detector block generates an impulse (the value 1) in a given output channel when the corresponding channel of the input transitions from zero to a nonzero value. When the input does not transition from zero to a nonzero value, the block generates a zero in the corresponding output channel.

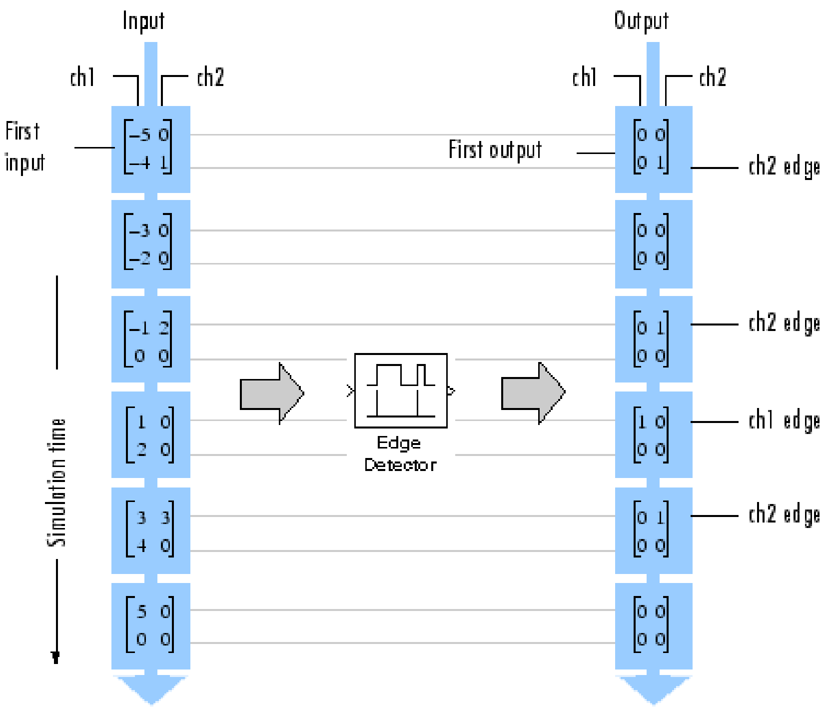
The output has the same dimension and sample rate as the input. When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block counts an edge that is split across two consecutive frames in the frame that contains the nonzero value. For example, if there is a zero at the bottom of the first frame and a nonzero value at the top of the second frame, the block counts the edge in the second frame.

Examples

In the ex_edgedetector_ref model, the **Input processing** parameter of the Edge Detector block is set to `Columns as channels (frame based)`. Thus, the block interprets the 3-by-2 input as a multichannel signal with a frame size of 3. The Matrix Concatenate block concatenates the two input channels of the original signal with the two output channels of the Edge Detector block to create the four-channel workspace variable `sp_examples_yout`.



As shown in the following figure, the block finds edges at sample 7 in channel 1, and at samples 2, 5, and 9 in channel 2.



Parameters

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- Boolean — The block might output Boolean values depending on the input data type, and whether Boolean support is enabled or disabled.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers
- Enumerated

See Also

Counter	DSP System Toolbox
Event-Count Comparator	DSP System Toolbox

Introduced before R2006a

Event-Count Comparator

Detect threshold crossing of accumulated nonzero inputs

Library

Signal Management / Switches and Counters

`dspswit3`

Description

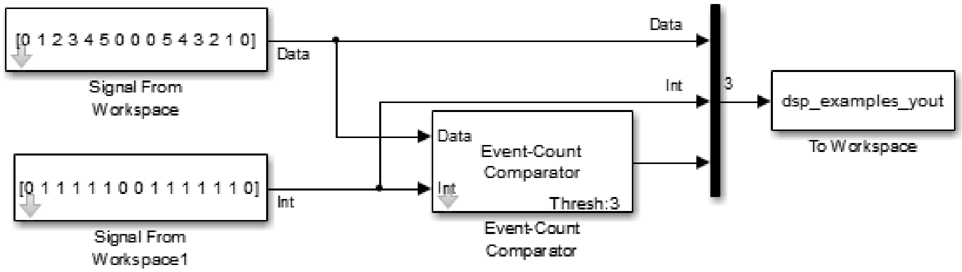
The Event-Count Comparator block records the number of nonzero inputs to the Data port during the period that the block is enabled by a high signal (the value 1) at the Int port. Both inputs must be scalars.

When the number of accumulated nonzero inputs first equals the **Event threshold** setting, the block waits one additional sample interval, and then sets the output high (1). The block holds the output high until recording is restarted by a low-to-high (0-to-1) transition at the Int port.

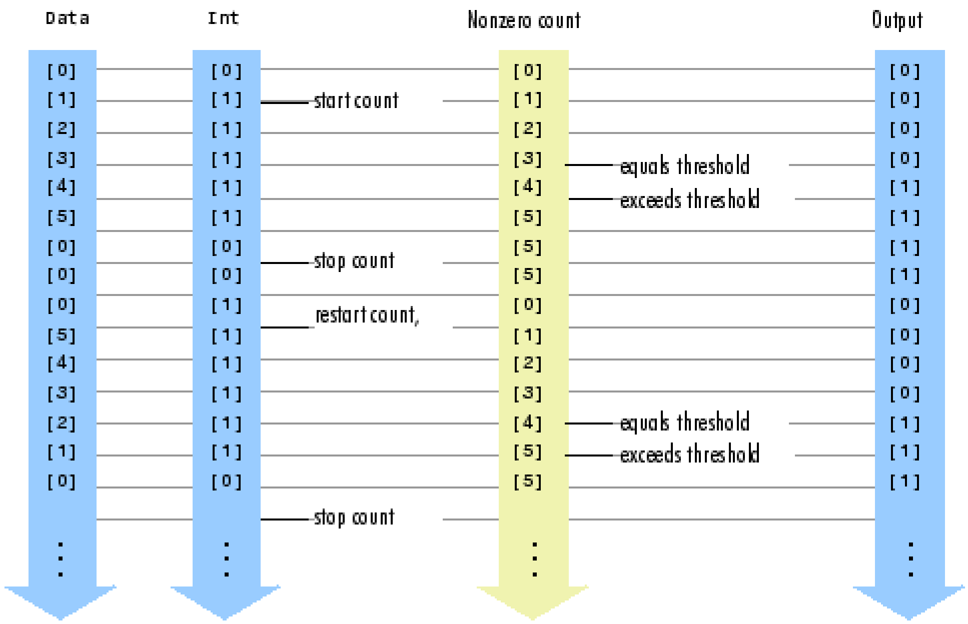
The Event-Count Comparator block accepts real and complex floating-point and fixed-point inputs. However, because the block has discrete state, it does not support constant or continuous sample times. Therefore, at least one input or output port of the Event-Count Comparator block must be connected to a block whose **Sample time** parameter is discrete. The Event-Count Comparator block inherits this non-infinite discrete sample time.

Examples

In the ex_eventcountcomp_ref model, the Event-Count Comparator block (**Event threshold** = 3) detects two threshold crossings in the input to the Data port, one at sample 4 and one at sample 12.



All inputs and outputs are multiplexed into the workspace variable `yout`, whose contents are shown in the figure below. The two left columns in the illustration show the inputs to the Data and Int ports, the center column shows the state of the block's internal counter, and the right column shows the block's output.



Parameters

Event threshold

Specify the value against which to compare the number of nonzero inputs. Tunable (Simulink).

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers
- Enumerated

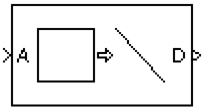
See Also

Counter	DSP System Toolbox
Edge Detector	DSP System Toolbox

Introduced before R2006a

Extract Diagonal

Extract main diagonal of input matrix



Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

Description

The Extract Diagonal block populates the unoriented output vector with the elements on the main diagonal of the M-by-N input matrix `A`.

```
D = diag(A)     Equivalent MATLAB code
```

The output vector has length `min(M,N)`.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- Boolean — Block outputs are always Boolean.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

See Also

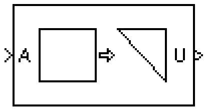
Create Diagonal Matrix	DSP System Toolbox

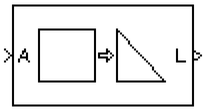
Extract Triangular Matrix	DSP System Toolbox
`diag`	MATLAB

Introduced before R2006a

Extract Triangular Matrix

Extract lower or upper triangle from input matrices





Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

Description

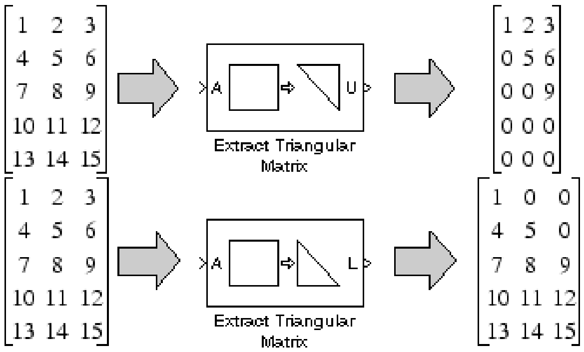
The Extract Triangular Matrix block creates a triangular matrix output from the upper or lower triangular elements of an M-by-N input matrix. The block treats length-M unoriented vector inputs as an M-by-1 matrix.

The **Extract** parameter selects between the two components of the input:

- `Upper` — Copies the elements on and above the main diagonal of the input matrix to an output matrix of the same size. The first *row* of the output matrix is therefore identical to the first *row* of the input matrix. The elements below the main diagonal of the output matrix are zero.
- `Lower` — Copies the elements on and below the main diagonal of the input matrix to an output matrix of the same size. The first *column* of the output matrix is therefore identical to the first *column* of the input matrix. The elements above the main diagonal of the output matrix are zero.

Examples

The ex_extracttriang_ref model below shows the extraction of upper and lower triangles from a 5-by-3 input matrix.



Parameters

Extract

The component of the matrix to copy to the output: upper triangle or lower triangle.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
U	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned)

Port	Supported Data Types
	• Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
L	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

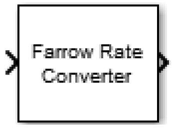
See Also

Autocorrelation LPC	DSP System Toolbox
Cholesky Factorization	DSP System Toolbox
Extract Diagonal	DSP System Toolbox
Forward Substitution	DSP System Toolbox
LDL Factorization	DSP System Toolbox
LU Factorization	DSP System Toolbox
`tril`	MATLAB
`triu`	MATLAB

Introduced before R2006a

Farrow Rate Converter

Polynomial sample-rate converter with arbitrary conversion factor



Library

Signal Operations

`dspsigops`

Description

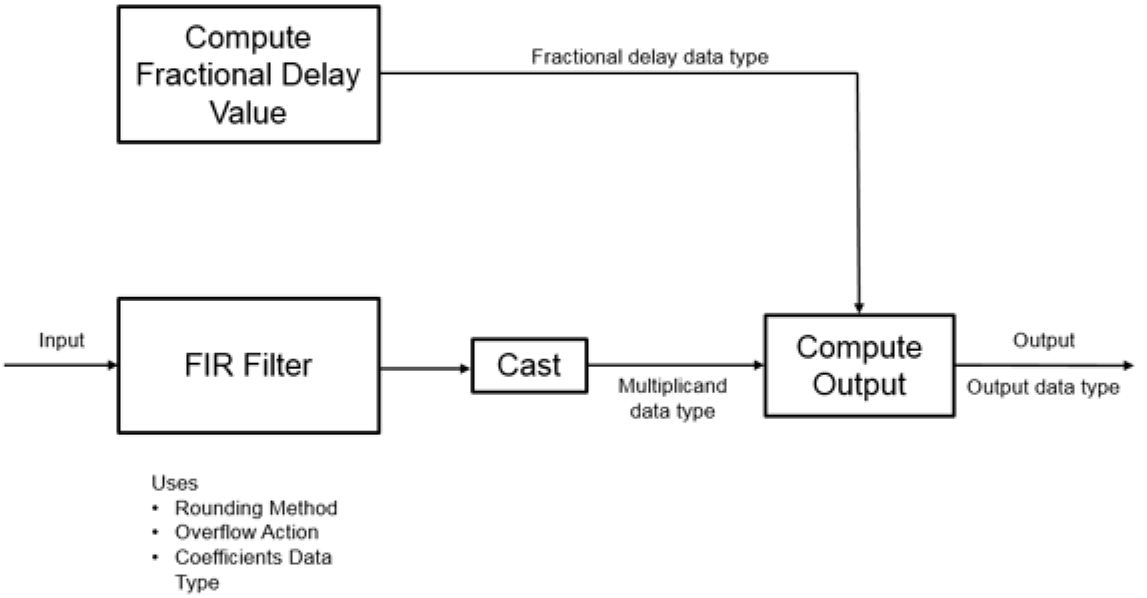
The Farrow Rate Converter block converts the sample rate of an input signal using polynomial fit sample-rate conversion. Polynomial-based filters are efficient at implementing fractional sample rate conversion. Farrow structures are implementations of polynomial-based filters. This block uses a Farrow structure to implement arbitrary rate-change factors efficiently. The rate-change factors can be irrational.

The input frame size must be a multiple of the decimation factor of the rate converter. The decimation factor depends on the parameter settings of the block. To determine the decimation factor, in the block dialog box click **View Info** .

Each column of the input signal is treated as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then it is interpreted as a single channel. The inputs to the block can be single, double, or fixed-point data type.

Fixed-Point Data Types

The signal flow diagram shows the data types used within the farrow rate converter for fixed-point signals and floating-point signals. You can specify these data types in the Farrow rate converter. If the input is floating point, all the data types with in the signal flow diagram have the same data type as the input, which can be single or double.



If the input is fixed point, the FIR filter uses the rounding mode, overflow mode, and the coefficients data type data you provide to the Farrow rate converter. The accumulator data type and the product data type in the signal flow are full precision.

Dialog Box

Main tab:

Sample rate of input signal (Hz)

Sample rate of the input signal, specified as a positive scalar in Hz. The input sample rate must be greater than the bandwidth of interest. The default is `48e3`.

Sample rate of output signal (Hz)

Sample rate of the output signal, specified as a positive scalar in Hz. The output sample rate must be higher or lower than the input sample rate. The default is `98e3`.

Tolerance for output sample rate

Maximum allowed tolerance for the output sample rate, specified as a positive scalar in the range [`0` to`0.5`]. The default is `0`.

The actual output sample rate varies but is within the specified range. For example, suppose that you set the **Tolerance for output sample rate** to 0.01. Then the actual output sample rate is in the range given by sample rate of output signal ± 1%. This flexibility allows for a simpler filter design.

Specification method

Method used to specify the polynomial interpolator coefficients, specified as one of the following:

- `Polynomial order` — Specify the order of the Lagrange interpolation filter polynomial through the **Polynomial order** parameter.
- `Coefficients` — Specify the polynomial coefficients directly through the **Coefficients** parameter.

Polynomial order

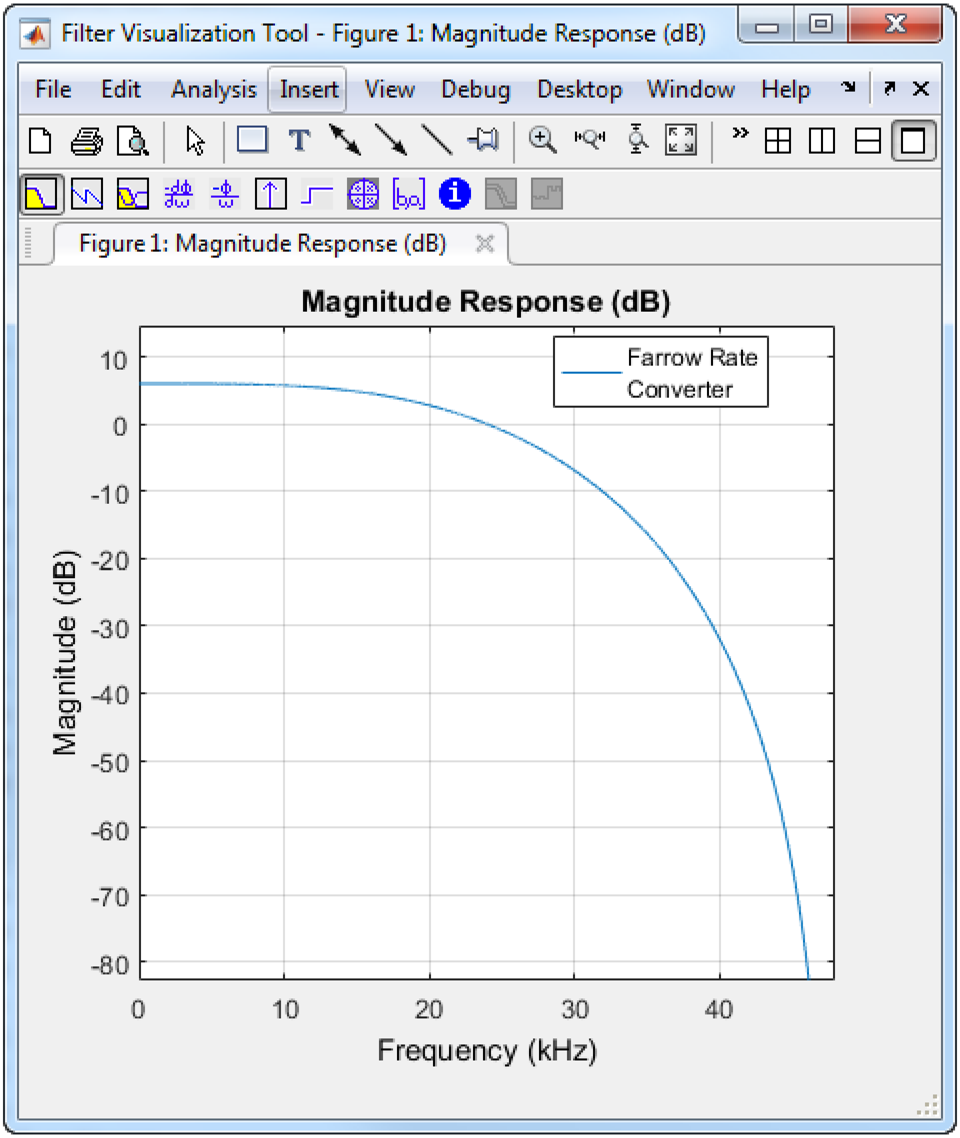
Order of filter polynomial, specified as a `1`, `2`, `3`, or `4`. The default is `3`. This parameter applies only when you set **Specification method** to `Polynomial order`.

Coefficients

Filter polynomial coefficients, specified as a real-valued square matrix. The default is `[-1 1; 1 0]`. This property applies only when you set **Specification method** to `Coefficients`.

View Filter Response

Opens the fvtool and displays the magnitude/phase response of the Farrow Rate Converter. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.

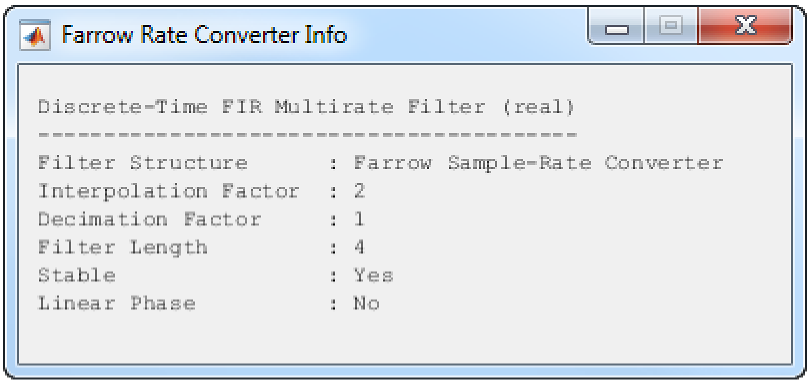


To update the magnitude response while fvtool is running, modify the dialog box parameters and click **Apply**.

View Info

Display the following information about the Farrow filter system:

- `Filter Structure`
- `Interpolation Factor`
- `Decimation Factor`
- `Filter Length`
- `Stable`
- `Linear Phase`



This button brings the functionality of the `dsp.FarrowRateConverter.info` method into the Simulink environment.

Simulate using

Type of simulation to run. You can set this parameter to:

- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Data Types tab:

Rounding mode

Rounding mode for fixed-point operations, specified as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Zero`. The default is `Floor`. For more information on the rounding modes, see “Precision and Range”.

This property is not tunable.

Overflow mode

Overflow action for fixed-point operations, specified as `Wrap` | `Saturate` . The default is `Wrap`. For more details on the overflow action to select, see the 'Overflow Handling' section of “Precision and Range”.

This property is not tunable.

Coefficients

Data type of the filter coefficients, specified as a signed fixed-point object. The default, `fixdt(1,16)`, corresponds to a signed fixed-point type object with 16-bit coefficients. To give the best possible precision, fraction length is determined based on the coefficient values.

This property is not tunable.

Fractional Delay

Data type of the fractional delay, specified as an unsigned fixed-point object. The default, `fixdt(0,8)`, corresponds to an unsigned fixed-point data type object with 8-bit word length. To give the best possible precision, fractional length computed based on the fractional delay values.

This property is not tunable.

Multiplicand

Data type of the multiplicand, specified as a signed fixed-point object. The default, `fixdt(1,16,13)`, corresponds to a signed fixed-point multiplicand data type with 16-bit word length and 13-bit fraction length.

This property is not tunable.

Output

Word length and fraction length of the output data type, specified as one of the following:

- `Inherit: Same word length as input` (default) — Output word length and fraction lengths are the same as the input.
- `Inherit: Same as accumulator` — Output word length and fraction lengths are the same as the accumulator.
- `fixdt(1,16)` — Signed fixed-point data type with 16-bit word length. To give the best possible precision, fraction length is computed based on the input range. The dynamic range of the input is preserved.
- `fixdt(1,16,0)` — Signed fixed-point data type with 16-bit word length and zero fraction length.

This property is not tunable.

Output Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink uses this value to perform automatic scaling of fixed-point data types.

Output Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink uses this value to perform automatic scaling of fixed-point data types.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Signed fixed point • 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Signed fixed point • 8-, 16-, and 32-bit signed integers

See Also

dsp.FarrowRateConverter	DSP System Toolbox
`Sample-Rate Converter`	DSP System Toolbox

Algorithm

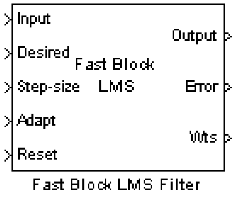
This block brings the capabilities of the dsp.FarrowRateConverter System object to the Simulink environment.

For information on the algorithms used by this block, see the Algorithms on page 4-649 section of `dsp.FarrowRateConverter`.

Introduced in R2015b

Fast Block LMS Filter

Compute output, error, and weights using LMS adaptive algorithm



Library

Filtering / Adaptive Filters

`dspadpt3`

Description

The Fast Block LMS Filter block implements an adaptive least mean-square (LMS) filter, where the adaptation of the filter weights occurs once for every block of data samples. The block estimates the filter weights, or coefficients, needed to convert the input signal into the desired signal. Connect the signal you want to filter to the Input port. The input signal can be a scalar or a column vector. Connect the signal you want to model to the Desired port. The desired signal must have the same data type, complexity, and dimensions as the input signal. The Output port outputs the filtered input signal. The Error port outputs the result of subtracting the output signal from the desired signal.

The block calculates the filter weights using the Block LMS Filter equations. For more information, see Block LMS Filter. The Fast Block LMS Filter block implements the convolution operation involved in the calculations of the filtered output, y, and the weight update function in the frequency domain using the FFT algorithm used in the Overlap-Save FFT Filter block. See Overlap-Save FFT Filter for more information.

Use the **Filter length** parameter to specify the length of the filter weights vector.

The **Block size** parameter determines how many samples of the input signal are acquired before the filter weights are updated. The input frame length must be a multiple of the **Block size** parameter.

The **Step-size (mu)** parameter corresponds to μ in the equations. You can either specify a step-size using the input port, Step-size, or enter a value in the Block Parameters: Block LMS Filter dialog box.

Use the **Leakage factor (0 to 1)** parameter to specify the leakage factor, $0 < 1 - \mu\alpha \le 1$, in the leaky LMS algorithm shown below.

$$\mathbf{w}(k) = (1 - \mu\alpha)\mathbf{w}(k-1) - f(\mathbf{u}(n), e(n), \mu)$$

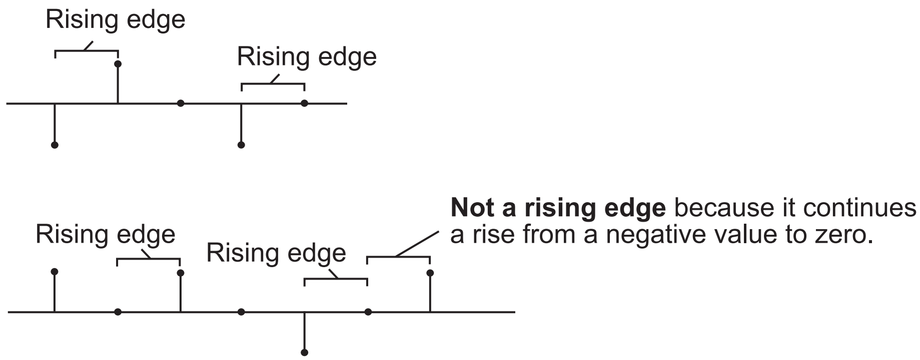
Enter the initial filter weights, $\mathbf{w}(0)$, as a vector or a scalar in the **Initial value of filter weights** text box. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value.

When you select the **Adapt port** check box, an Adapt port appears 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 at their current values.

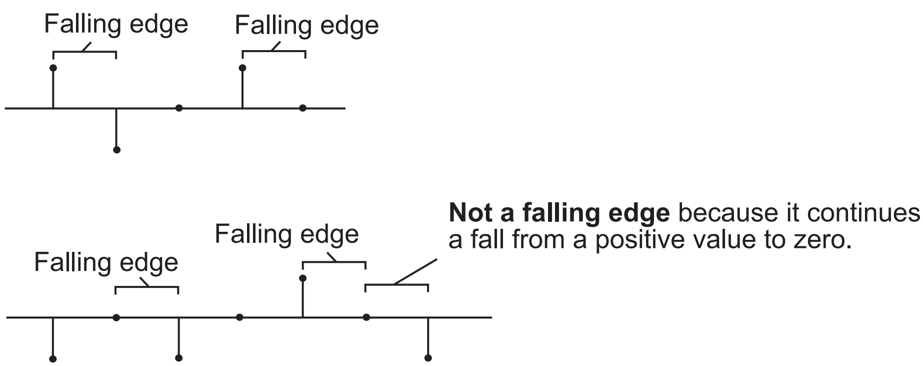
When you want to reset the value of the filter weights to their initial values, use the **Reset input** parameter. The block resets the filter weights whenever a reset event is detected at the Reset port. The reset signal rate must be the same rate as the data signal input.

From the **Reset input** list, select `None` to disable the Reset port. To enable the Reset port, select one of the following from the **Reset input** list:

- `Rising edge` — Triggers a reset operation when the Reset input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- `Falling edge` — Triggers a reset operation when the Reset input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- `Either edge` — Triggers a reset operation when the Reset input is a `Rising edge` or `Falling edge` (as described above)
- `Non-zero sample` — Triggers a reset operation at each sample time that the Reset input is not zero

Select the **Output filter weights** check box to create a Wts port on the block. For each iteration, the block outputs the current updated filter weights from this port.

Parameters

Filter length

Enter the length of the FIR filter weights vector. The sum of the **Block size** and the **Filter length** must be a power of 2.

Block size

Enter the number of samples to acquire before the filter weights are updated. The number of rows in the input must be an integer multiple of the **Block size**. The sum of the **Block size** and the **Filter length** must be a power of 2.

Specify step-size via

Select `Dialog` to enter a value for mu, or select `Input port` to specify mu using the Step-size input port.

Step-size (mu)

Enter the step-size. Tunable (Simulink).

Leakage factor (0 to 1)

Enter the leakage factor, $0 < 1 - \mu\alpha \leq 1$. Tunable (Simulink).

Initial value of filter weights

Specify the initial values of the FIR filter weights.

Adapt port

Select this check box to enable the Adapt input port.

Reset input

Select this check box to enable the Reset input port.

Output filter weights

Select this check box to export the filter weights from the Wts port.

References

Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Desired	• Must be the same as Input
Step-size	• Must be the same as Input
Adapt	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Reset	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Same as Input
Error	• Same as Input
Wts	• Same as Input

See Also

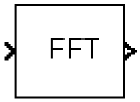
Block LMS Filter	DSP System Toolbox
Kalman Adaptive Filter (Obsolete)	DSP System Toolbox
LMS Filter	DSP System Toolbox
RLS Filter	DSP System Toolbox

See “Adaptive Filters in Simulink” for related information.

Introduced before R2006a

FFT

Fast Fourier transform (FFT) of input



Library

Transforms

`dspxfrm3`

Description

The FFT block computes the fast Fourier transform (FFT) across the first dimension of an *N*-D input array, *u*. For user-specified FFT lengths, not equal to *P*, zero padding or truncating, or modulo-length data wrapping occurs before the FFT operation.

```
y = fft(u,M)  % P ≤ M
```

Wrapping:

```
y(:,l) = fft(datawrap(u(:,l),M))  % P > M; l = 1,...,N
```

Truncating:

```
y (:,l) = fft(u,M)     % P > M; l = 1,...,N
```

When the input length, *P*, is greater than the FFT length, *M*, you may see magnitude increases in your FFT output. These magnitude increases occur because the FFT block uses modulo-*M* data wrapping to preserve all available input samples.

To avoid such magnitude increases, you can truncate the length of your input sample, *P*, to the FFT length, *M*. To do so, place a `Pad` block before the FFT block in your model.

The *k*th entry of the *l*th output channel, $y(k, l)$, equals the *k*th point of the *M*-point discrete Fourier transform (DFT) of the *l*th input channel:

$$y(k,l) = \sum_{p=1}^{P} u(p,l) e^{-j2\pi(p-1)(k-1)/M} \qquad k = 1, \ldots, M$$

The block uses one of two possible FFT implementations. You can select an implementation based on the FFTW library or an implementation based on a collection of Radix-2 algorithms. You can select `Auto` to allow the block to choose the implementation.

FFTW Implementation

The FFTW implementation provides an optimized FFT calculation including support for power-of-two and non-power-of-two transform lengths in both simulation and code generation. Generated code using the FFTW implementation will be restricted to those computers which are capable of running MATLAB. The input data type must be floating-point.

Radix-2 Implementation

The Radix-2 implementation supports bit-reversed processing, fixed or floating-point data, and allows the block to provide portable C-code generation using the Simulink Coder. The dimension M of the M-by-N input matrix, must be a power of two. To work with other input sizes, use the `Pad` block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation.

With Radix-2 selected, the block implements one or more of the following algorithms:

- Butterfly operation
- Double-signal algorithm
- Half-length algorithm
- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm

Radix-2 Algorithms for Real and Complex Signals

Complexity of Input	Output Ordering	Algorithms Used for FFT Computation
Complex	Linear	Bit-reversed operation and radix-2 DIT
Complex	Bit-reversed	Radix-2 DIF
Real	Linear	Bit-reversed operation and radix-2 DIT in conjunction with the half-length and double-signal algorithms
Real	Bit-reversed	Radix-2 DIF in conjunction with the half-length and double-signal algorithms

The efficiency of the FFT algorithm can be enhanced for real input signals by forming complex-valued sequences from the real-valued sequences prior to the computation of the DFT. When there are $2N+1$ real input channels, the FFT block forms these complex-valued sequences by applying the double-signal algorithm to the first $2N$ input channels, and the half-length algorithm to the last odd-numbered channel.

For real input signals with fixed-point data types, it is possible to see different numerical results in the output of the last odd numbered channel, even when all input channels are identical. This numerical difference results from differences in the double-signal algorithm and the half-length algorithm.

You can eliminate this numerical difference in two ways:

- Using full precision arithmetic for fixed-point input signals
- Changing the input data type to floating point

For more information on the double-signal and half-length algorithms, see [2]. "Efficient Computation of the DFT of Two Real Sequences" on page 475 describes the double signal algorithm. "Efficient Computation of the DFT of a 2N-Point Real Sequence" on page 476 describes the half-length algorithm.

Radix-2 Optimization for the Table of Trigonometric Values

In certain situations, the block's Radix–2 algorithm computes all the possible trigonometric values of the twiddle factor

$$e^{j\frac{2\pi k}{K}}$$

where K is the greater value of either M or N and $k = 0, \cdots, K-1$. The block stores these values in a table and retrieves them during simulation. The number of table entries for fixed-point and floating-point is summarized in the following table:

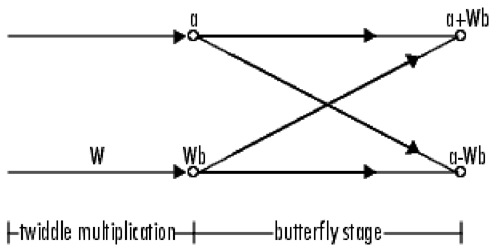
Number of Table Entries for N-Point FFT	
floating-point	3 N/4
fixed-point	N

Fixed-Point Data Types

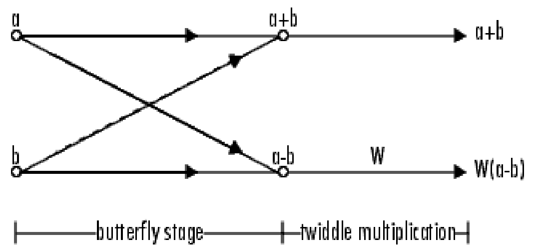
The following diagrams show the data types used in the FFT block for fixed-point signals. You can set the Sine table, Accumulator, Product output, and Output data types displayed in the diagrams in the FFT dialog box as discussed in "Dialog Box" on page 2-671.

Inputs to the FFT block are first cast to the output data type and stored in the output buffer. Each butterfly stage then processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type. The block multiplies in a twiddle factor before each butterfly stage in a decimation-in-time FFT and after each butterfly stage in a decimation-in-frequency FFT.

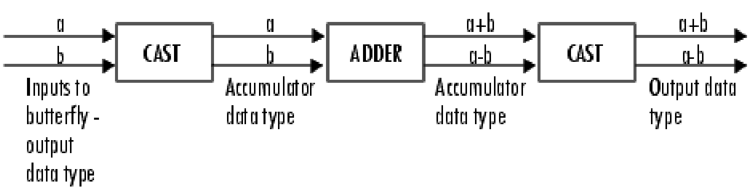
Decimation-in-time IFFT



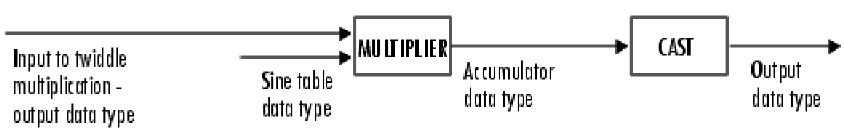
Decimation-in-frequency IFFT



Butterfly stage data types



Twiddle multiplication data types

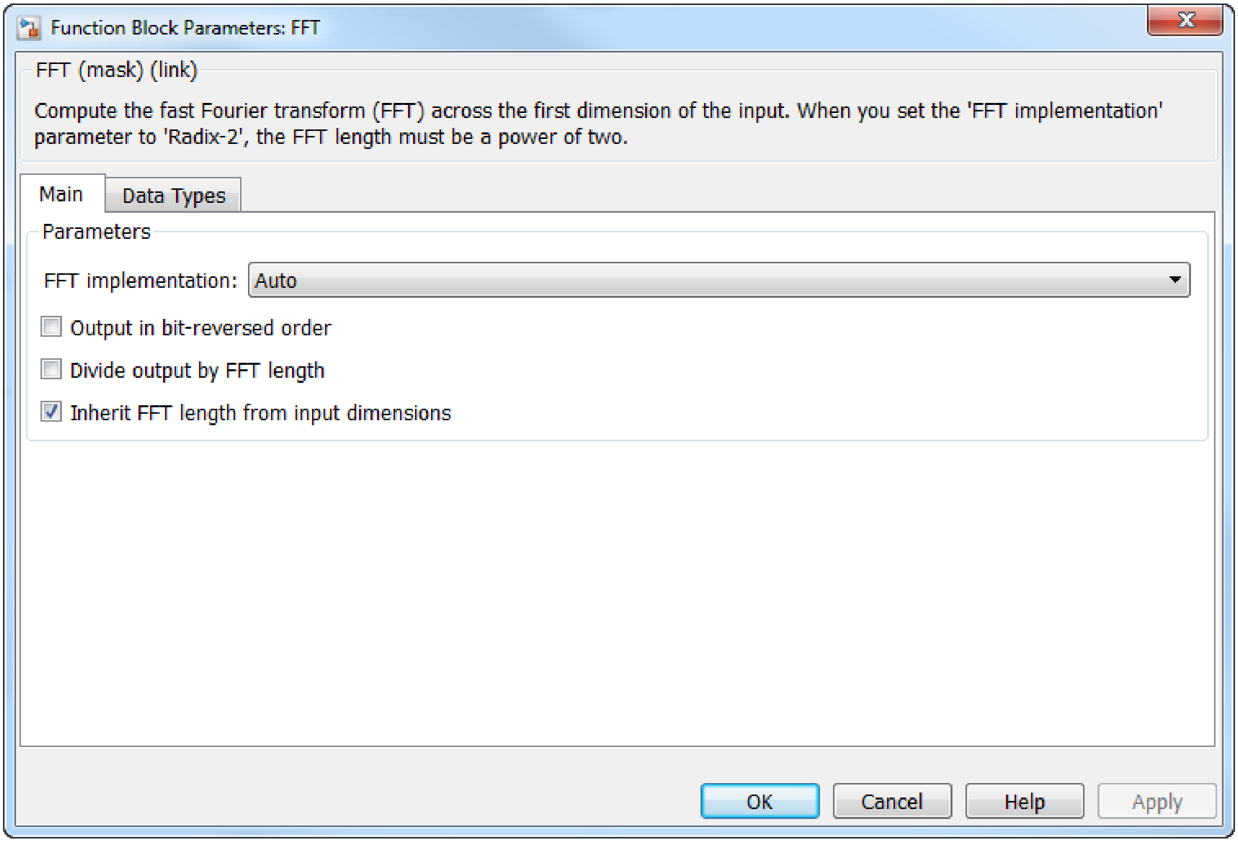


The output of the multiplier appears in the accumulator data type because both of the inputs to the multiplier are complex. For details on the complex multiplication performed, see “Multiplication Data Types”.

Note: When the block input is fixed point, all internal data types are signed fixed point.

Dialog Box

The **Main** pane of the FFT block dialog appears as follows.



FFT implementation

Set this parameter to `FFTW` to support an arbitrary length input signal. The block restricts generated code with FFTW implementation to host computers capable of running MATLAB.

Set this parameter to `Radix-2` for bit-reversed processing, fixed or floating-point data, or for portable C-code generation using the Simulink Coder. The dimension M of the M-by-N input matrix, must be a power of two. To work with other input sizes, use the `Pad` block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation. See "Radix-2 Implementation" on page 2-667.

Set this parameter to `Auto` to let the block choose the FFT implementation. For non-power-of-two transform lengths, the block restricts generated code to MATLAB host computers.

Output in bit-reversed order

Designate the order of the output channel elements relative to the ordering of the input elements. When you select this check box, the output channel elements appear in bit-reversed order relative to the input ordering. If you clear this check box, the output channel elements appear in linear order relative to the input ordering.

Linearly ordering the output requires extra data sorting manipulation, so in some situations it might be better to output in bit-reversed order.

Note: The FFT block calculates its output in bit-reversed order. Linearly ordering the FFT block output requires an extra bit-reversal operation. Thus, in many situations, you can increase the speed of the FFT block by selecting the **Output in bit-reversed order** check box.

For more information ordering of the output, see "Linear and Bit-Reversed Output Order".

Divide output by FFT length

When you select this parameter, the block divides the output of the FFT by the FFT length. This option is useful when you want the output of the FFT to stay in the same amplitude range as its input. This is particularly useful when working with fixed-point data types.

Inherit FFT length from input dimensions

Select to inherit the FFT length from the input dimensions. When you select this check box, the input length must be a power of two. When you do not select this check box, the **FFT length** parameter becomes available to specify the length.

FFT length

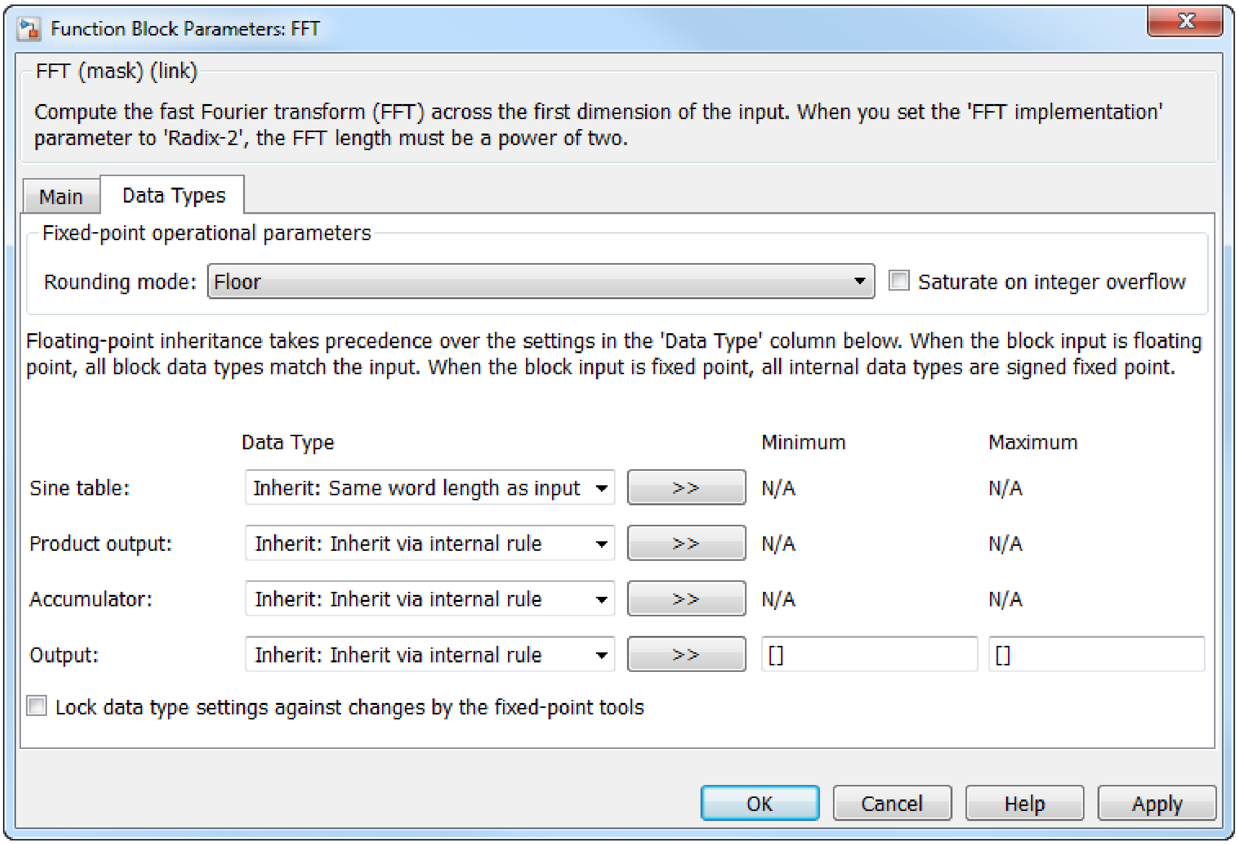
Specify FFT length. This parameter becomes available only when you do not select the **Inherit FFT length from input dimensions** parameter.

When you set the **FFT implementation** parameter to `Radix-2`, or when you check the **Output in bit-reversed order** check box, this value must be a power of two.

Wrap input data when FFT length is shorter than input length

Choose to wrap or truncate the input, depending on the FFT length. If this parameter is checked, modulo-length data wrapping occurs before the FFT operation, given FFT length is shorter than the input length. If this property is unchecked, truncation of the input data to the FFT length occurs before the FFT operation. The default is checked.

The **Data Types** pane of the FFT block dialog appears as follows.



Rounding mode

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameter; instead, they always round to `Nearest`.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations. The sine table values do not obey this parameter; instead, they are always saturated.

Sine table data type

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

The sine table values do not obey the **Rounding mode** and **Saturate on integer overflow** parameters; instead, they are always saturated and rounded to `Nearest`.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Sine table data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Product output data type

Specify the product output data type. See and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`.

  When you select `Inherit: Inherit via internal rule`, the block calculates the output word length and fraction length automatically. The equations that the block uses to calculate the ideal output word length and fraction length depend on the setting of the **Divide output by FFT length** check box.

  - When you select the **Divide output by FFT length** check box, the ideal output word and fraction lengths are the same as the input word and fraction lengths.
  - When you clear the **Divide output by FFT length** check box, the block computes the ideal output word and fraction lengths according to the following equations:

$$WL_{ideal\ output} = WL_{input} + floor(\log_2(FFT\ length - 1)) + 1$$

$$FL_{ideal\ output} = FL_{input}$$

Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information, see "Inherit via Internal Rule".

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Control Signal Data Types" (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges" (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges" (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Examples

See the section on "Transform Time-Domain Data into Frequency Domain" in the *DSP System Toolbox User's Guide.*

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

References

[1] Orfanidis, S. J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1996, p. 497.

[2] Proakis, John G. and Dimitris G. Manolakis. *Digital Signal Processing*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 1996.

[3] FFTW (http://www.fftw.org)

[4] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT,"*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- When the following conditions apply, the executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB:
  - **FFT implementation** is set to `FFTW`.
  - **Inherit FFT length from input dimensions** is cleared, and **FFT length** is set to a value that is not a power of two.

  Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- When the FFT length is a power of two, you can generate standalone C and C++ code from this block.

See Also

See Also

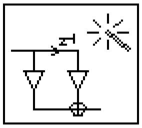
Functions
`bitrevorder` | `fft` | `ifft`

Blocks
`DCT` | `IFFT` | `Pad`

Introduced before R2006a

Filter Realization Wizard

Construct filter realizations using digital filter blocks or Sum, Gain, and Delay blocks



Library

Filtering / Filter Implementations

`dsparch4`

Description

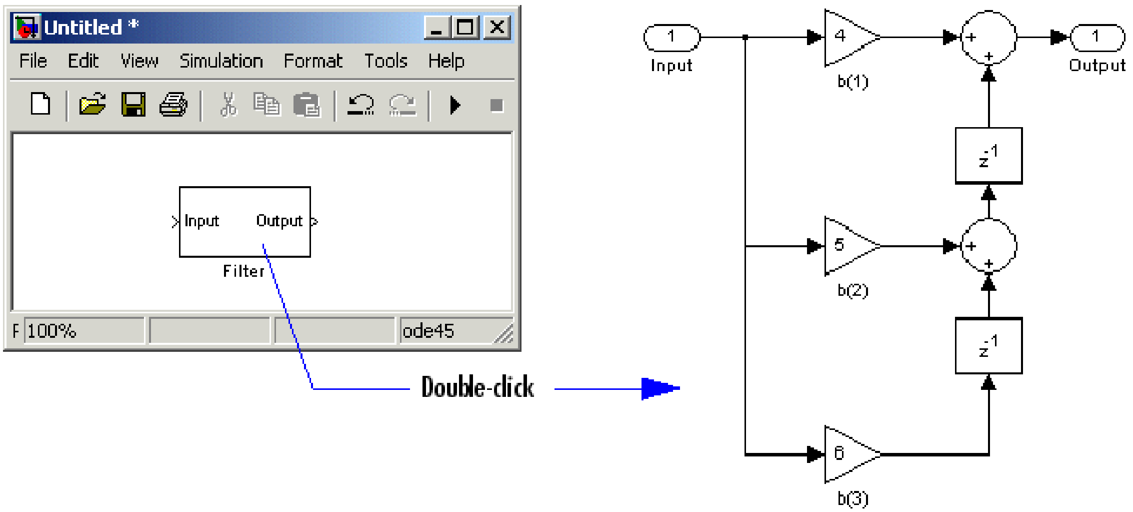
Note: Use this block to implement fixed-point or floating-point digital filters using Sum, Gain, and Delay blocks or digital filter blocks from the DSP System Toolbox library. You can either design a filter by using the block parameters, or import the coefficients of a filter you have designed elsewhere.

The following blocks also implement digital filters, but serve slightly different purposes:

- `Discrete FIR Filter` and `Biquad Filter`— Use to implement floating-point or fixed-point filters that you have already designed
- `Digital Filter Design` — Use to design, analyze, and then implement floating-point filters.

The Filter Realization Wizard is a tool for automatically implementing a digital filter. You must specify a filter, its structure, and the data types for the inputs, outputs, and computations. The filter can support double-precision, single-precision, or fixed-point data types.

The Filter Realization Wizard can implement a digital filter in one of two ways. It can use digital filter blocks from the DSP System Toolbox library, or it can create a subsystem block that implements the specified filter using Sum, Gain, and `Delay` blocks. If the Filter Realization Wizard creates a block, double-click the block to open the dialog box. If it creates a subsystem, double-click the subsystem block to see the filter implementation as shown in the figure below.



For more information about filter implementation, see “Specify the Filter Implementation” on page 2-682.

The parameters of the Filter Realization Wizard are a part of a larger app, the filter designer (`filterDesigner`). You can use filter designer to design and analyze your filter, and then use the Filter Realization Wizard parameters to implement the filter in your models.

Specify the Filter and Data Types

To specify a purely double-precision filter, you can either design a filter using the **Design Filter** panel, or import a filter using the **Import Filter** panel. In the **Import Filter** panel, you can specify the coefficients directly or specify the workspace variables which store the coefficients.

You can also specify a fixed-point filter or a single-precision filter by using the **Set Quantization Parameters** panel.

Note: *Running* a model containing implementations of fixed-point filters requires the Fixed-Point Designer product, but you can still edit models containing such filter implementations without it. See the Fixed-Point Designer documentation for more information.

See the following topics to learn how to use the panels to specify your filter:

- For more information on the **Design Filter** panel, see "Filter Designer" (Signal Processing Toolbox).
- For more information on the **Import Filter** panel, see "Importing a Filter Design".
- For more information on the **Set Quantization Parameters** panel, see "Access the Quantization Features of Filter Designer".

To open a panel, click the appropriate button in the lower-left corner of filter designer.

Supported Filter Structures

The Filter Realization Wizard supports the following structures:

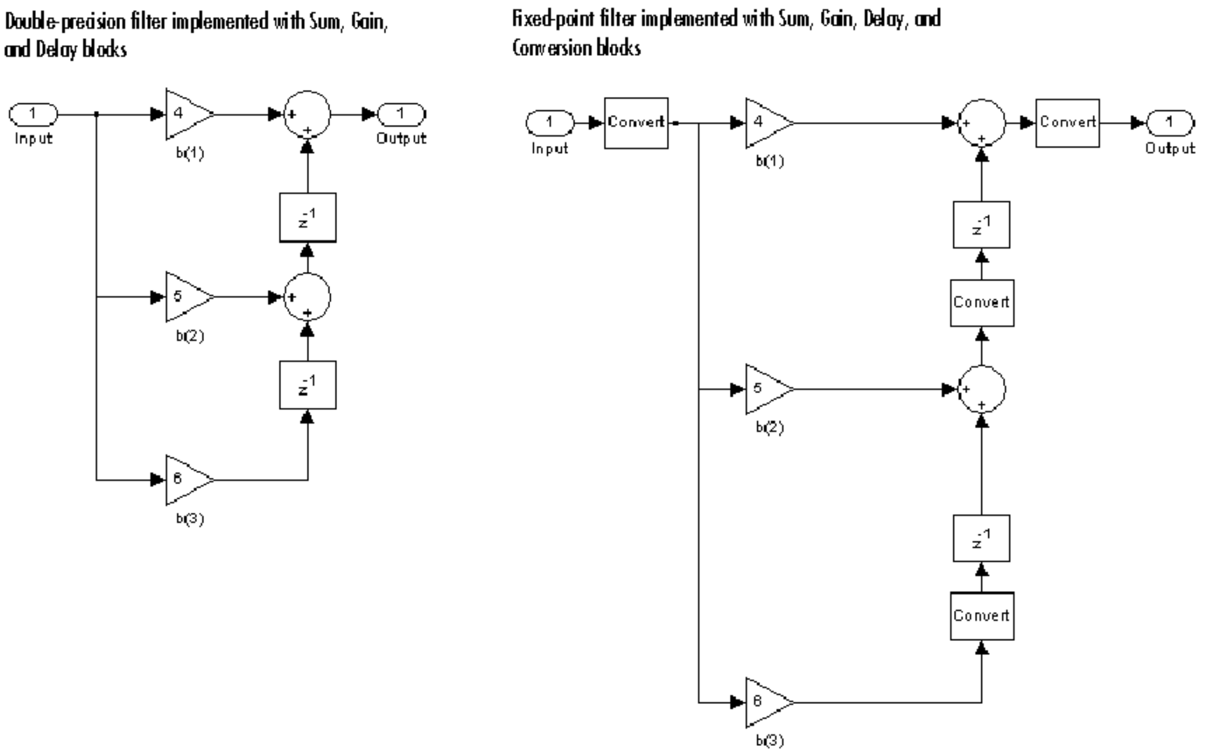
- Direct form I
- Direct form I, second-order sections
- Direct form I transposed
- Direct form I transposed, second-order sections
- Direct form II
- Direct form II, second-order sections
- Direct form II transposed
- Direct form II transposed, second-order sections
- Direct form FIR
- Direct form FIR transposed
- Direct form symmetric FIR
- Direct form antisymmetric FIR
- Lattice all-pass
- Lattice AR

- Lattice ARMA
- Lattice MA for maximum phase
- Lattice MA for minimum phase
- Cascade
- Parallel

Specify the Filter Implementation

You can determine how the Filter Realization Wizard models the specified filter. In the **Realize Model** panel, select the **Build model using basic elements** check box. When you select this check box and click on the **Realize Model** button, the Filter Realization Wizard creates a subsystem block that implements your filter using Sum, Gain, and `Delay` blocks. When you clear this check box, the Filter Realization Wizard uses a digital filter block to implement your filter. The **Build model using basic elements** check box is available only when your filter can be implemented using a digital filter block available in the DSP System Toolbox library.

The Filter Realization Wizard can generate a subsystem that represents either a double-precision or fixed-point filter. You must install the Fixed-Point Designer product to simulate a fixed-point filter. You can still edit the blocks used to implement the filter without installing the Fixed-Point Designer product.



Implementations of Double-Precision and Fixed-Point Filters

Command Line Alternative to Realize Model Button

You can enter `realizemdl(`*`sysobj`*`)` in the MATLAB command prompt to generate an architectural model of the filter System object, *`sysobj`*, in a Simulink subsystem block using individual sum, gain, and delay blocks, according to user-defined specifications. For more information, see `realizemdl`.

Parameters

Note: The following parameters for the Filter Realization Wizard are in the **Realize Model** pane of the filter designer app. To open different panels of filter designer, click the different buttons at the lower-left corner. For more information about relevant panels, see “Specify the Filter and Data Types” on page 2-680.

Block Name

Enter the name of the new filter block.

Destination

Specify where the new filter block should be created. This can be in a new model or in the current (most recently selected) model.

User Defined

Specify the name of the target subsystem in which the Filter Realization Wizard should create the new filter block.

Overwrite generated block “Filter” block

When selected, the block overwrites any filter block in the current model with the name specified in the **Block Name** parameter. This parameter is enabled when the **Destination** parameter is set to `Current`.

Build model using basic elements

Select this check box to implement your filter using Sum, Gain, and Delay blocks. Clear this check box to implement your filter using digital filter blocks from the DSP System Toolbox library. This parameter is available only when your filter can be modeled using an available digital filter block.

Optimize for zero gains

Select this check box to remove zero-gain paths from the filter structure. For an example, see “Optimize the Filter Structure”.

Optimize for unity gains

Select this check box to substitute gains equal to 1 with a wire (short circuit). For an example, see “Optimize the Filter Structure”.

Optimize for negative gains

Select this check box to substitute gains equal to -1 with a wire (short circuit), and change the corresponding sums to subtractions. For an example, see “Optimize the Filter Structure”.

Optimize delay chains

Select this check box to substitute any delay chains made up of n unit delays with a single delay by n. For an example, see “Optimize the Filter Structure”.

Optimize for unity scale values

Select this check box to remove all scale value multiplications by 1 from the filter structure.

Input processing

Specify how the generated filter block or subsystem block processes the input. Depending on the type of filter you are designing, one or both of the following options may be available:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

For more information about sample- and frame-based processing, see “Sample- and Frame-Based Concepts”.

Rate options

For multirate filters, specify how the block should process the input. You can set this parameter to one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples.

Realize Model

Click to create a filter block according to the settings you’ve specified. When the **Build model using basic elements** check box is selected, the filter is implemented as a subsystem block consisting of Sum, Gain, and Delay blocks. To see the filter implementation, double-click the subsystem block in your model.

Note: For more information about relevant parameters in other panels of filter designer, see “Specify the Filter and Data Types” on page 2-680.

Supported Data Types

- Double-precision floating point

- Single-precision floating point — Supported only when you install Fixed-Point Designer.
- Fixed point (signed and unsigned) — Supported only when you install Fixed-Point Designer and Fixed-Point Designer.

References

[1] Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[2] Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

See Also

See Also

Functions
`filter` | `realizemdl`

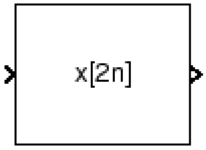
Blocks
`Biquad Filter` | `Digital Filter Design` | `Discrete FIR Filter`

Topics
"Filter Design"
"Filter Analysis"
"Select a Filter Design Block"

Introduced before R2006a

FIR Decimation

Filter and downsample input signals



Library

Filtering / Multirate Filters

`dspmlti4`

Description

The FIR Decimation block resamples the discrete-time input at a rate *K* times slower than the input sample rate, where *K* is the integer value you specify for the **Decimation factor** parameter. To do so, the block implements a polyphase filter structure and performs the following operations:

1. Filters the data in each channel of the input using a direct-form FIR filter.
2. Downsamples each channel of filtered data by discarding *K*–1 consecutive samples following each sample that is retained.

The block uses a polyphase filter implementation because it is more efficient than straightforward filter-then-decimate algorithms. See Fliege [1] for more information.

You can use the FIR Decimation block inside triggered subsystems when you set the **Rate options** parameter to `Enforce single-rate processing`.

This block supports variable-size input. This means that while the block is simulating, the frame size (number of rows) can change. The output dimensions always equal those of the input signal.

Specifying the Filter Coefficients

To specify the filter coefficients, select the mode you want the FIR Decimation block to operate in. Select the mode in the **Coefficient source** group box.

- **Dialog parameters** — Enter information about the filter, such as structure and coefficients, in the block dialog box.
- **Input port** — Specify the filter coefficients as an input to the block. Coefficient values are tunable (can change during simulation), while their properties must remain constant.
- **Filter object** — Specify the filter using a dsp.FIRDecimator System object.
- **Auto** (default) — Choose the filter coefficients of an FIR Nyquist filter, predesigned for the decimation factor specified in the block dialog box.

When you select **Dialog parameters**, you use the **FIR filter coefficients** parameter to specify the numerator coefficients of the FIR filter transfer function $H(z)$.

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

You can generate the FIR filter coefficient vector, `[b(1) b(2) ... b(m)]`, using one of the DSP System Toolbox filter design functions such as `designMultirateFIR`, `firnyquist`, `firhalfband`, `firgr`, or `firceqrip`.

The filter you specify must be a lowpass filter with a normalized cutoff frequency no greater than $1/K$. The block internally initializes all filter states to zero.

When you select **Auto**, the block designs an FIR decimator with the decimation factor specified in **Decimation factor**. The `designMultirateFIR` function designs the filter and returns the coefficients used by the block. For more information on the filter design, see Orfanidis [2].

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block resamples each column of the input over time. In this mode, the block can perform either single-rate or multirate processing. You can use the **Rate options** parameter to specify how the block resamples the input:

- When you set the **Rate options** parameter to `Enforce single-rate processing`, the input and output of the block have the same sample rate. To decimate the output while maintaining the input sample rate, the block resamples the data in each column of the input such that the frame size of the output (M_o) is $1/K$ times that of the input ($M_o = M_i/K$),

  In this mode, the input frame size, M_i, must be a multiple of the **Decimation factor**, K.

  For an example of single-rate FIR Decimation, see “Example 1 — Single-Rate Processing” on page 2-691.
- When you set the **Rate options** parameter to `Allow multirate processing`, the input and output of the FIR Decimation block are the same size, but the sample rate of the output is K times slower than that of the input. In this mode, the block treats an M_i-by-N matrix input as N independent channels. The block decimates each column of the input over time by keeping the frame size constant (M_i=M_o), and making the output frame period (T_{fo}) K times longer than the input frame period ($T_{fo} = K*T_{fi}$).

  See “Example 2— Multirate Frame-Based Processing” on page 2-691 for an example that uses the FIR Decimation block in this mode.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an M-by-N matrix input as $M*N$ independent channels, and decimates each channel over time. The output sample period (T_{so}) is K times longer than the input sample period ($T_{so} = K*T_{si}$), and the input and output sizes are identical.

Latency

When you use the FIR Decimation block in sample-based processing mode, the block always has zero-tasking latency. *Zero-tasking latency* means that the block propagates the first filtered input sample (received at time t= 0) as the first output sample. That first output sample is then followed by filtered input samples K+1, 2K+1, and so on.

When you use the FIR Decimation block in frame-based processing mode with a frame size greater than one, the block may exhibit *one-frame latency*. Cases of one-frame latency can occur when the input frame size is greater than one, and you set the **Input processing** and **Rate options** parameters of the FIR Decimation block as follows:

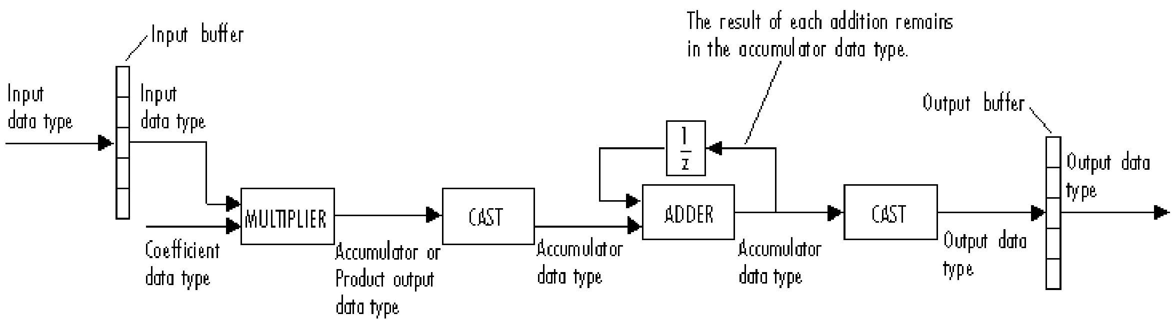
- **Input processing** = `Columns as channels (frame based)`
- **Rate options** = `Allow multirate processing`

In cases of one-frame latency, you can define the value of the first M_i output rows by setting the **Output buffer initial conditions** parameter. The default value of the **Output buffer initial conditions** parameter is `0`. However, you can enter a matrix containing one value for each channel of the input, or a scalar value to be applied to all channels. The first filtered input sample (first filtered row of the input matrix) appears in the output as sample M_i+ 1. That sample is then followed by filtered input samples K+ 1, $2K$+ 1, and so on.

Note: For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

Fixed-Point Data Types

The following diagram shows the data types used within the FIR Decimation block for fixed-point signals.



You can set the coefficient, product output, accumulator, and output data types in the block dialog box as discussed in the “Dialog Box” on page 2-692 section. This diagram shows that data is stored in the input buffer with the same data type and scaling as the input. The block stores filtered data and any initial conditions in the output buffer using the output data type and scaling that you set in the block dialog box.

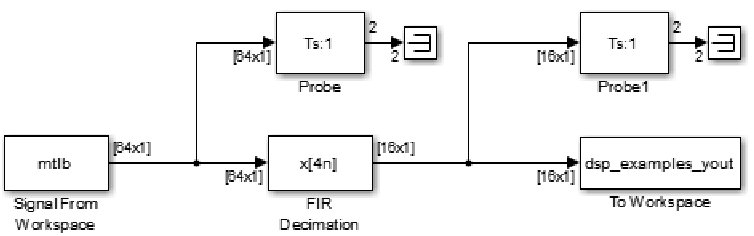
When at least one of the inputs to the multiplier is real, the output of the multiplier is in the product output data type. When both inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed by this block, see "Multiplication Data Types".

Note: When the block input is fixed point, all internal data types are signed fixed-point values.

Examples

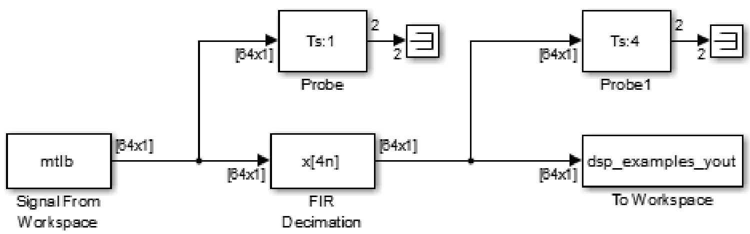
Example 1 — Single-Rate Processing

In the ex_firdecimation_ref2 model, the FIR Decimation block decimates a single-channel input with a frame size of 64. Because the block is doing single-rate processing and the **Decimation factor** parameter is set to 4, the output of the FIR Decimation block has a frame size of 16. As shown in the following figure, the input, and output of the FIR Decimation block have the same sample rate.



Example 2— Multirate Frame-Based Processing

In the ex_firdecimation_ref1 model, the FIR Decimation block decimates a single-channel input with a frame period of one second. Because the block is doing multirate frame-based processing and the **Decimation factor** parameter is set to 4, the frame period of the output is 4 seconds. As shown in the following figure, the input, and output of the FIR Decimation block have the same frame size, but the sample rate of the output is four times that of the input.



Example 3

The ex_polyphasedec model illustrates the underlying polyphase implementations of the FIR Decimation block. Run the model, and view the results on the scope. The output of the FIR Decimation block matches the output of the Polyphase Decimation Filter block.

Example 4

The ex_mrf_nlp model illustrates the use of the FIR Decimation block in several multistage multirate filters.

Dialog Box

Coefficient Source

The FIR Decimation block can operate in four different modes. Select the mode in the **Coefficient source** group box.

- **Dialog parameters** — Enter information about the filter, such as structure and coefficients, in the block mask.
- **Input port** — Specify the filter coefficients with a **Num** input port. The **Num** input port appears when you select the **Input port** option. Coefficient values obtained

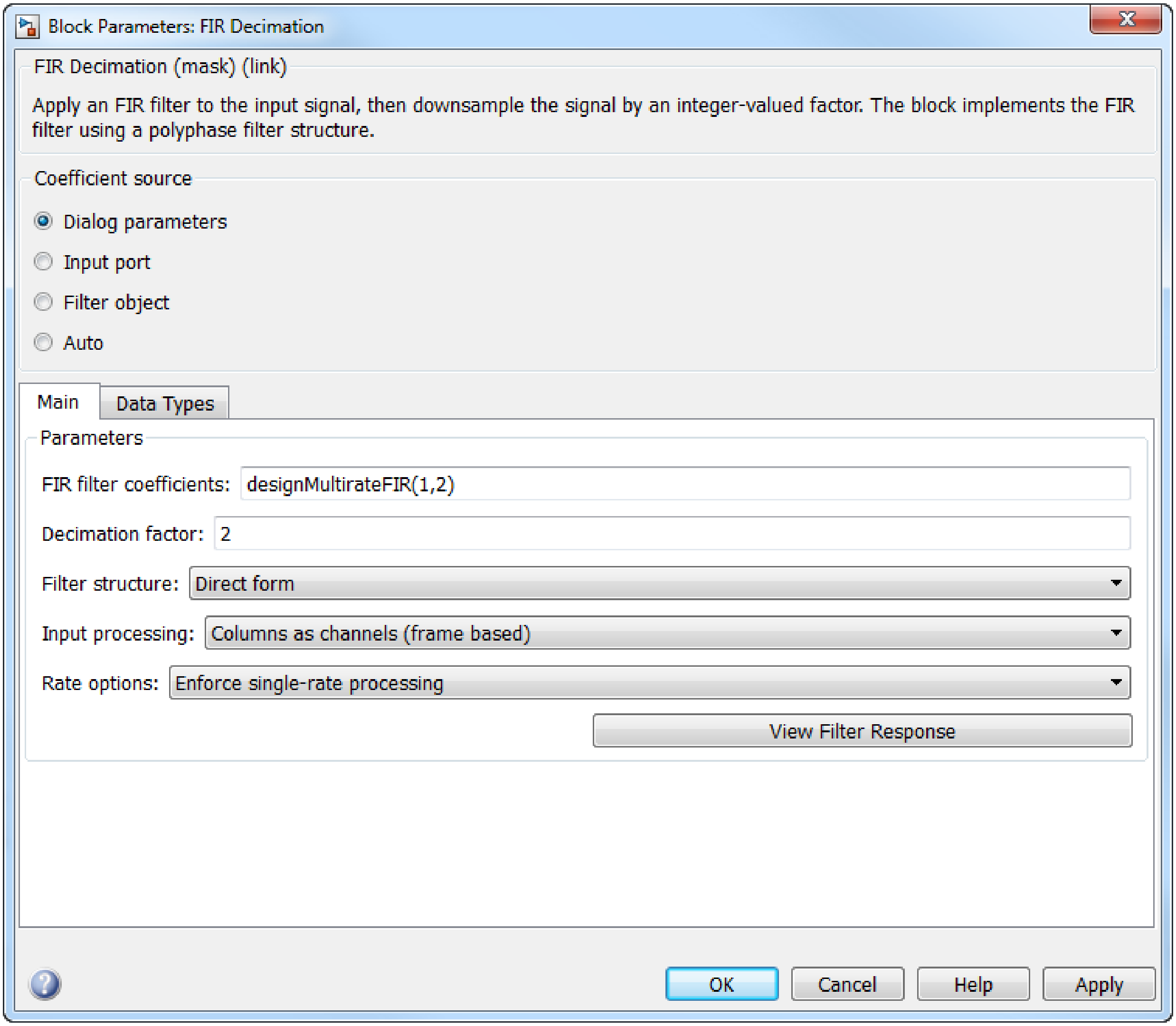
through **Num** are tunable (can change during simulation), while their properties must remain constant.

- **Filter object** — Specify the filter using a dsp.FIRDecimator System object.
- **Auto** (default) — Choose the coefficients of an FIR Nyquist filter, predesigned for the decimation factor specified in the block dialog box.

Different items appear on the FIR Decimation block dialog box depending on whether you select **Dialog parameters**, **Input port**, **Filter object**, or **Auto** in the **Coefficient source** group box.

Specify Filter Characteristics in Dialog Box

The **Main** pane of the FIR Decimation block dialog box appears as follows when you select **Dialog parameters** in the **Coefficient source** group box.



FIR filter coefficients

Specify the lowpass FIR filter coefficients, in descending powers of z. By default, `designMultirateFIR(1,2)` computes the filter coefficients.

Decimation factor

Specify the integer factor, *K*, by which to decrease the sample rate of the input sequence. The default is `2`.

Filter Structure

Choose whether to implement a `Direct form` (default) or `Direct form transposed` filter.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

Specify the method by which the block should decimate the input. You can select one of the following options:

- `Enforce single-rate processing` (default) — When you select this option, the block maintains the input sample rate and decimates the signal by decreasing the output frame size by a factor of *K*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block decimates the signal such that the output sample rate is *K* times slower than the input sample rate.

Output buffer initial conditions

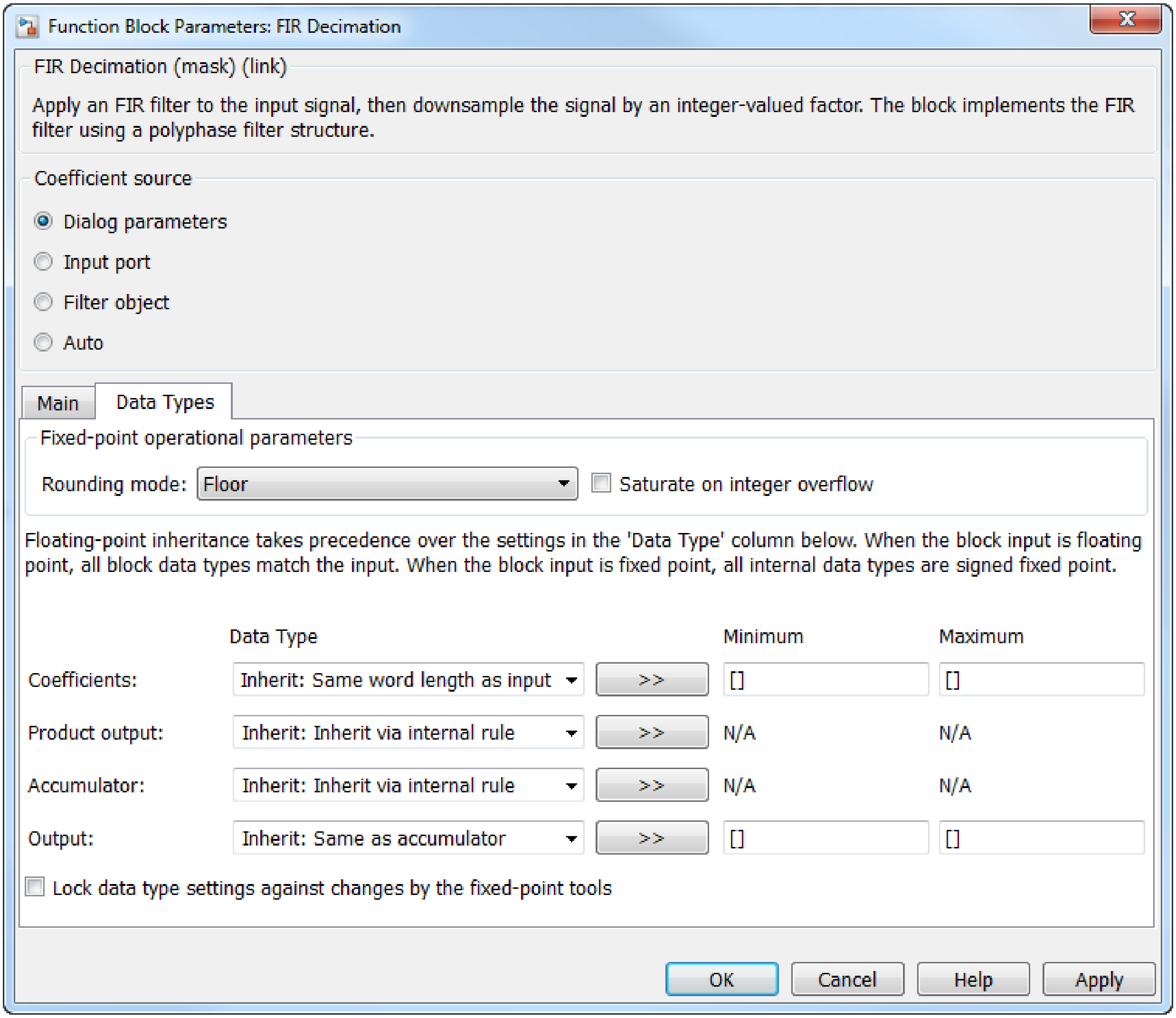
In the case of *one-frame latency*, this parameter specifies the output of the block until the first filtered input sample is available. The default value of this parameter is `0`, but you can enter a matrix containing one value for each channel, or a scalar value to be applied to all signal channels. This parameter appears only when you configure the block to perform multirate processing.

See “Latency” on page 2-689 for more information about latency in the FIR Decimation block.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the filter defined in the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

The **Data Types** pane of the FIR Decimation block dialog box appears as follows when you select **Dialog parameters** in the **Coefficient source** group box.



Rounding mode

Select the rounding mode for fixed-point operations. The default is `Floor`. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Note: The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output data type** is `Inherit: Inherit via internal rule`
- **Accumulator data type** is `Inherit: Inherit via internal rule`
- **Output data type** is `Inherit: Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients data type

Specify the coefficients data type. See “Fixed-Point Data Types” on page 2-690 and “Multiplication Data Types” for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Coefficients data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Coefficients Minimum

Specify the minimum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Coefficients Maximum

Specify the maximum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-690 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-690 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-690 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

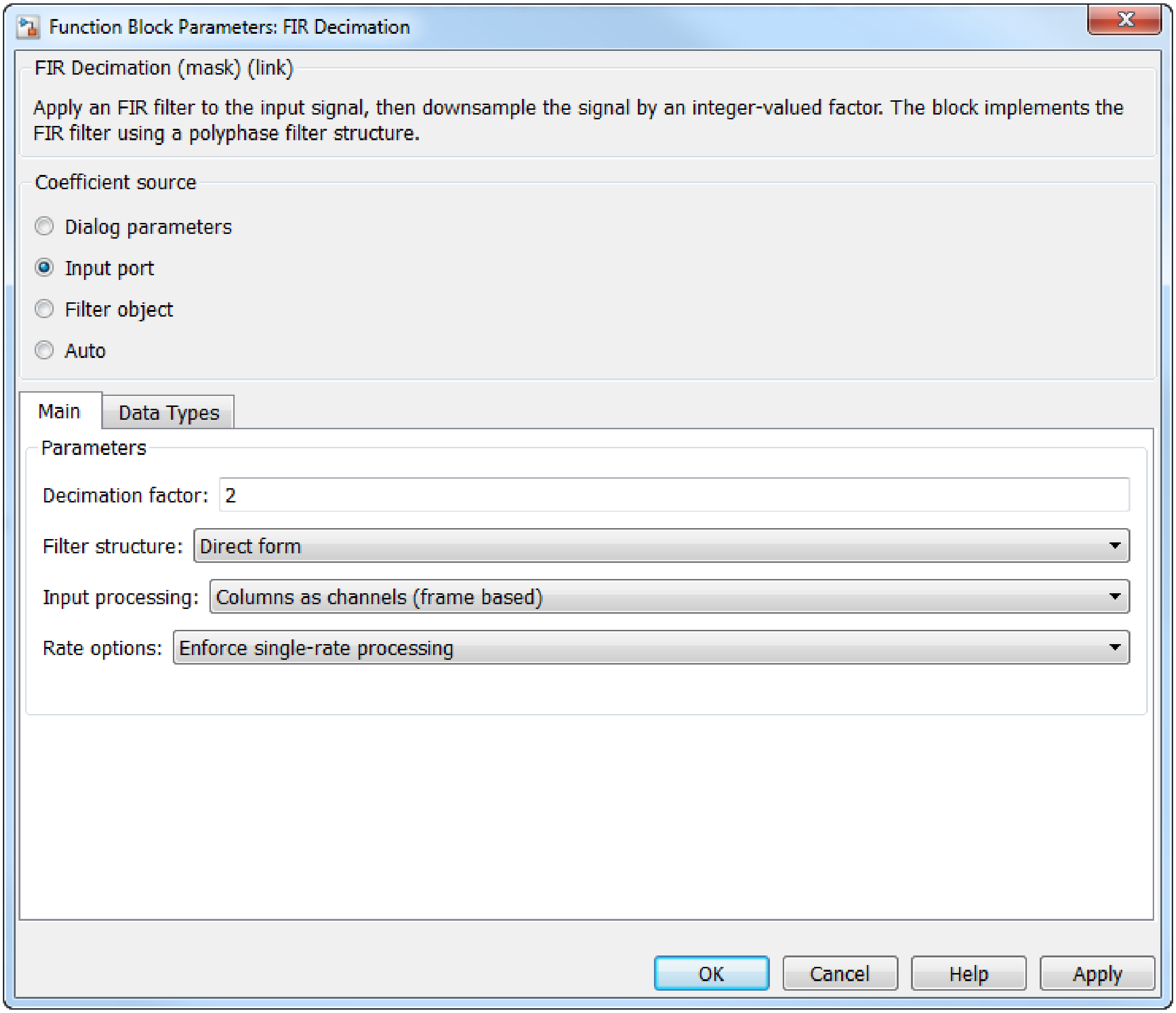
- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Provide Filter Coefficients Through Input Port

The **Main** pane of the FIR Decimation block dialog box appears as follows when you select **Input port** in the **Coefficient source** group box.



Decimation factor

Specify the integer factor, *K*, by which to decrease the sample rate of the input sequence. The default is 2.

Filter Structure

Choose whether to implement a `Direct form` (default) or `Direct form transposed` filter.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

Specify the method by which the block should decimate the input. You can select one of the following options:

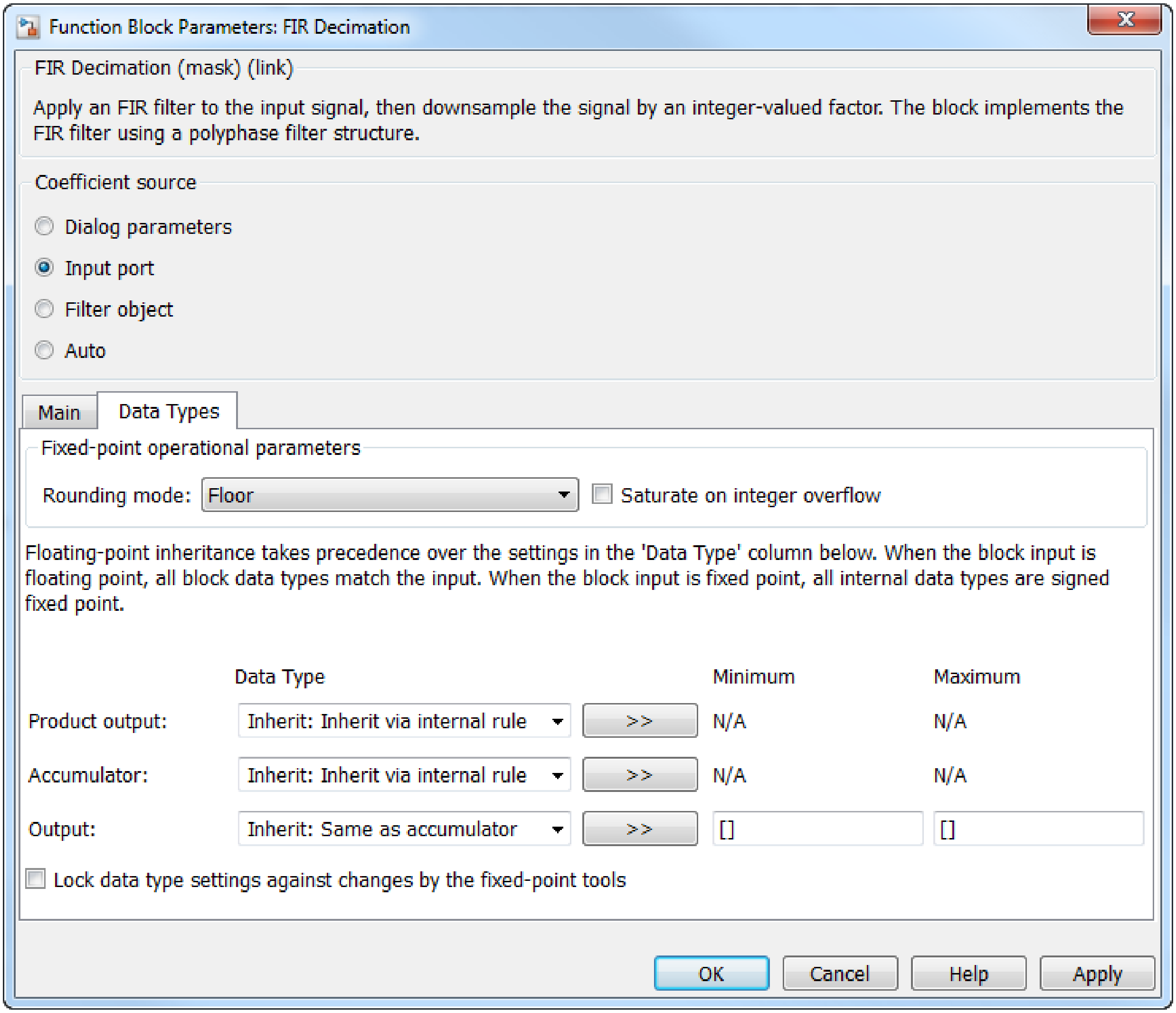
- `Enforce single-rate processing` (default) — When you select this option, the block maintains the input sample rate and decimates the signal by decreasing the output frame size by a factor of K. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block decimates the signal such that the output sample rate is K times slower than the input sample rate.

Output buffer initial conditions

In the case of *one-frame latency*, this parameter specifies the output of the block until the first filtered input sample is available. The default value of this parameter is `0`, but you can enter a matrix containing one value for each channel, or a scalar value to be applied to all signal channels. This parameter appears only when you configure the block to perform multirate processing.

See "Latency" on page 2-689 for more information about latency in the FIR Decimation block.

The **Data Types** pane of the FIR Decimation block dialog box appears as follows when you select **Input port** in the **Coefficient source** group box.



Rounding mode

Select the rounding mode for fixed-point operations. The default is `Floor`. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Note: The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output data type** is `Inherit: Inherit via internal rule`
- **Accumulator data type** is `Inherit: Inherit via internal rule`
- **Output data type** is `Inherit: Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-690 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button `>>` to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-690 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-690 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

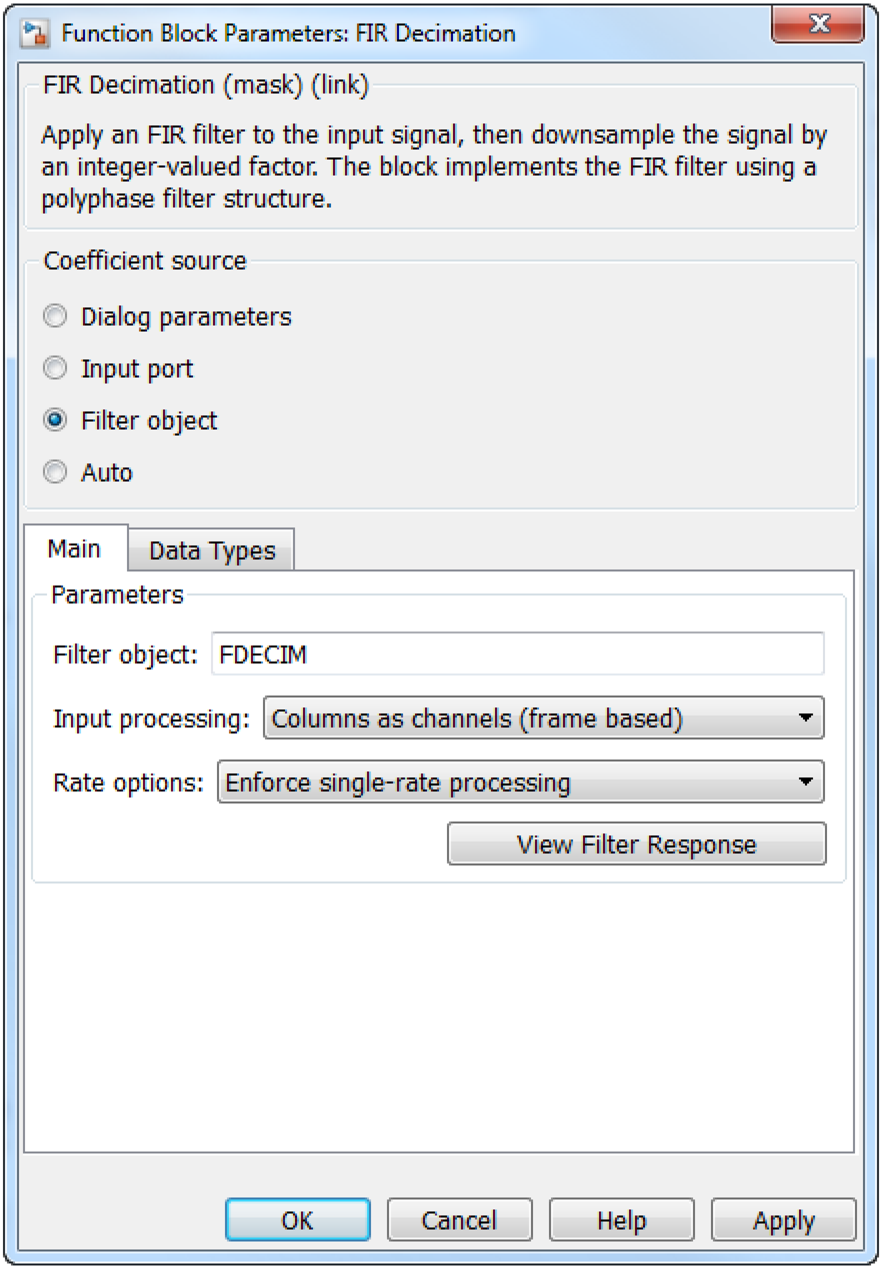
- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Specify Multirate Filter Object

The **Main** pane of the FIR Decimation block dialog box appears as follows when you select **Filter object** in the **Coefficient source** group box.


Function Block Parameters: FIR Decimation
FIR Decimation (mask) (link)
Apply an FIR filter to the input signal, then downsample the signal by an integer-valued factor. The block implements the FIR filter using a polyphase filter structure.
Coefficient source
Dialog parameters
Input port
Filter object
Auto
Main
Data Types
Parameters
Filter object:
FDECIM
Input processing:
Columns as channels (frame based)
Rate options:
Enforce single-rate processing
View Filter Response
OK
Cancel
Help
Apply

Filter object

Specify the name of the multirate filter object that you want the block to implement. You must specify the filter as a dsp.FIRDecimator System object.

You can define the System object in the block mask or in a MATLAB workspace variable.

For information on creating System objects, see “Define Basic System Objects” (MATLAB).

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

Specify the method by which the block should decimate the input. You can select one of the following options:

- `Enforce single-rate processing` (default) — When you select this option, the block maintains the input sample rate and decimates the signal by decreasing the output frame size by a factor of *K*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block decimates the signal such that the output sample rate is *K* times slower than the input sample rate.

Output buffer initial conditions

In the case of *one-frame latency*, this parameter specifies the output of the block until the first filtered input sample is available. The default value of this parameter is `0`, but you can enter a matrix containing one value for each channel, or a scalar value to be applied to all signal channels. This parameter appears only when you configure the block to perform multirate processing.

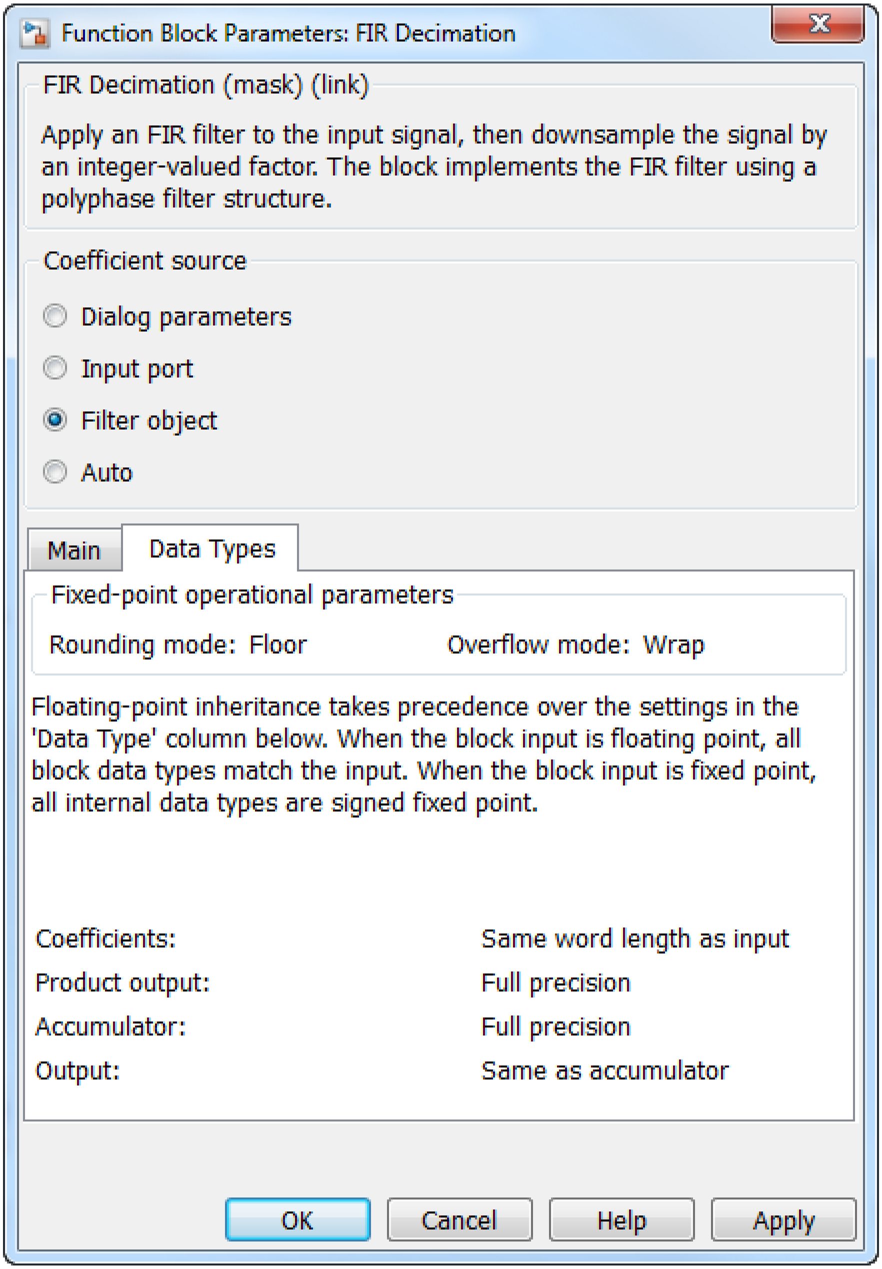
See “Latency” on page 2-689 for more information about latency in the FIR Decimation block.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the System object specified in the **Filter object** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note: If you specify a filter in the **Filter object** parameter, you must apply the filter by clicking the **Apply** button before using the **View filter response** button.

The **Data Types** pane of the FIR Decimation block dialog box appears as follows when you select **Filter object** in the **Coefficient source** group box.


Function Block Parameters: FIR Decimation
FIR Decimation (mask) (link)
Apply an FIR filter to the input signal, then downsample the signal by an integer-valued factor. The block implements the FIR filter using a polyphase filter structure.
Coefficient source
Dialog parameters
Input port
Filter object
Auto
Main
Data Types
Fixed-point operational parameters
Rounding mode: Floor
Overflow mode: Wrap
Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input. When the block input is fixed point, all internal data types are signed fixed point.
Coefficients:
Same word length as input
Product output:
Full precision
Accumulator:
Full precision
Output:
Same as accumulator
OK
Cancel
Help
Apply

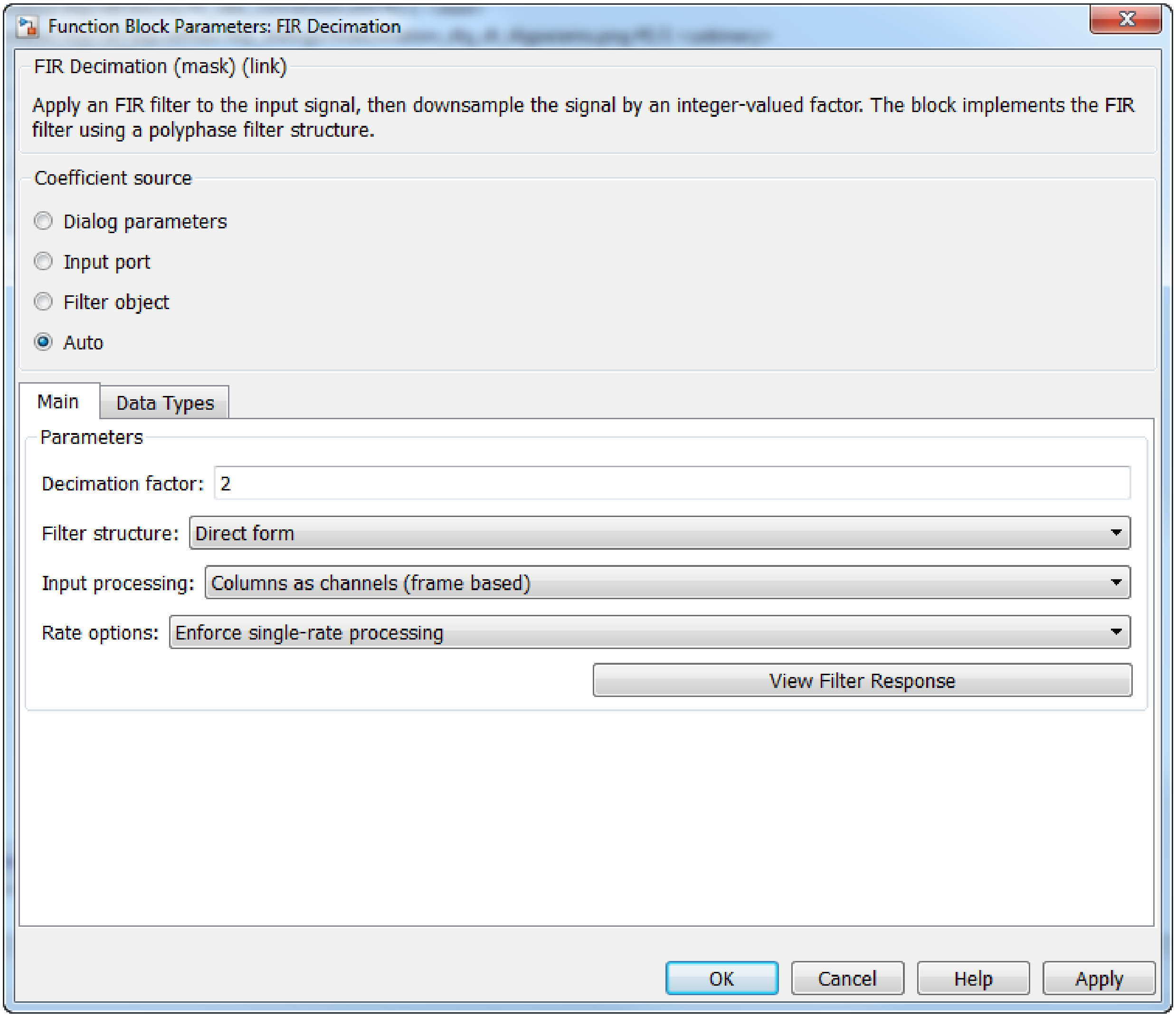
The fixed-point settings of the filter object specified on the **Main** pane are displayed on the **Data Types** pane. You cannot change these settings directly on the block mask. To change the fixed-point settings, edit the filter object directly.

For more information on System objects, see “What Are System Objects?” (MATLAB).

Choose Filter Coefficients Automatically

When you select **Auto** in the **Coefficient source** group box, the block chooses the filter coefficients automatically. For more information on the filter design algorithm the block uses, see “Specifying the Filter Coefficients” on page 2-688.

The **Main** pane of the FIR Decimation block dialog box appears as follows when you select **Auto** in the **Coefficient source** group box.



Decimation factor

Specify the integer factor, K, by which to decrease the sample rate of the input sequence. The default is 2.

Filter Structure

Choose whether to implement a `Direct form` (default) or `Direct form transposed` filter.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

Specify the method by which the block should decimate the input. You can select one of the following options:

- `Enforce single-rate processing` (default) — When you select this option, the block maintains the input sample rate and decimates the signal by decreasing the output frame size by a factor of K. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block decimates the signal such that the output sample rate is K times slower than the input sample rate.

Output buffer initial conditions

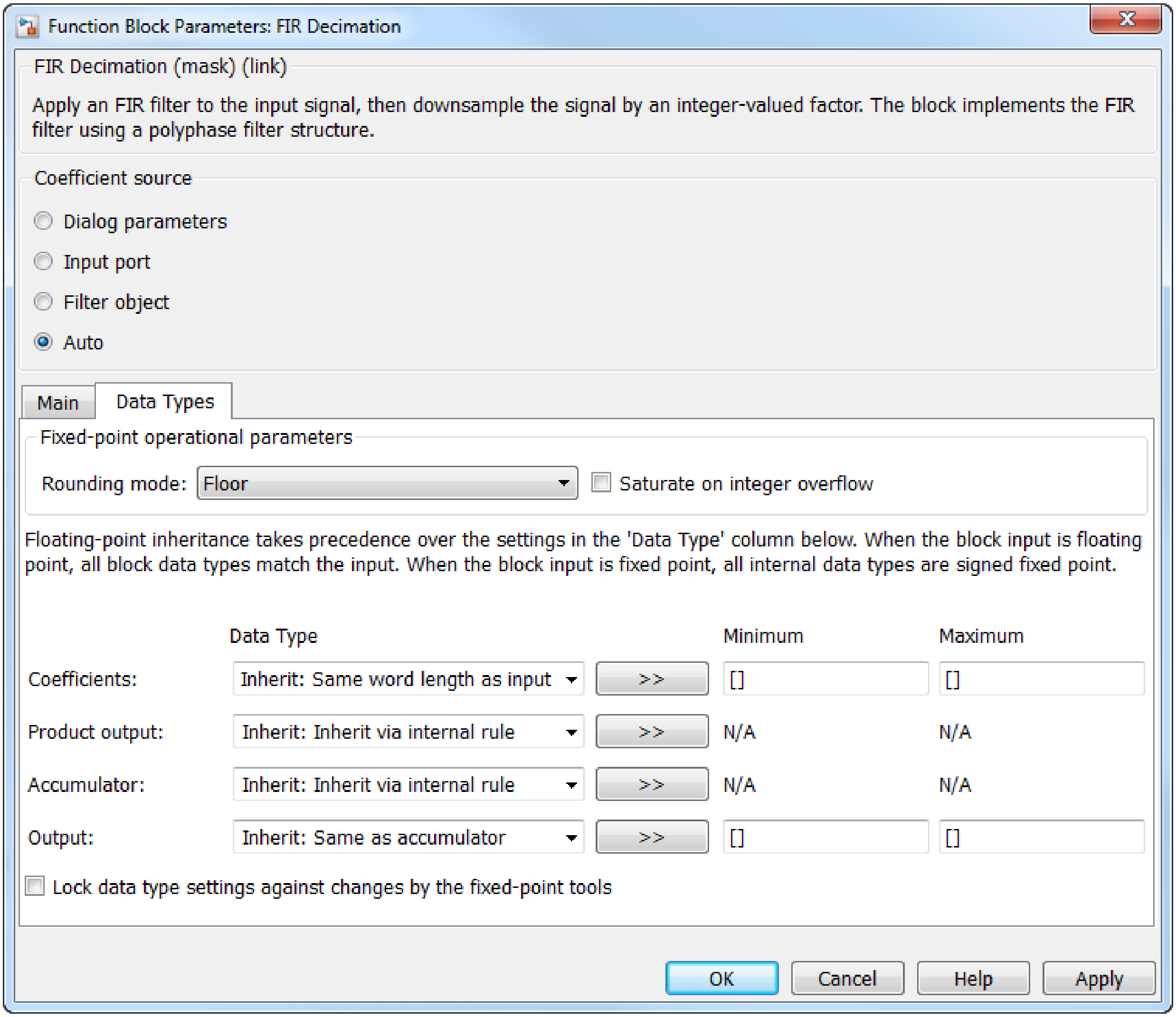
In the case of *one-frame latency*, this parameter specifies the output of the block until the first filtered input sample is available. The default value of this parameter is `0`, but you can enter a matrix containing one value for each channel, or a scalar value to be applied to all signal channels. This parameter appears only when you configure the block to perform multirate processing.

See "Latency" on page 2-689 for more information about latency in the FIR Decimation block.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the filter defined in the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

The **Data Types** pane of the FIR Decimation block dialog box appears as follows when you select **Auto** in the **Coefficient source** group box.



Rounding mode

Select the rounding mode for fixed-point operations. The default is `Floor`. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Note: The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output data type** is `Inherit: Inherit via internal rule`
- **Accumulator data type** is `Inherit: Inherit via internal rule`
- **Output data type** is `Inherit: Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients data type

Specify the coefficients data type. See “Fixed-Point Data Types” on page 2-690 and “Multiplication Data Types” for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Coefficients data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Coefficients Minimum

Specify the minimum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Coefficients Maximum

Specify the maximum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-690 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-690 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-690 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see FIR Decimation.

References

[1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

[2] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

`Downsample`	DSP System Toolbox
`FIR Interpolation`	DSP System Toolbox
`FIR Rate Conversion`	DSP System Toolbox
`CIC Decimation`	DSP System Toolbox
`Digital Down-Converter`	DSP System Toolbox
`FIR Halfband Interpolator`	DSP System Toolbox
`FIR Halfband Decimator`	DSP System Toolbox
`IIR Halfband Interpolator`	DSP System Toolbox
`IIR Halfband Decimator`	DSP System Toolbox
`CIC Compensation Interpolator`	DSP System Toolbox

`CIC Compensation Decimator`	DSP System Toolbox
`Digital Up-Converter`	DSP System Toolbox
dsp.FIRDecimator	DSP System Toolbox
dsp.CICCompensationDecimator	DSP System Toolbox
dsp.FIRHalfbandDecimator	DSP System Toolbox
`firnyquist`	DSP System Toolbox
`firhalfband`	DSP System Toolbox
`firgr`	DSP System Toolbox
`firceqrip`	DSP System Toolbox

Introduced before R2006a

Discrete FIR Filter

Model FIR filters

Library

Filtering / Filter Implementations

`dsparch4`

Description

This block is the same as the Simulink Discrete FIR Filter block. For more information, see the Discrete FIR Filter reference page in the Simulink documentation.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Discrete FIR Filter.

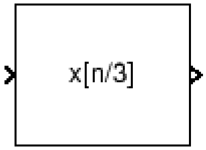
See Also

dsp.FIRFilter

Introduced in R2011a

FIR Interpolation

Upsample and filter input signals



Library

Filtering / Multirate Filters

`dspmlti4`

Description

The FIR Interpolation block resamples the discrete-time input at a rate L times faster than the input sample rate, where L is the integer value you specify for the **Interpolation factor** parameter. To do so, the block implements a polyphase filter structure and performs the following operations:

- Upsamples each channel of the input to a higher rate by inserting L–1 zeros between samples.
- Filters each channel of the upsampled data using a direct-form FIR filter.

The block uses a polyphase filter implementation because it is more efficient than straightforward upsample-then-filter algorithms. See Fliege [1] for more information.

You can use the FIR Interpolation block inside triggered subsystems when you set the **Rate options** parameter to `Enforce single-rate processing`.

Specifying the Filter Coefficients

To specify the filter coefficients, select the mode you want the FIR Interpolation block to operate in. Select the mode in the **Coefficient source** group box.

- **Dialog parameters** — Enter information about the filter, such as coefficients in the block dialog box.

- **Input port** — Specify the filter coefficients as an input to the block. Coefficient values are tunable (can change during simulation), while their properties must remain constant.
- **Filter object** — Specify the filter using a dsp.FIRInterpolator System object.
- **Auto** (default) — Choose the filter coefficients of an FIR Nyquist filter, predesigned for the interpolation factor specified in the block dialog box.

When you select **Dialog parameters**, you use the **FIR filter coefficients** parameter to specify the numerator coefficients of the FIR filter transfer function $H(z)$.

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

You can generate the FIR filter coefficient vector, `[b(1) b(2) ... b(m)]`, using one of the DSP System Toolbox filter design functions such as `designMultirateFIR`, `firnyquist`, `firhalfband`, `firgr` or `firceqrip`.

The filter you specify must be a lowpass filter with a length greater than the interpolation factor (m>L) and a normalized cutoff frequency no greater than **1**/L. The block internally initializes all filter states to zero.

When you select **Auto**, the block designs an FIR interpolator with the interpolation factor specified in **Interpolation factor**. The `designMultirateFIR` function designs the filter and returns the coefficients used by the block. For more information on the filter design, see Orfanidis [2].

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block resamples each column of the input over time. In this mode, the block can perform either single-rate or multirate processing. You can use the **Rate options** parameter to specify how the block resamples the input:

- When you set the **Rate options** parameter to `Enforce single-rate processing`, the input and output of the block have the same sample rate. To interpolate the output while maintaining the input sample rate, the block resamples the data in each column of the input such that the frame size of the output (M_o) is L times larger than that of the input ($M_o = M_i * L$).

  For an example of single-rate FIR Interpolation, see “Example 1 — Single-Rate Processing” on page 2-724.

- When you set the **Rate options** parameter to `Allow multirate processing`, the input and output of the FIR Interpolation block are the same size. However, the sample rate of the output is L times faster than that of the input. In this mode, the block treats an M_i-by-N matrix input as N independent channels. The block interpolates each column of the input over time by keeping the frame size constant (M_i=M_o), while making the output frame period (T_{fo}) L times shorter than the input frame period ($T_{fo} = T_{fi}/L$).

  See "Example 2 — Multirate Frame-Based Processing" on page 2-725 for an example that uses the FIR Interpolation block in this mode.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an M-by-N matrix input as $M*N$ independent channels, and interpolates each channel over time. The output sample period (T_{so}) is L times shorter than the input sample period ($T_{so} = T_{si}/L$), while the input and output sizes remain identical.

Latency

When you run your models in Simulink `SingleTasking` mode or set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`, the FIR Interpolation block always has zero-tasking latency. *Zero-tasking latency* means that the block propagates the first filtered input sample (received at time t=0) as the first output sample. That first output sample is then followed by L–1 interpolated values, the second filtered input sample, and so on.

The only time the FIR Interpolation block exhibits latency is when you set the **Rate options** parameter set to `Allow multirate processing` and run your models in Simulink `MultiTasking` mode. The amount of latency for multirate, multitasking operation depends on the setting of the **Input processing** parameter, as shown in the following table.

Input processing	Latency
`Elements as channels (sample based)`	L samples

Input processing	Latency
`Columns as channels (frame based)`	L frames (M_i samples per frame)

When the block exhibits latency, the default initial condition is zero. Alternatively, you can use the **Output buffer initial conditions** parameter to specify a matrix of initial conditions containing one value for each channel or a scalar initial condition to be applied to all channels. The block scales the **Output buffer initial conditions** by the **Interpolation factor** and outputs the scaled initial conditions until the first filtered input sample becomes available.

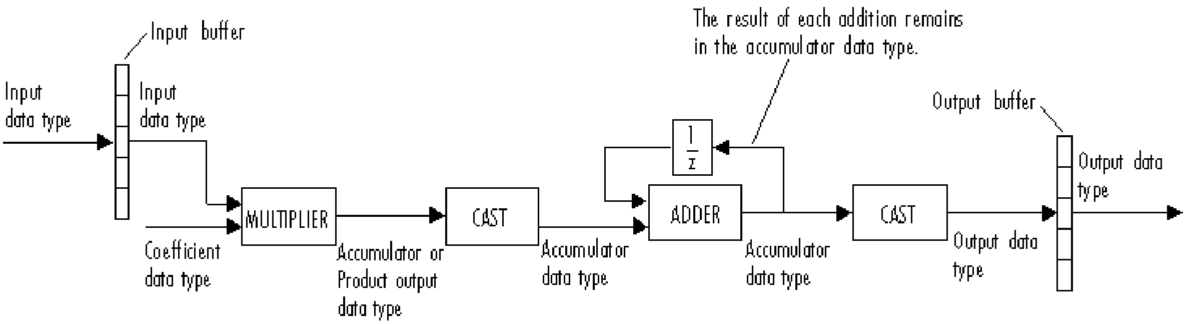
When the block is in sample-based processing mode, the block outputs the scaled initial conditions at the start of each channel, followed immediately by the first filtered input sample, then L–1 interpolated values, and so on.

When the block is in frame-based processing mode and using the default initial condition of zero, the first M_i*L output rows contain zeros, where M_i is the input frame size. The first filtered input sample (first filtered row of the input matrix) appears in the output as sample M_i*L+1. That value is then followed by L–1 interpolated values, the second filtered input sample, and so on.

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Time-Based Scheduling and Code Generation" (Simulink Coder).

Fixed-Point Data Types

The following diagram shows the data types used within the FIR Interpolation block for fixed-point signals.



You can set the coefficient, product output, accumulator, and output data types in the block dialog box as discussed in "Dialog Box" on page 2-726 section. This diagram shows that input data is stored in the input buffer with the same data type and scaling as the input. The block stores filtered data and any initial conditions in the output buffer using the output data type and scaling that you set in the block dialog box.

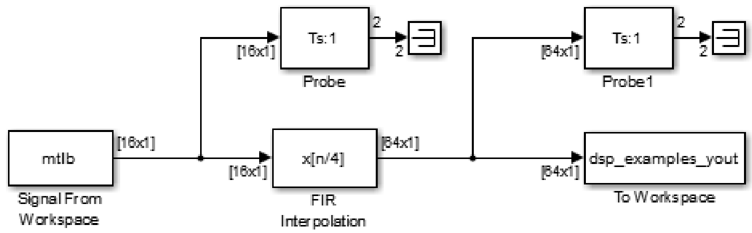
When at least one of the inputs to the multiplier is real, the output of the multiplier is in the product output data type. When both inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed by this block, see "Multiplication Data Types".

Note: When the block input is fixed point, all internal data types are signed fixed point.

Examples

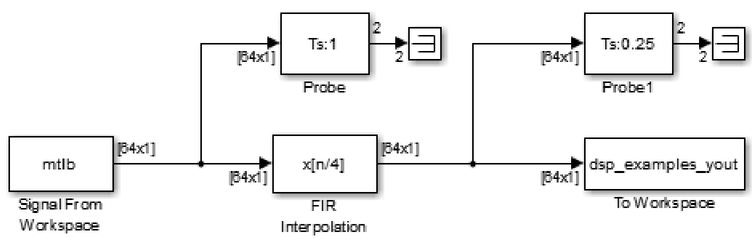
Example 1 — Single-Rate Processing

In the ex_firinterpolation_ref2, the FIR Interpolation block interpolates a single-channel input with a frame size of 16. Because the block is doing single-rate processing and the **Interpolation factor** parameter is set to 4, the output of the FIR Interpolation block has a frame size of 64. As shown in the following figure, the input, and output of the FIR Interpolation block have the same sample rate.



Example 2 — Multirate Frame-Based Processing

In the ex_firinterpolation_ref1, the FIR Interpolation block interpolates a single-channel input with a frame period of 1 second (**Sample time** = 1/64 and **Samples per frame** = 64). Because the block is doing multirate frame-based processing and the **Interpolation factor** parameter is set to 4, the output of the FIR Interpolation block has a frame period of 0.25 seconds. As shown in the following figure, the input and output of the FIR Interpolation block have the same frame size, but the sample rate of the output is 1/4 times that of the input.



Example 3

The ex_polyphaseinterp model illustrates the underlying polyphase implementations of the FIR Interpolation block. Run the model, and view the results on the scope. The output of the FIR Interpolation block matches the output of the Polyphase Interpolation Filter block.

Example 4

The ex_mrf_nlp model illustrates the use of the FIR Interpolation block in a number of multistage multirate filters.

Dialog Box

Coefficient Source

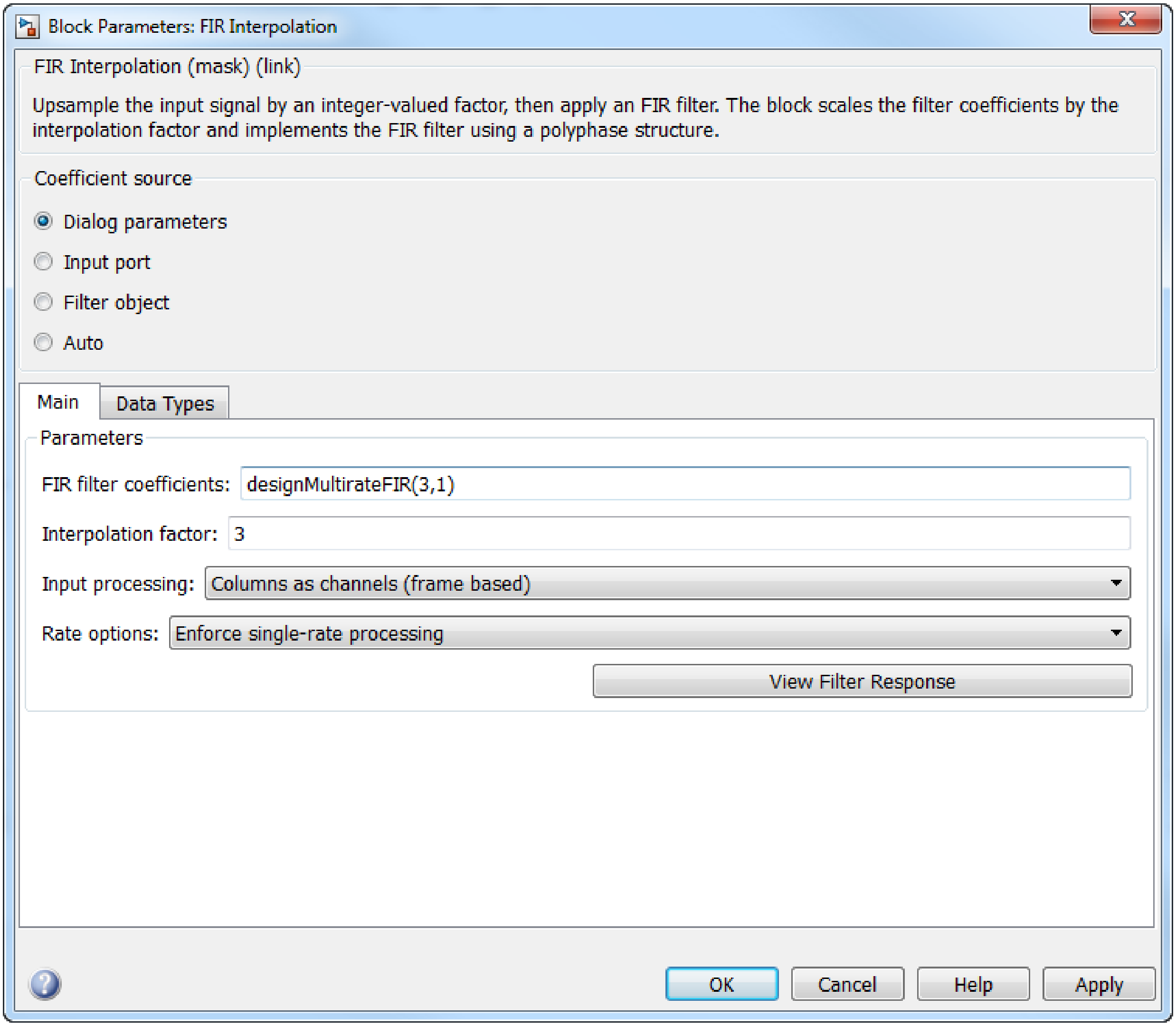
The FIR Interpolation block can operate in four different modes. Select the mode in the **Coefficient source** group box.

- **Dialog parameters** — Enter information about the filter, such as coefficients, in the block mask.
- **Input port** — Specify the filter coefficients with a **Num** input port. The **Num** input port appears when you select the **Input port** option. Coefficient values obtained through **Num** are tunable (can change during simulation), while their properties must remain constant.
- **Filter object** — Specify the filter using a dsp.FIRInterpolator System object.
- **Auto** (default) — Choose the coefficients of an FIR Nyquist filter, predesigned for the Interpolation factor specified in the block dialog box.

Different items appear on the FIR Interpolation block dialog box depending on whether you select **Dialog parameters**, **Input port**, **Filter object**, or **Auto** in the **Coefficient source** group box.

Specify Filter Characteristics in dialog box

The **Main** pane of the FIR Interpolation block dialog box appears as follows when you select **Dialog parameters** in the **Coefficient source** group box.



FIR filter coefficients

Specify the FIR filter coefficients, in descending powers of z. By default, `designMultirateFIR(3,1)` computes the filter coefficients.

Interpolation factor

Specify the integer factor, *L*, by which to increase the sample rate of the input sequence. The default is `3`.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

Specify the method by which the block should interpolate the input. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate, and interpolates the signal by increasing the output frame size by a factor of *L*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block interpolates the signal such that the output sample rate is *L* times faster than the input sample rate.

Output buffer initial conditions

In cases of nonzero latency, the block divides this parameter by the **Interpolation factor** and outputs the results at the output port until the first filtered input sample is available. The default initial condition value is `0`, but you can enter a matrix containing one value for each channel, or a scalar to be applied to all signal channels. This parameter appears only when you configure the block to perform multirate processing.

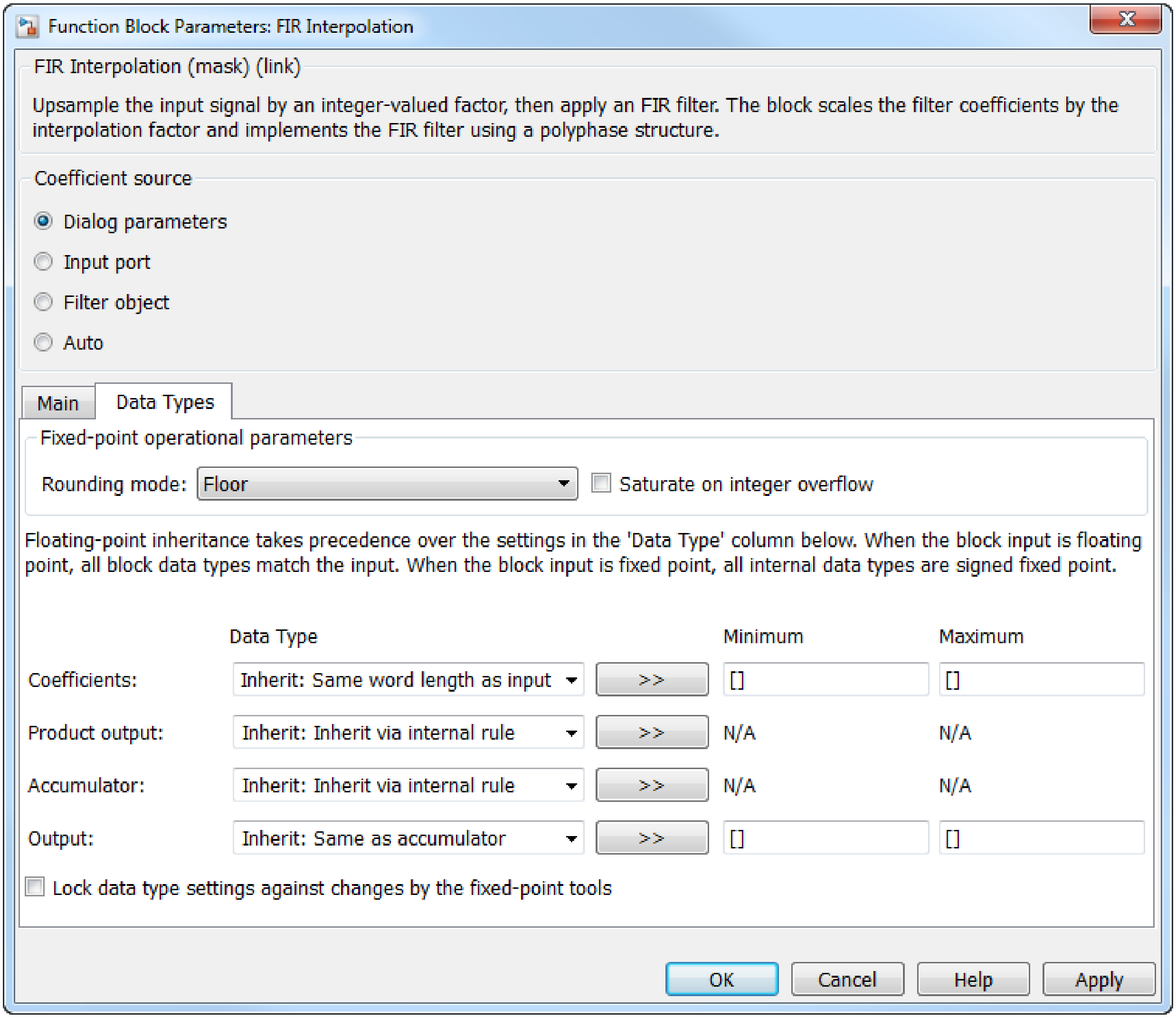
Output buffer initial conditions are stored in the output data type and scaling.

See “Latency” on page 2-722 for more information about latency in the FIR Interpolation block.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the filter defined in the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

The **Data Types** pane of the FIR Interpolation block dialog box appears as follows when you select **Dialog parameters** in the **Coefficient source** group box.



Rounding mode

Select the rounding mode for fixed-point operations. The default is `Floor`. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Note: The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output data type** is `Inherit: Inherit via internal rule`
- **Accumulator data type** is `Inherit: Inherit via internal rule`
- **Output data type** is `Inherit: Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients data type

Specify the coefficients data type. See “Fixed-Point Data Types” on page 2-723 and “Multiplication Data Types” for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Coefficients data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Coefficients Minimum

Specify the minimum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Coefficients Maximum

Specify the maximum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-723 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-723 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-723 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

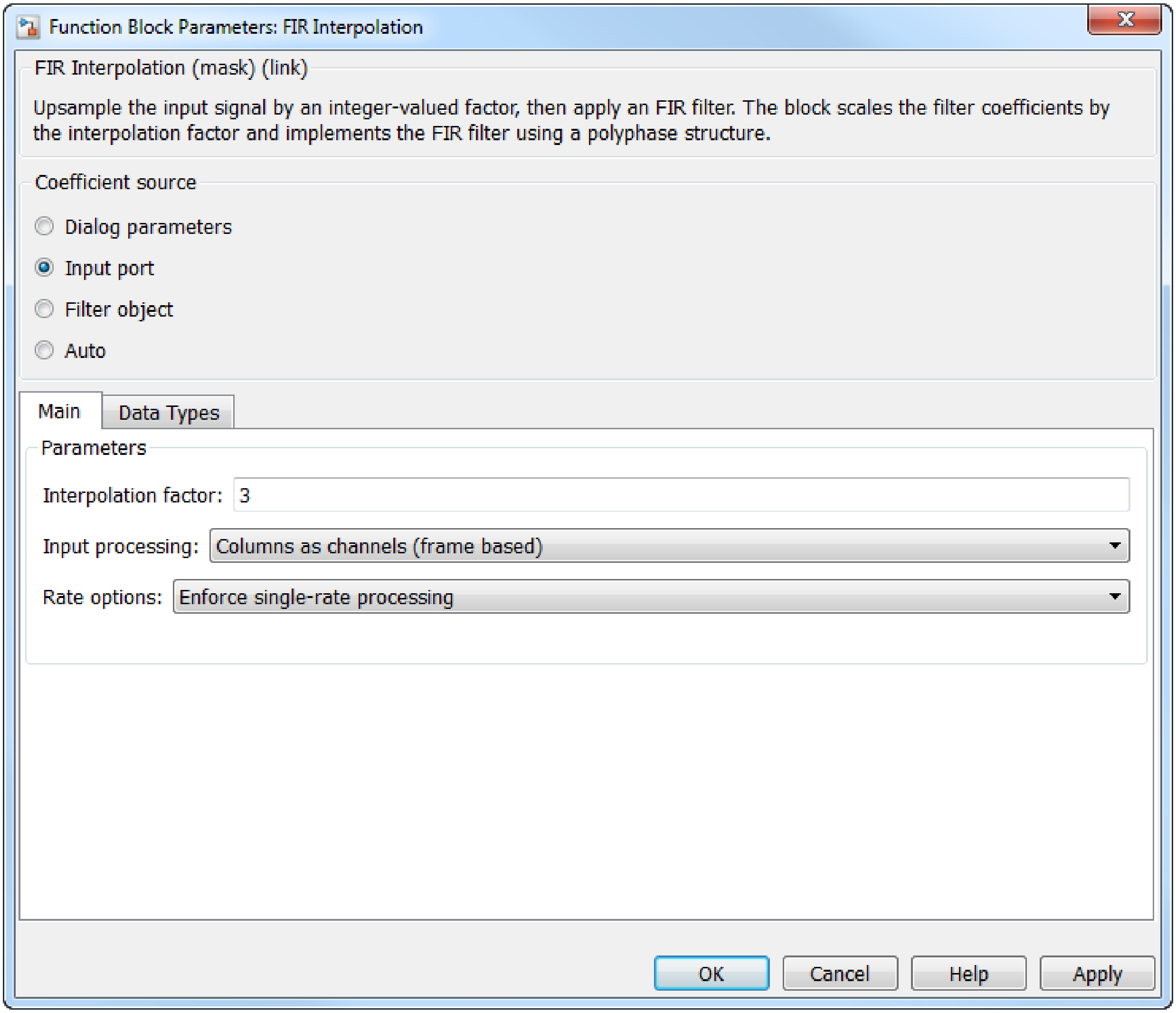
- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Provide Filter Coefficients Through Input Port

The **Main** pane of the FIR Interpolation block dialog box appears as follows when you select **Input port** in the **Coefficient source** group box.



Interpolation factor

Specify the integer factor, L, by which to increase the sample rate of the input sequence. The default is 3.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

Specify the method by which the block should interpolate the input. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate, and interpolates the signal by increasing the output frame size by a factor of *L*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block interpolates the signal such that the output sample rate is *L* times faster than the input sample rate.

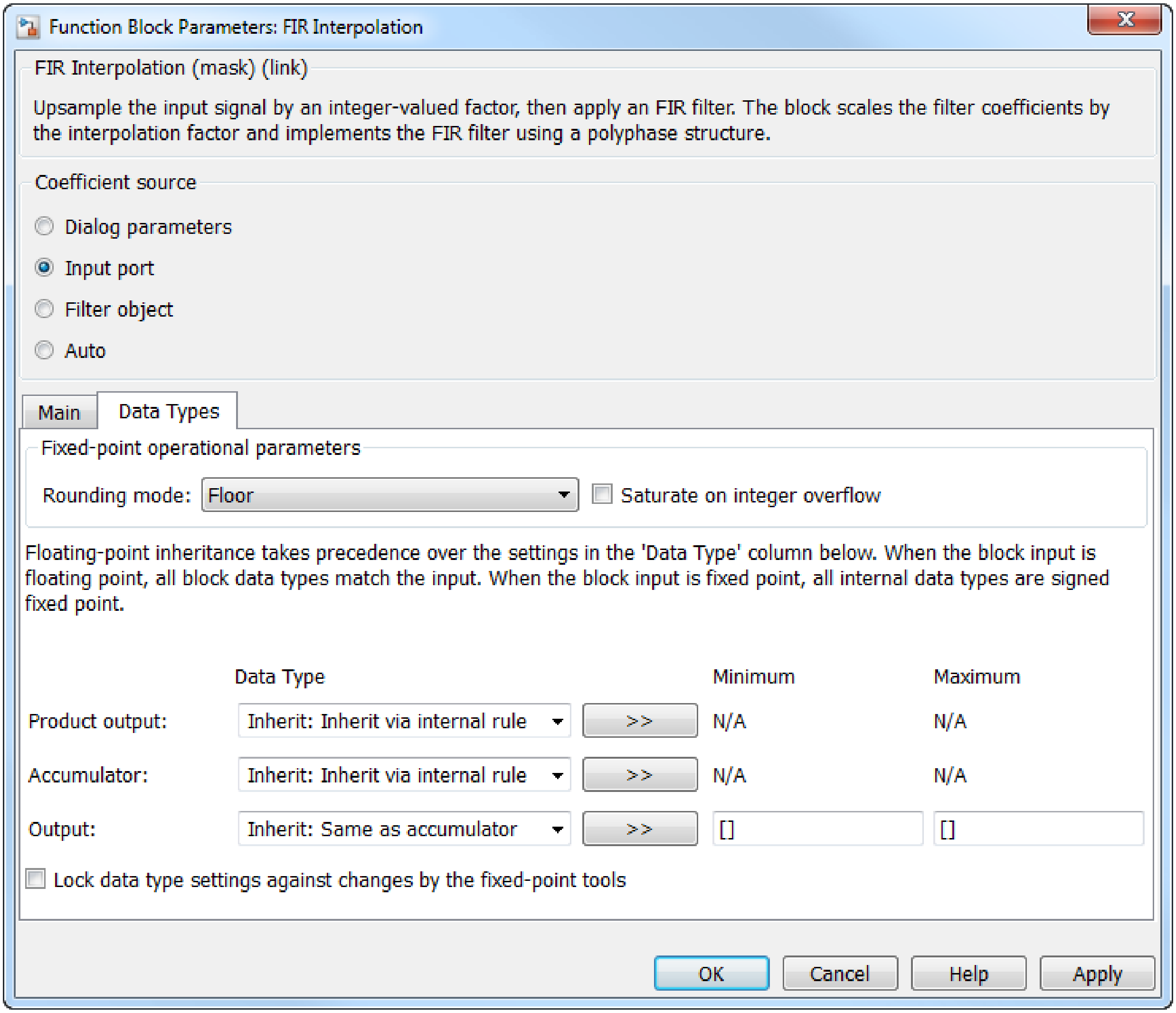
Output buffer initial conditions

In cases of nonzero latency, the block divides this parameter by the **Interpolation factor** and outputs the results at the output port until the first filtered input sample is available. The default initial condition value is `0`, but you can enter a matrix containing one value for each channel, or a scalar to be applied to all signal channels. This parameter appears only when you configure the block to perform multirate processing.

Output buffer initial conditions are stored in the output data type and scaling.

See "Latency" on page 2-722 for more information about latency in the FIR Interpolation block.

The **Data Types** pane of the FIR Interpolation block dialog box appears as follows when you select **Input port** in the **Coefficient source** group box.



Rounding mode

Select the rounding mode for fixed-point operations. The default is `Floor`. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Note: The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output data type** is `Inherit: Inherit via internal rule`
- **Accumulator data type** is `Inherit: Inherit via internal rule`
- **Output data type** is `Inherit: Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-723 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see "Inherit via Internal Rule".
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-723 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see "Inherit via Internal Rule".

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-723 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

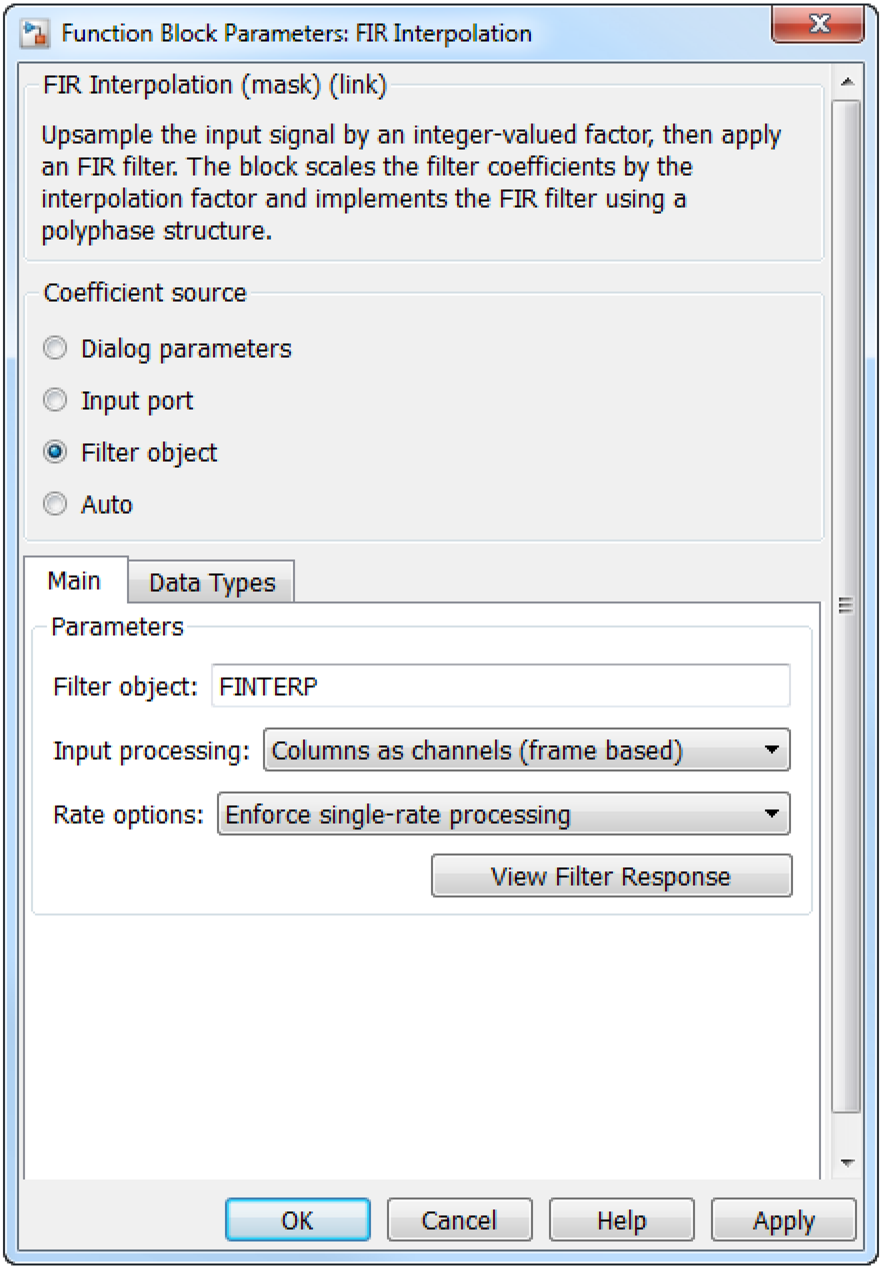
- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Specify Multirate Filter Object

The **Main** pane of the FIR Interpolation block dialog box appears as follows when you select **Filter object** in the **Coefficient source** group box.


Function Block Parameters: FIR Interpolation
FIR Interpolation (mask) (link)
Upsample the input signal by an integer-valued factor, then apply an FIR filter. The block scales the filter coefficients by the interpolation factor and implements the FIR filter using a polyphase structure.
Coefficient source
Dialog parameters
Input port
Filter object
Auto
Main
Data Types
Parameters
Filter object:
FINTERP
Input processing:
Columns as channels (frame based)
Rate options:
Enforce single-rate processing
View Filter Response
OK
Cancel
Help
Apply

Filter object

Specify the name of the multirate filter object that you want the block to implement. You must specify the filter as a dsp.FIRInterpolator System object.

You can define the System object in the block mask or in a MATLAB workspace variable.

For information on creating System objects, see "Define Basic System Objects" (MATLAB).

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

Specify the method by which the block should interpolate the input. You can select one of the following options:

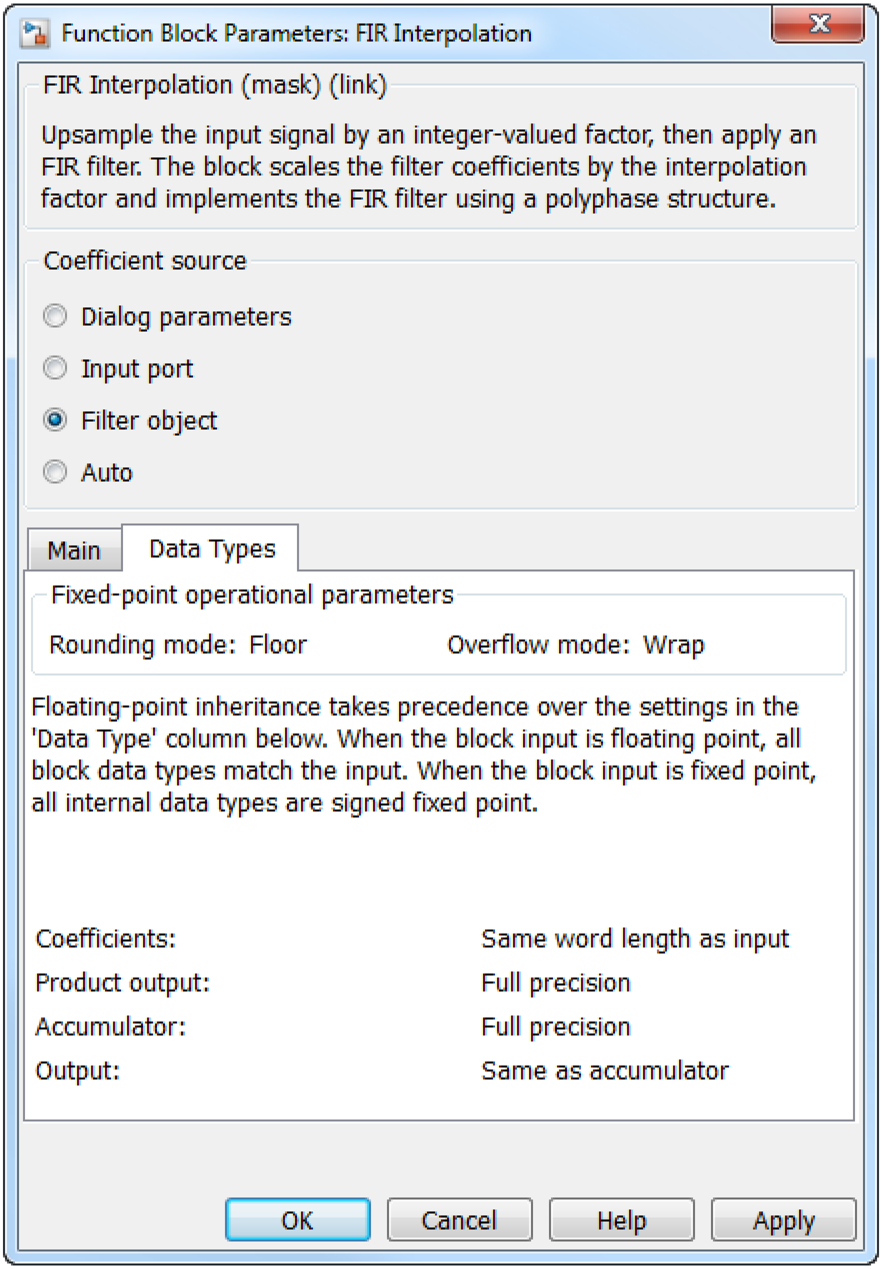
- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate, and interpolates the signal by increasing the output frame size by a factor of *L*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block interpolates the signal such that the output sample rate is *L* times faster than the input sample rate.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the System object specified in the **Filter object** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note: If you specify a filter in the **Filter object** parameter, you must apply the filter by clicking the **Apply** button before using the **View filter response** button.

The **Data Types** pane of the FIR Interpolation block dialog box appears as follows when you select **Filter object** in the **Coefficient source** group box.


Function Block Parameters: FIR Interpolation
FIR Interpolation (mask) (link)
Upsample the input signal by an integer-valued factor, then apply an FIR filter. The block scales the filter coefficients by the interpolation factor and implements the FIR filter using a polyphase structure.
Coefficient source
Dialog parameters
Input port
Filter object
Auto
Main
Data Types
Fixed-point operational parameters
Rounding mode: Floor
Overflow mode: Wrap
Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input. When the block input is fixed point, all internal data types are signed fixed point.
Coefficients:
Same word length as input
Product output:
Full precision
Accumulator:
Full precision
Output:
Same as accumulator
OK
Cancel
Help
Apply

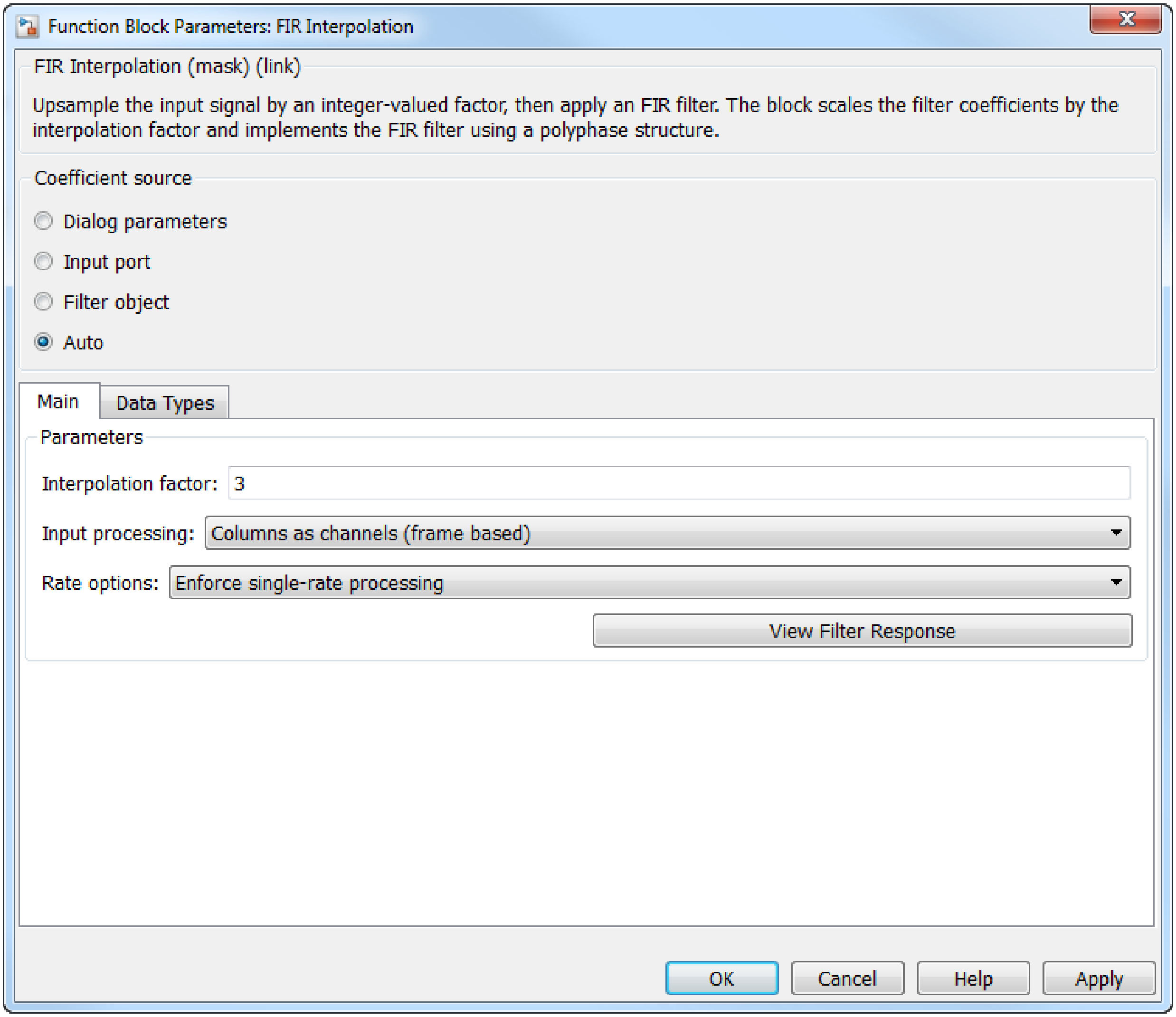
The fixed-point settings of the filter object specified on the **Main** pane are displayed on the **Data Types** pane. You cannot change these settings directly on the block mask. To change the fixed-point settings, edit the filter object directly.

For more information on System objects, see “Define Basic System Objects” (MATLAB).

Choose Filter Coefficients Automatically

When you select **Auto** in the **Coefficient source** group box, the block chooses the filter coefficients automatically. For more information on the filter design algorithm the block uses, see “Specifying the Filter Coefficients” on page 2-720.

The **Main** pane of the FIR Interpolation block dialog box appears as follows when you select **Auto** in the **Coefficient source** group box.



Interpolation factor

Specify the integer factor, L, by which to increase the sample rate of the input sequence. The default is 3.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

Specify the method by which the block should interpolate the input. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate, and interpolates the signal by increasing the output frame size by a factor of L. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block interpolates the signal such that the output sample rate is L times faster than the input sample rate.

Output buffer initial conditions

In cases of nonzero latency, the block divides this parameter by the **Interpolation factor** and outputs the results at the output port until the first filtered input sample is available. The default initial condition value is `0`, but you can enter a matrix containing one value for each channel, or a scalar to be applied to all signal channels. This parameter appears only when you configure the block to perform multirate processing.

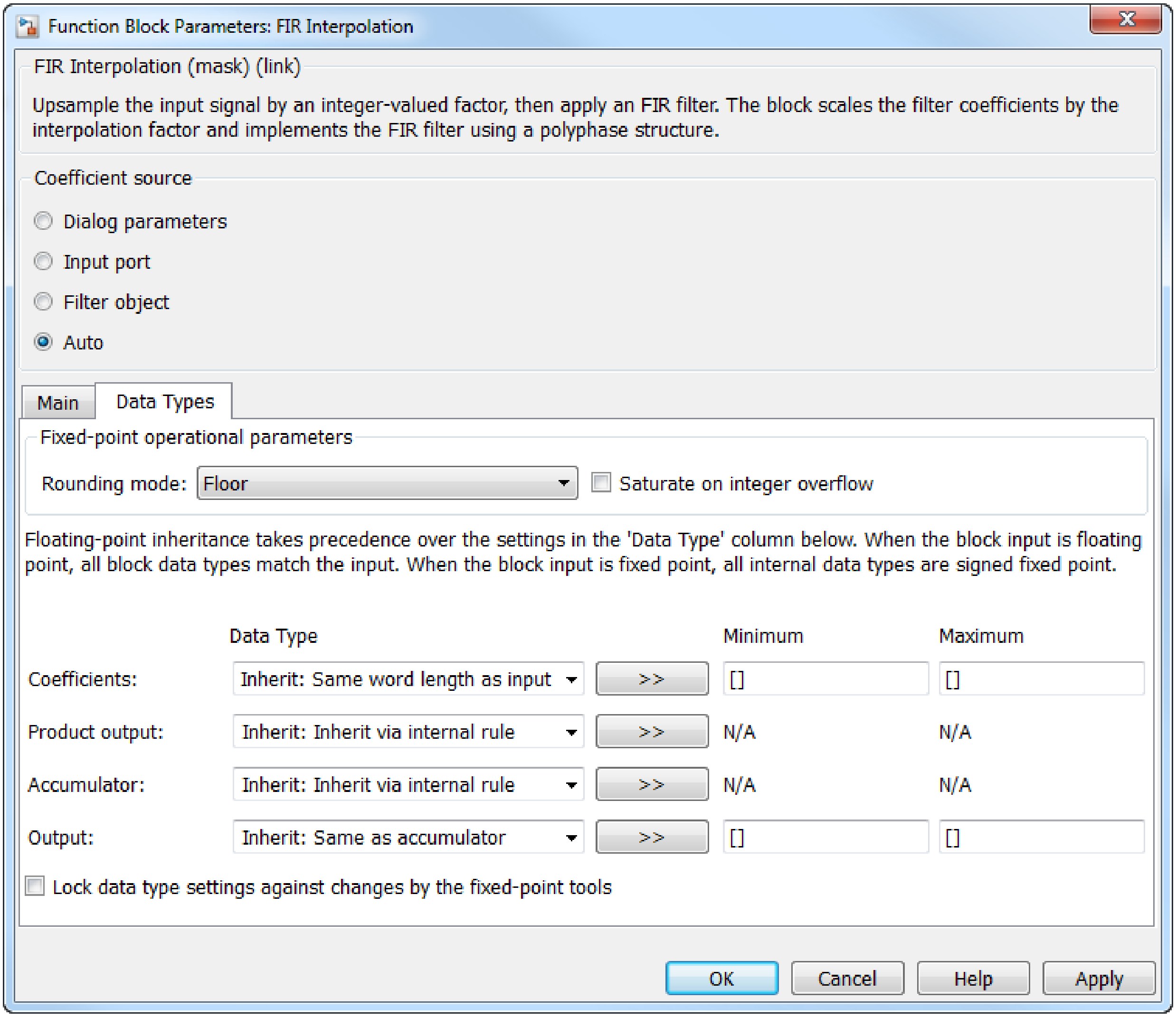
Output buffer initial conditions are stored in the output data type and scaling.

See “Latency” on page 2-722 for more information about latency in the FIR Interpolation block.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the filter defined in the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

The **Data Types** pane of the FIR Interpolation block dialog box appears as follows when you select **Auto** in the **Coefficient source** group box.



Rounding mode

Select the rounding mode for fixed-point operations. The default is `Floor`. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Note: The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output data type** is `Inherit: Inherit via internal rule`
- **Accumulator data type** is `Inherit: Inherit via internal rule`
- **Output data type** is `Inherit: Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients data type

Specify the coefficients data type. See "Fixed-Point Data Types" on page 2-723 and "Multiplication Data Types" for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Coefficients data type** parameter.

See "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink) for more information.

Coefficients Minimum

Specify the minimum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Coefficients Maximum

Specify the maximum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-723 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see "Inherit via Internal Rule".
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-723 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see "Inherit via Internal Rule".
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-723 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see FIR Interpolation.

References

[1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

[2] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

`FIR Decimation`	DSP System Toolbox
`FIR Rate Conversion`	DSP System Toolbox
`FIR Halfband Interpolator`	DSP System Toolbox
`FIR Halfband Decimator`	DSP System Toolbox
`IIR Halfband Interpolator`	DSP System Toolbox
`IIR Halfband Decimator`	DSP System Toolbox
`CIC Compensation Interpolator`	DSP System Toolbox
`CIC Compensation Decimator`	DSP System Toolbox
`Upsample`	DSP System Toolbox
`CIC Interpolation`	DSP System Toolbox
`Digital Down-Converter`	DSP System Toolbox

`Digital Up-Converter`	DSP System Toolbox
dsp.FIRInterpolator	DSP System Toolbox
dsp.CICCompensationInterpolator	DSP System Toolbox
dsp.FIRHalfbandInterpolator	DSP System Toolbox
`firnyquist`	DSP System Toolbox
`firhalfband`	DSP System Toolbox
`firgr`	DSP System Toolbox
`firceqrip`	DSP System Toolbox

Introduced before R2006a

FIR Halfband Decimator

Decimate signal using polyphase FIR halfband filter

Library

Filtering/Filter Designs

`dspfdesign`

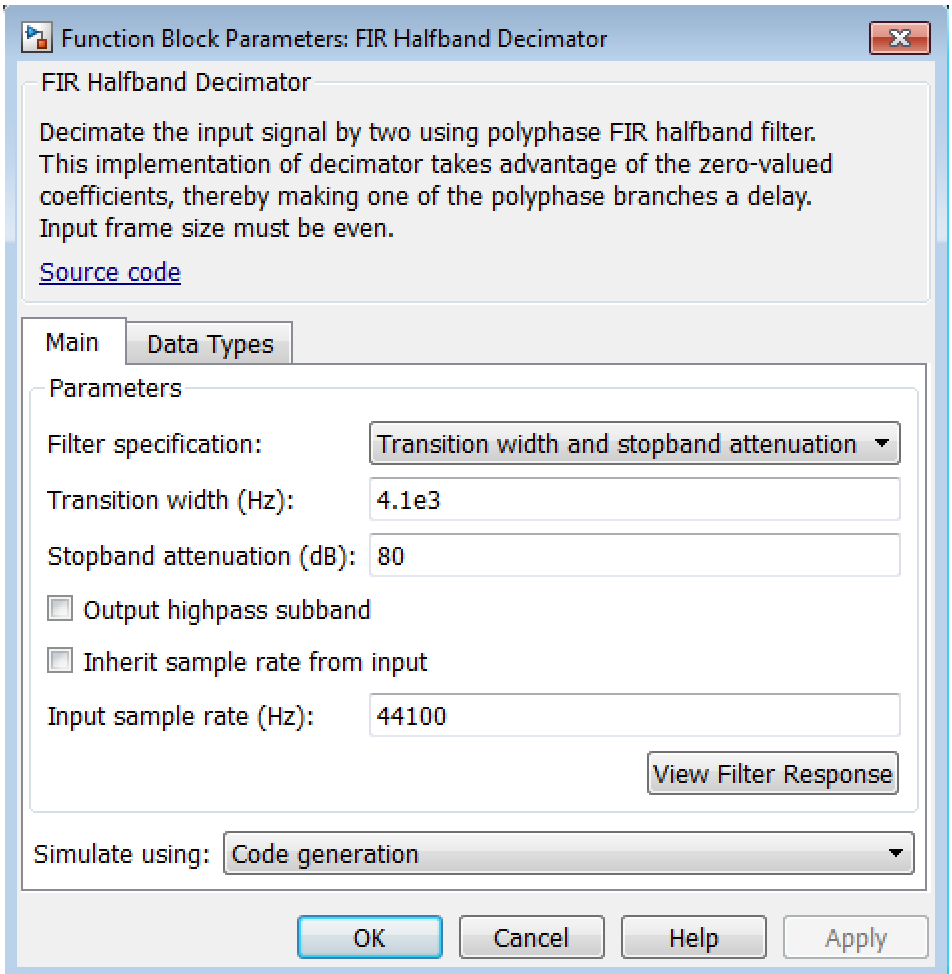
Description

The FIR Halfband Decimator block performs polyphase decimation of the input signal by a factor of two. The block uses an FIR equiripple design to construct the halfband filters. The implementation takes advantage of the zero-valued coefficients of the FIR halfband filter, making one of the polyphase branches a delay. You can also use the block to implement the analysis portion of a two-band filter bank to separate a signal into lowpass and highpass subbands.

The input signal can be a real- or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The number of rows in the input signal must be a multiple of 2. The block supports fixed-point operations and ARM® Cortex® code generation. For more information on ARM Cortex code generation, refer “ARM Cortex-M and ARM Cortex-A Optimization”.

Dialog Box

Main Tab



Filter specification

Parameters used to design the FIR halfband filter. Because the filter design has only two degrees of freedom, you can specify only two of three parameters:

- `Transition width and stopband attenuation` (default) — Design the filter using **Transition width (Hz)** and **Stopband attenuation (dB)**. This design is the minimum order design.
- `Filter order and transition width` — Design the filter using **Filter order** and **Transition width (Hz)**.
- `Filter order and stopband attenuation` — Design the filter using **Filter order** and **Stopband attenuation (dB)**.

Transition width (Hz)

Transition width, specified as a real positive scalar in Hz. The transition width must be less than half the input sample rate. You can specify the transition width when **Filter specification** is set to `Filter order and transition width` or `Transition width and stopband attenuation`. The default is `4.1e3`.

Filter order

Filter order, specified as an even positive integer. You can specify the filter order when **Filter specification** is set to `Filter order and transition width` or `Filter order and stopband attenuation`. The default is `52`.

Stopband attenuation (dB)

Stopband attenuation, specified as a real positive scalar in dB. You can specify the stopband attenuation when **Filter specification** is set to `Filter order and stopband attenuation` or `Transition width and stopband attenuation`. The default is `80`.

Output highpass subband

When you select this check box, the block acts as a synthesis filter bank and synthesizes a signal from the highpass and lowpass subbands. When you clear this check box, the block acts as an FIR halfband decimator and accepts a single vector– or matrix–valued input.

Inherit sample rate from input

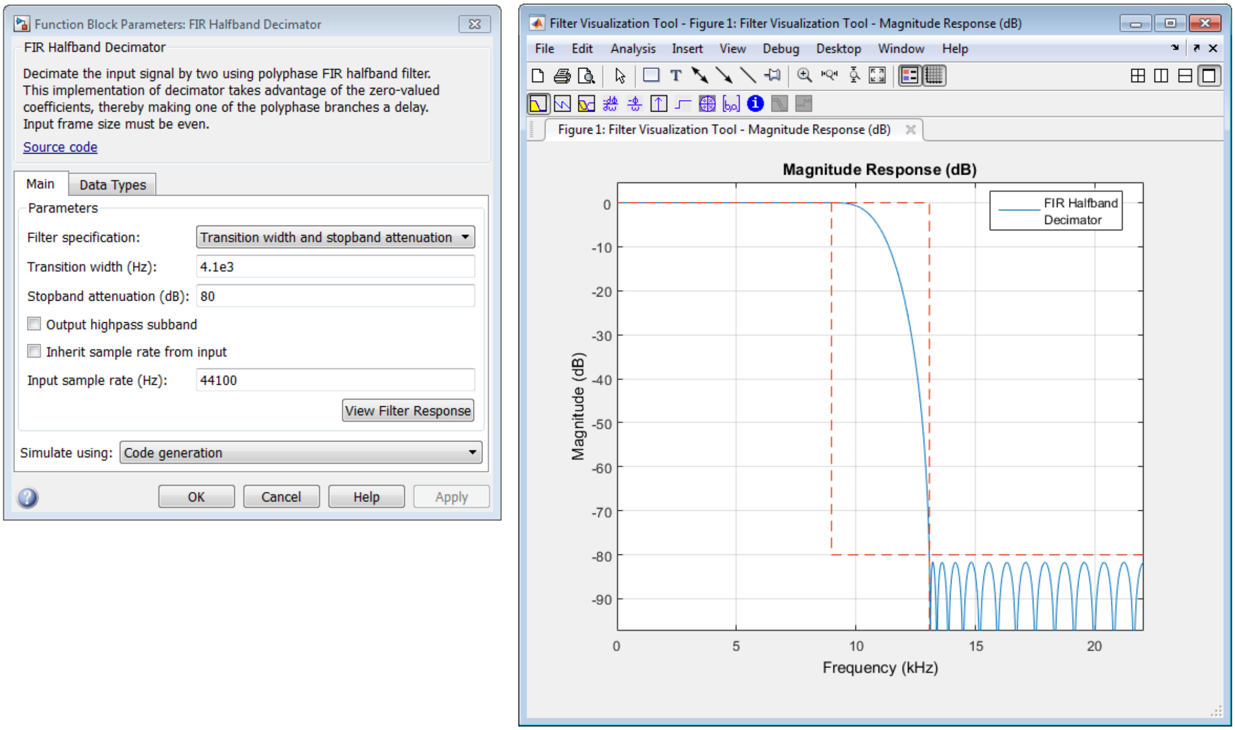
When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**.

Input sample rate (Hz)

Input sample rate, specified as a scalar in Hz. The default is `44100`.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the FIR Halfband Decimator. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

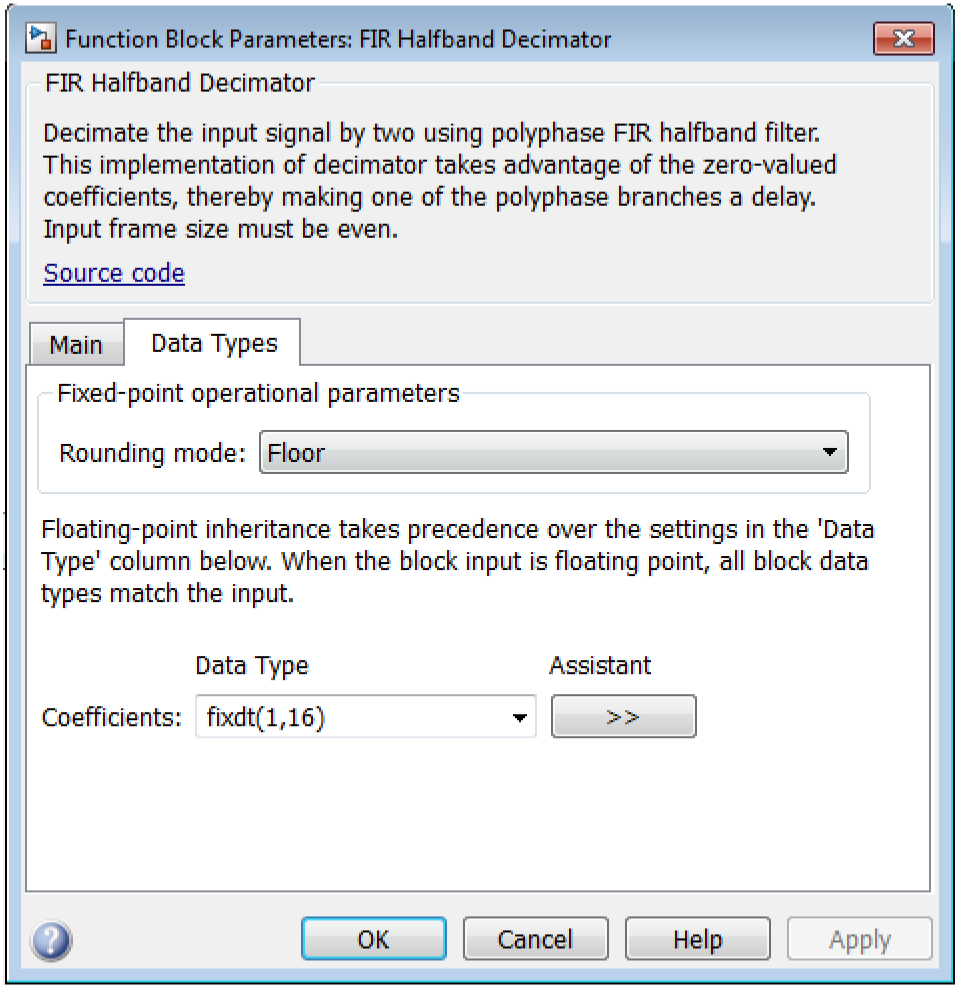
- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Data Types Tab



Rounding mode

Rounding method for the output fixed-point operations. The rounding methods are `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, and `Zero`. The default is `Floor`.

Coefficients data type

Fixed-point data type of the coefficients, specified as one of the following:

- `fixdt(1,16)` (default) — Signed fixed-point data type of word length `16`, with binary point scaling. The block determines the fraction length automatically from the coefficient values in such a way that the coefficients occupy maximum representable range without overflowing.
- `fixdt(1,16,0)` — Signed fixed-point data type of word length `16` and fraction length, `0`. You can change the fraction length to any other integer value.
- <data type expression> — Specify the data type using an expression that evaluates to a data type object, for example, numeric type (`fixdt`,`[ ]`,`16`, `15`). Specify the sign mode of this data type as `[ ]` or `true`.
- `Refresh Data Type` — Refreshes to the default data type.

Click the **Show data type assistant** button >> to display the data type assistant, which helps you set the stage input parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed or unsigned) • 8-, 16-, 32-, and 64-bit signed integers • real and complex data
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only)

Port	Supported Data Types
	• 8-, 16-, 32-, and 64-bit signed integers • real and complex data

See Also

dsp.FIRHalfbandInterpolator	DSP System Toolbox
dsp.FIRHalfbandDecimator	DSP System Toolbox
`FIR Halfband Interpolator`	DSP System Toolbox

Algorithm

This block brings the capabilities of the dsp.FIRHalfbandDecimator System object to the Simulink environment.

For information on the algorithms used by this block, see the Algorithms on page 4-781 section of `dsp.FIRHalfbandDecimator`.

Introduced in R2015b

FIR Halfband Interpolator

Interpolate signal using polyphase FIR half band filter

Library

Filtering / Filter Designs

`dspfdesign`

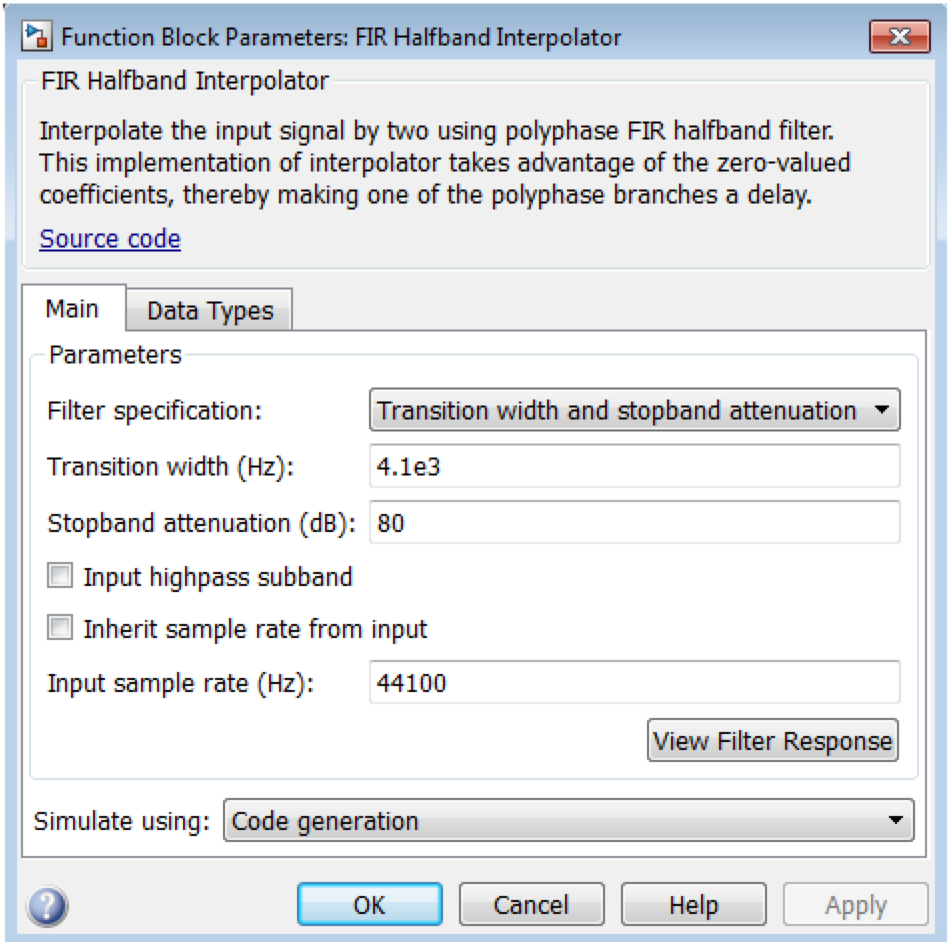
Description

The FIR Halfband Interpolator block performs interpolation of the input signal by a factor of two. The block uses an FIR equiripple design to construct the halfband filters. To filter the input, the block uses an efficient polyphase implementation. The implementation takes advantage of the zero-valued coefficients of the FIR halfband filter, making one of the polyphase branches a delay. You can also use the block to implement the synthesis portion of a two-band filter bank to synthesize a signal from lowpass and highpass subbands.

The input signal can be a real- or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The block supports fixed-point operations and ARM Cortex code generation. For more information on ARM Cortex code generation, refer “ARM Cortex-M and ARM Cortex-A Optimization”.

Dialog Box

Main Tab



Filter specification

Parameters used to design FIR halfband filter. The filter design has only two degrees of freedom, so you can specify only two of the three parameters:

- `Transition width and stopband attenuation` (default) — Design the filter using **Transition width (Hz)** and **Stopband attenuation (dB)**. This design is the minimum order design.
- `Filter order and transition width` — Design the filter using **Filter order** and **Transition width (Hz)**.
- `Filter order and stopband attenuation` — Design the filter using **Filter order** and **Stopband attenuation (dB)**.

Transition width (Hz)

Transition width, specified as a real positive scalar in Hz. The transition width must be less than half the input sample rate. You can specify the transition width only when **Filter specification** is set to `Filter order and transition width` or `Transition width and stopband attenuation`. The default is `4.1e3`.

Filter order

Filter order, specified as an even positive integer. You can specify only when **Filter specification** is set to `Filter order and transition width` or `Filter order and stopband attenuation`. The default is `52`.

Stopband attenuation (dB)

Stopband attenuation, specified as a real positive scalar in dB. You can specify the stopband attenuation only when **Filter specification** is set to `Filter order and stopband attenuation` or `Transition width and stopband attenuation`. The default is `80`.

Input highpass subband

When you select this check box, the block acts as a synthesis filter bank and synthesizes a signal from the highpass and lowpass subbands. When you clear this check box, the block acts as an FIR half band interpolator and accepts a single vector– or matrix–valued input.

Inherit sample rate from input

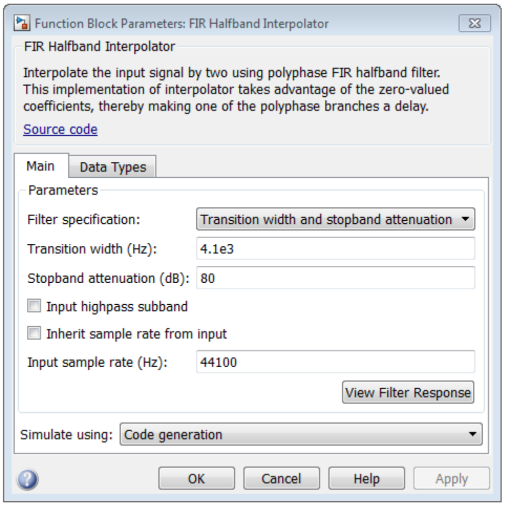
When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you specify the sample rate in **Input Sample Rate (Hz)**.

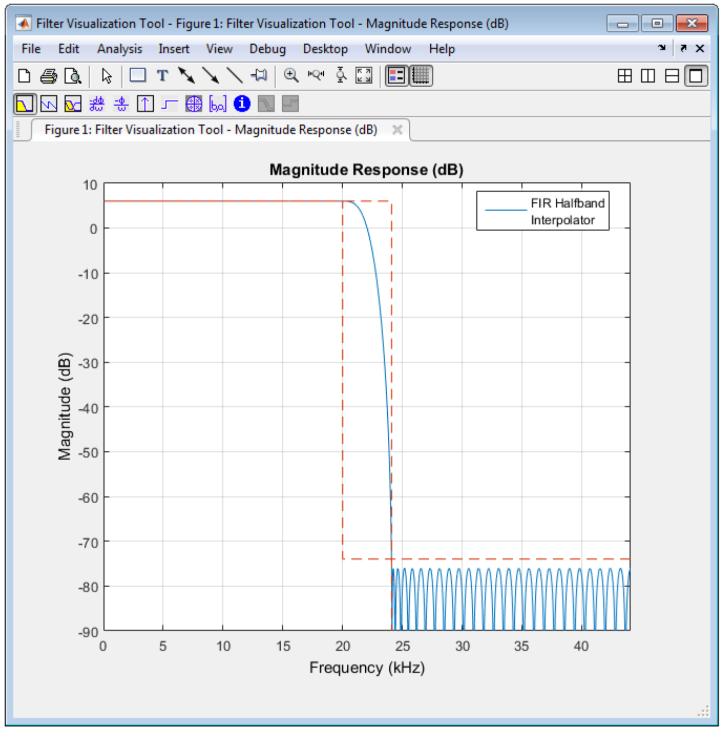
Input sample rate (Hz)

Input sample rate, specified as a scalar in Hz. The default value is `44100`.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the FIR Halfband Interpolator. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.





To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

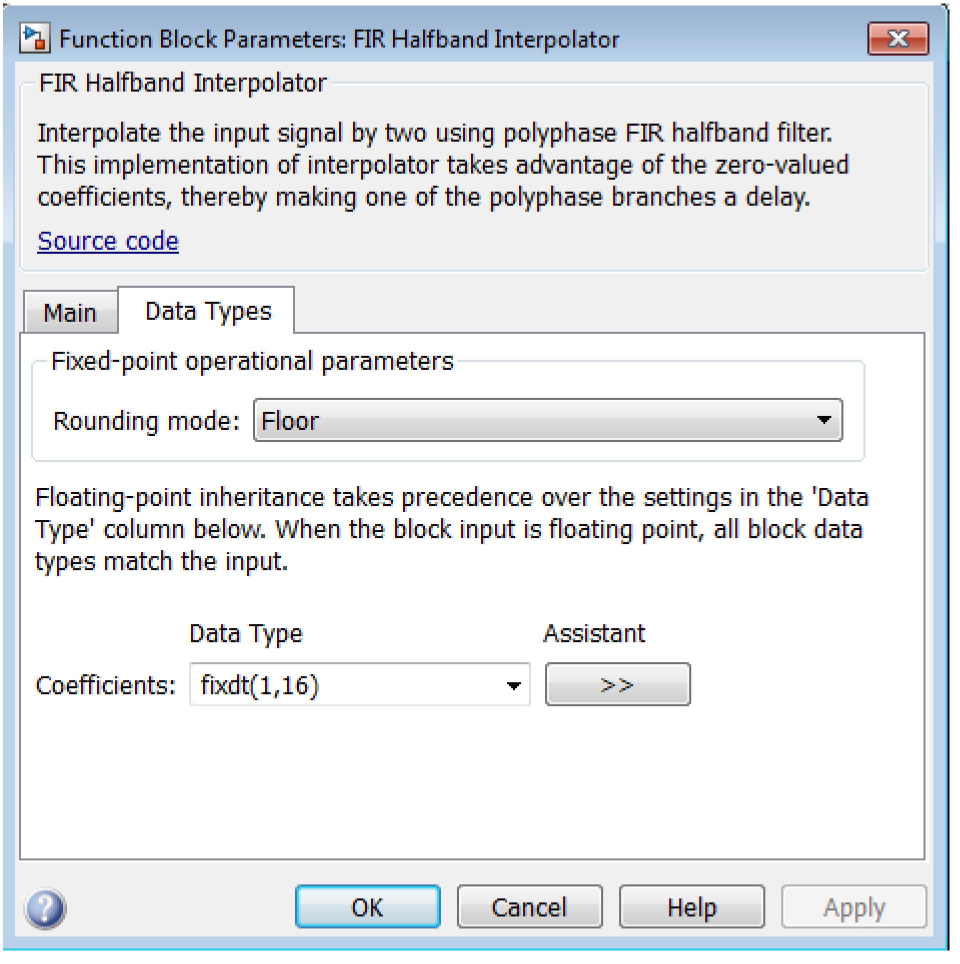
- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Data Types Tab



Rounding mode

Rounding method for the output fixed-point operations. The rounding methods are `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, and `Zero`. The default is `Floor`.

Coefficients data type

Fixed-point data type of the coefficients, specified as one of the following:

- `fixdt(1,16)` (default) — Signed fixed-point data type of word length `16`, with binary point scaling. The block determines the fraction length automatically from the coefficient values in such a way that the coefficients occupy maximum representable range without overflowing.
- `fixdt(1,16,0)` — Signed fixed-point data type of word length `16` and fraction length, `0`. You can change the fraction length to any other integer value.
- `<data type expression>` — Specify the data type using an expression that evaluates to a data type object, for example, numeric type (`[ ]`,`16`, `15`). Specify the sign mode of this data type as `[ ]` or `true`.
- `Refresh Data Type` — Refreshes to the default data type.

Click the **Show data type assistant** button >> to display the data type assistant, which helps you set the stage input parameter.

See "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink) for more information.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed or unsigned) • 8-, 16-, 32-, and 64-bit signed integers • real and complex data
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only)

Port	Supported Data Types
	• 8-, 16-, 32-, and 64-bit signed integers • real and complex data

See Also

dsp.FIRHalfbandInterpolator	DSP System Toolbox
dsp.FIRHalfbandDecimator	DSP System Toolbox
`FIR Halfband Decimator`	DSP System Toolbox

Algorithm

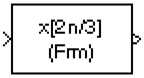
This block brings the capabilities of the dsp.FIRHalfbandInterpolator System object to the Simulink environment.

For information on the algorithms used by this block, see the Algorithms on page 4-815 section of `dsp.FIRHalfbandInterpolator`.

Introduced in R2015b

FIR Rate Conversion

Upsample, filter, and downsample input signals



Library

Filtering / Multirate Filters

`dspmlti4`

Description

The FIR Rate Conversion block resamples the discrete-time input such that its sample period is K/L times the input sample period (T_{si}). K is the integer value you specify for the **Decimation factor** parameter, and L is the integer value you specify for the **Interpolation factor** parameter.

The block treats each column of the input as a separate channel, and resamples the data in each channel independently over time. To do so, the block implements a polyphase filter structure and performs the following operations:

1. Upsamples the input to a higher rate by inserting L-1 zeros between input samples.
2. Passes the upsampled data through a direct-form II transpose FIR filter.
3. Downsamples the filtered data to a lower rate by discarding K-1 consecutive samples following each sample that the block retains.

The polyphase filter implementation is more efficient than a straightforward upsample-filter-decimate algorithm. See Orfanidis [1] for more information.

Specifying the Resampling Rate

You specify the resampling rate of the FIR Rate Conversion block using the **Decimation factor** and **Interpolation factor** parameters. For an M_i-by-N matrix input, the **Decimation factor**, K, and the **Interpolation factor**, L, must satisfy the following requirements:

- K and L must be relatively prime integers; that is, the ratio K/L cannot be reduced to a ratio of smaller integers.
- $\frac{K}{L} = \frac{M_i}{M_o}$, where M_i and M_o are the integer frame sizes of the input and output, respectively.

You can satisfy the second requirement by setting the **Decimation factor**, K, equal to the input frame size, M_i. When you do so, the output frame size, M_o, equals the **Interpolation factor**, L.

By changing the frame size in this way, the block is able to hold the frame period constant ($T_{fi} = T_{fo}$) and achieve the desired conversion of the sample period, such that

$$T_{so} = \frac{K}{L} \times T_{si}$$

where T_{so} is the output sample period.

Specifying the FIR Filter Coefficients

To specify the filter coefficients, select the mode you want the FIR Rate Conversion block to operate in. Select the mode in the **Coefficient source** group box.

- **Dialog parameters** — Enter information about the filter, such as coefficients, in the block dialog box.
- **Filter object** — Specify the filter using a dsp.FIRRateConverter System object.
- **Auto** (default) — Choose the filter coefficients automatically.

When you select the **Dialog parameters** option, you use the **FIR filter coefficients** parameter to specify the numerator coefficients of the FIR filter transfer function $H(z)$.

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

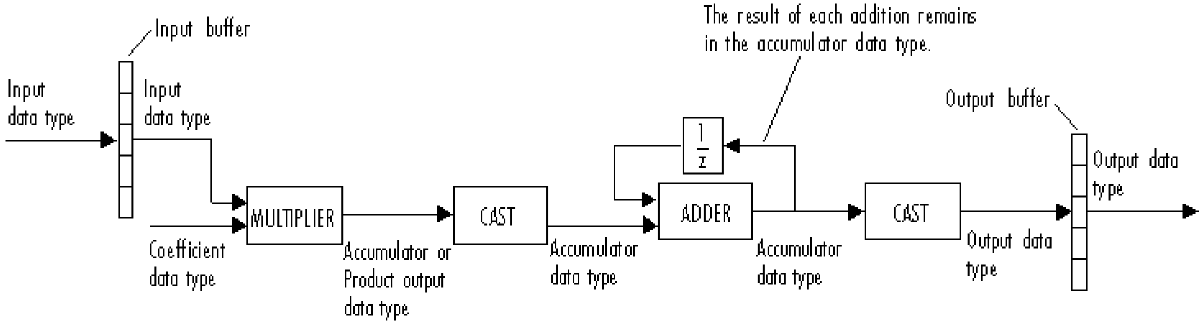
You can generate the FIR filter coefficient vector, `[b(1) b(2) ... b(m)]`, using one of the DSP System Toolbox filter design functions such as `designMultirateFIR`, `firnyquist`, `firhalfband`, `firgr` or `firceqrip`.

The coefficient vector you specify must have a length greater than the interpolation factor ($m>L$). The FIR filter must be a lowpass filter with a normalized cutoff frequency, no greater than `min`(1/L,1/K). The block internally initializes all filter states to zero.

When you select **Auto**, the block designs an FIR multirate filter with the decimation factor specified in **Decimation factor** and interpolation factor specified in **Interpolation factor**. The `designMultirateFIR` function designs the filter and returns the coefficients used by the block. For more information on the filter design, see Orfanidis [1].

Fixed-Point Data Types

The following diagram shows the data types used within the FIR Rate Conversion block for fixed-point signals.



You can set the coefficient, product output, accumulator, and output data types in the block dialog box as discussed in “Dialog Box” on page 2-769. The diagram shows that input data is stored in the input buffer in the same data type and scaling as the input. Filtered data resides in the output buffer in the output data type and scaling that you set in the block dialog. The block stores any initial conditions in the output buffer using the output data type and scaling that you set in the block dialog box.

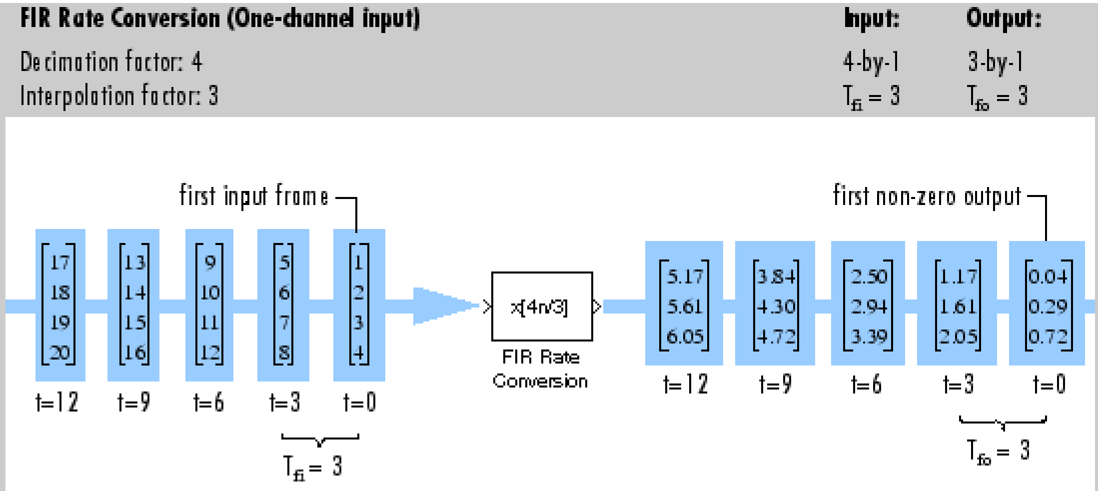
The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

Note: When the block input is fixed point, all internal data types are signed fixed point.

Examples

Example 1

The following figure shows how the FIR Rate Conversion block converts a 4-by-1 input with a sample period of 3/4, to a 3-by-1 output with a sample period of 1. The frame period (T_f) of 3 remains constant.



Example 2

The ex_audio_src model provides a simple illustration of one way to convert a speech signal from one sample rate to another. In this model, the data is first sampled at 22,050 Hz and then resampled at 8000 Hz. If you listen to the output, you can hear that the high frequency content has been removed from the signal, although the speech sounds basically the same.

Dialog Box

Coefficient Source

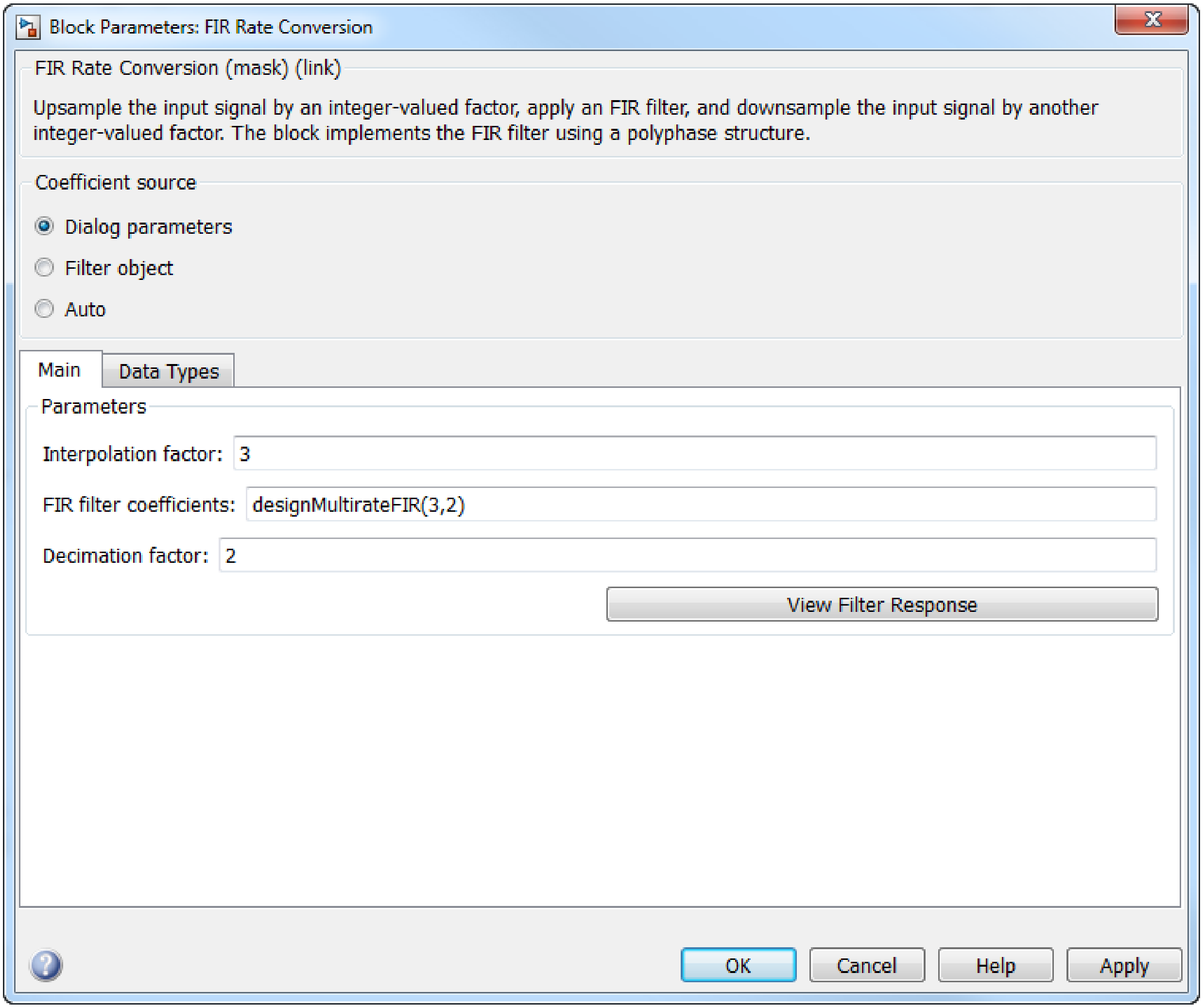
The FIR Rate Conversion block can operate in three different modes. Select the mode in the **Coefficient source** group box.

- **Dialog parameters** — Enter information about the filter, such as coefficients, in the block mask.
- **Filter object** — Specify the filter using a dsp.FIRRateConverter System object.
- **Auto** (default) — Choose the filter coefficients automatically.

Different items appear on the FIR Rate Conversion block dialog box depending on whether you select **Dialog parameters**, **Filter object**, or **Auto** in the **Coefficient source** group box.

Specify Filter Characteristics in Dialog

The **Main** pane of the FIR Rate Conversion block dialog appears as follows when **Dialog parameters** is selected in the **Coefficient source** group box.



Interpolation factor

Specify the integer factor, *L*, by which to upsample the signal before filtering. The default is 3.

FIR filter coefficients

Specify the FIR filter coefficients in descending powers of *z*. By default, `designMultirateFIR(3,2)` computes the filter coefficients.

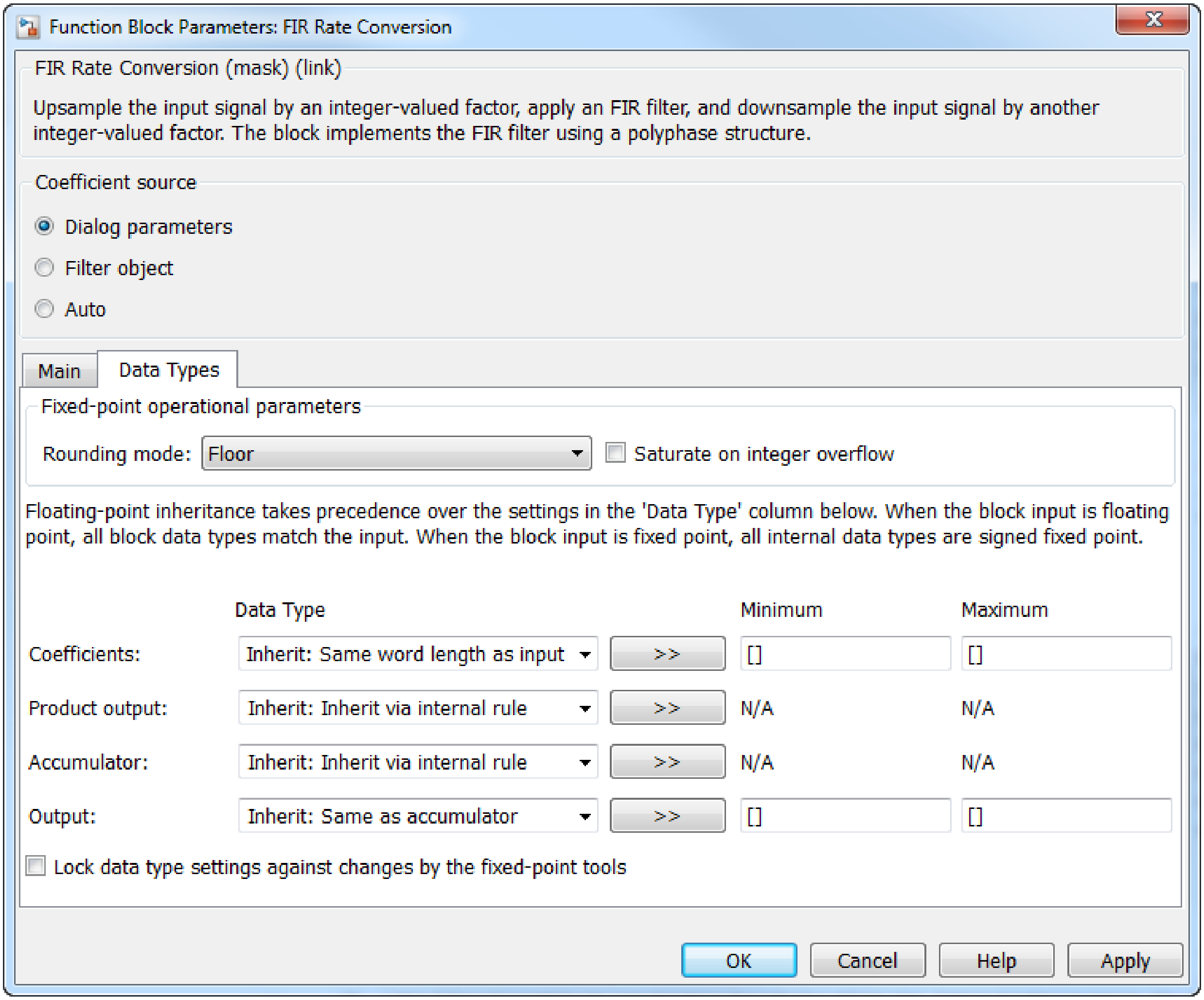
Decimation factor

Specify the integer factor, *K*, by which to downsample the signal after filtering. The default is `2`.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the filter defined in the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

The **Data Types** pane of the FIR Rate Conversion block dialog appears as follows when **Dialog parameters** is specified in the **Coefficient source** group box.



Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Note: The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output data type** is `Inherit: Inherit via internal rule`
- **Accumulator data type** is `Inherit: Inherit via internal rule`
- **Output data type** is `Inherit: Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients data type

Specify the coefficients data type. See “Fixed-Point Data Types” on page 2-768 and “Multiplication Data Types” for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Coefficients data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Coefficients Minimum

Specify the minimum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Coefficients Maximum

Specify the maximum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-768 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-768 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-768 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

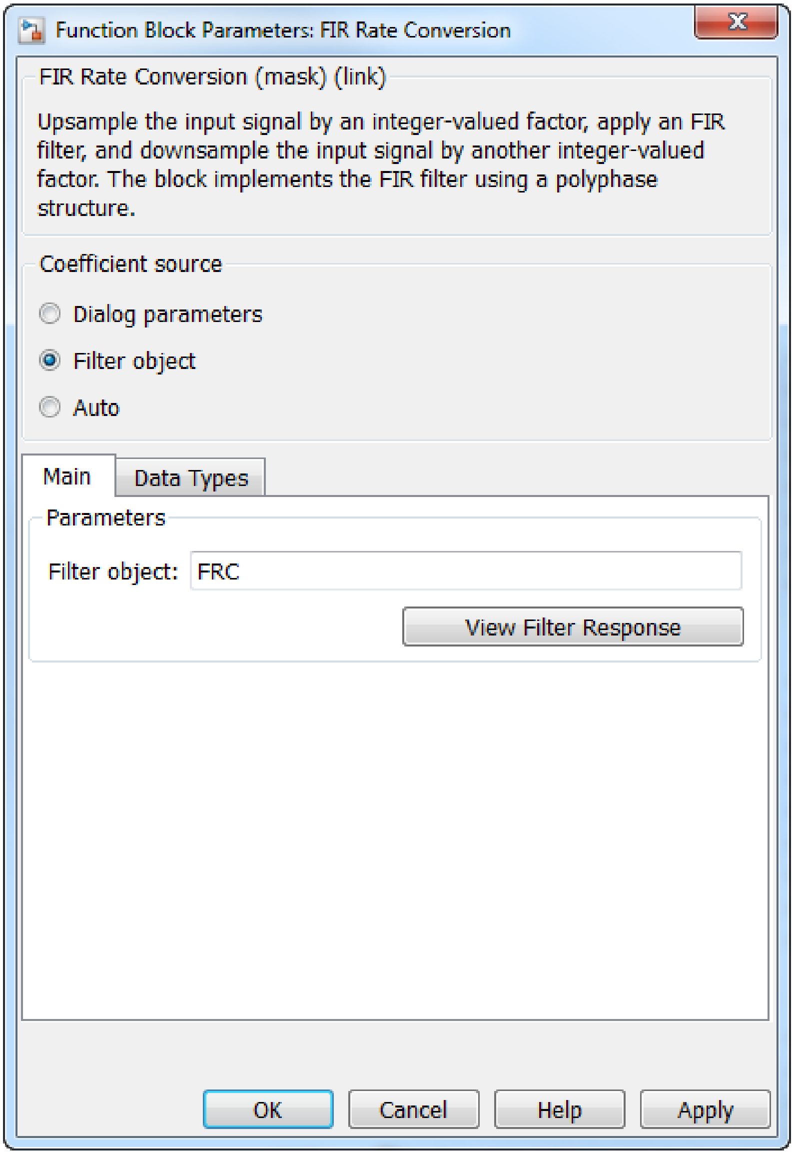
- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Specify Multirate Filter Object

The **Main** pane of the FIR Rate Conversion block dialog appears as follows when **Filter object** is specified in the **Coefficient source** group box.



Filter object

Specify the name of the multirate filter object that you want the block to implement. You must specify the filter as a dsp.FIRRateConverter System object.

You can define the System object in the block mask or in a MATLAB workspace variable.

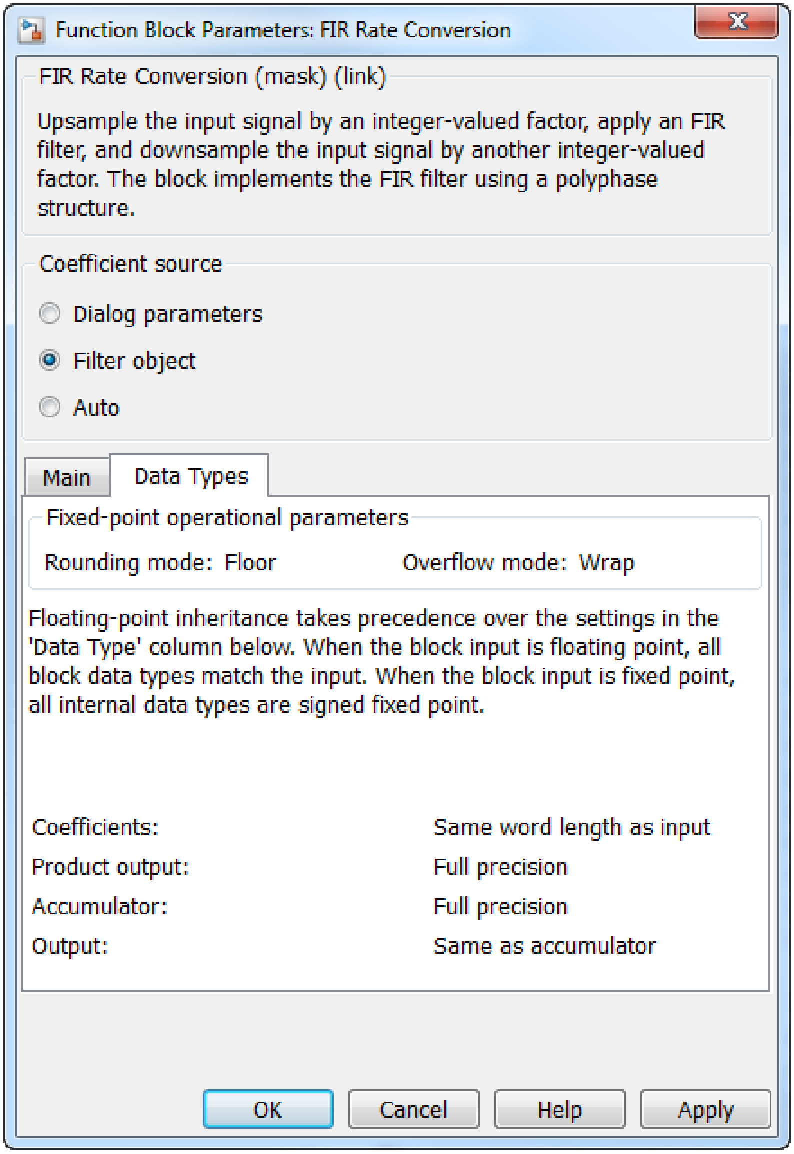
For information on creating System objects, see “Define Basic System Objects” (MATLAB).

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the System object specified in the **Filter object** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note: If you specify a filter in the **Filter object** parameter, you must apply the filter by clicking the **Apply** button before using the **View filter response** button.

The **Data Types** pane of the FIR Rate Conversion block dialog appears as follows when you select **Filter object** in the **Coefficient source** group box.



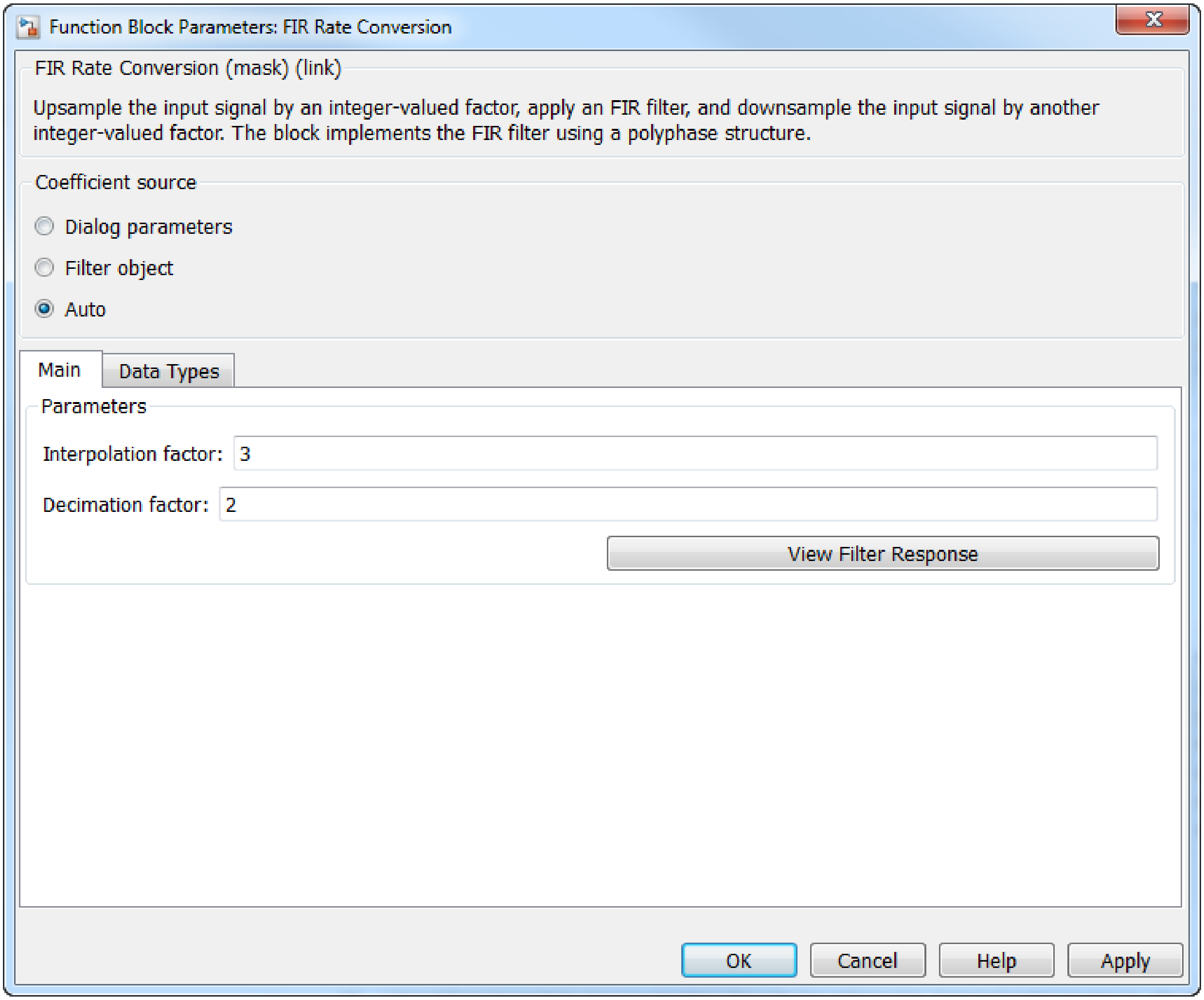
The fixed-point settings of the filter object specified on the **Main** pane are displayed on the **Data Types** pane. You cannot change these settings directly on the block mask. To change the fixed-point settings you must edit the filter object directly.

For more information on System objects, see the “What Are System Objects?” (MATLAB).

Choose Filter Coefficients Automatically

When you select **Auto** in the **Coefficient source** group box, the block chooses the filter coefficients automatically. For more information on the filter design algorithm the block uses, see “Specifying the FIR Filter Coefficients” on page 2-767.

The **Main** pane of the FIR Rate Conversion block dialog box appears as follows when you select **Auto** in the **Coefficient source** group box.



Interpolation factor

Specify the integer factor, L, by which to upsample the signal before filtering. The default is 3.

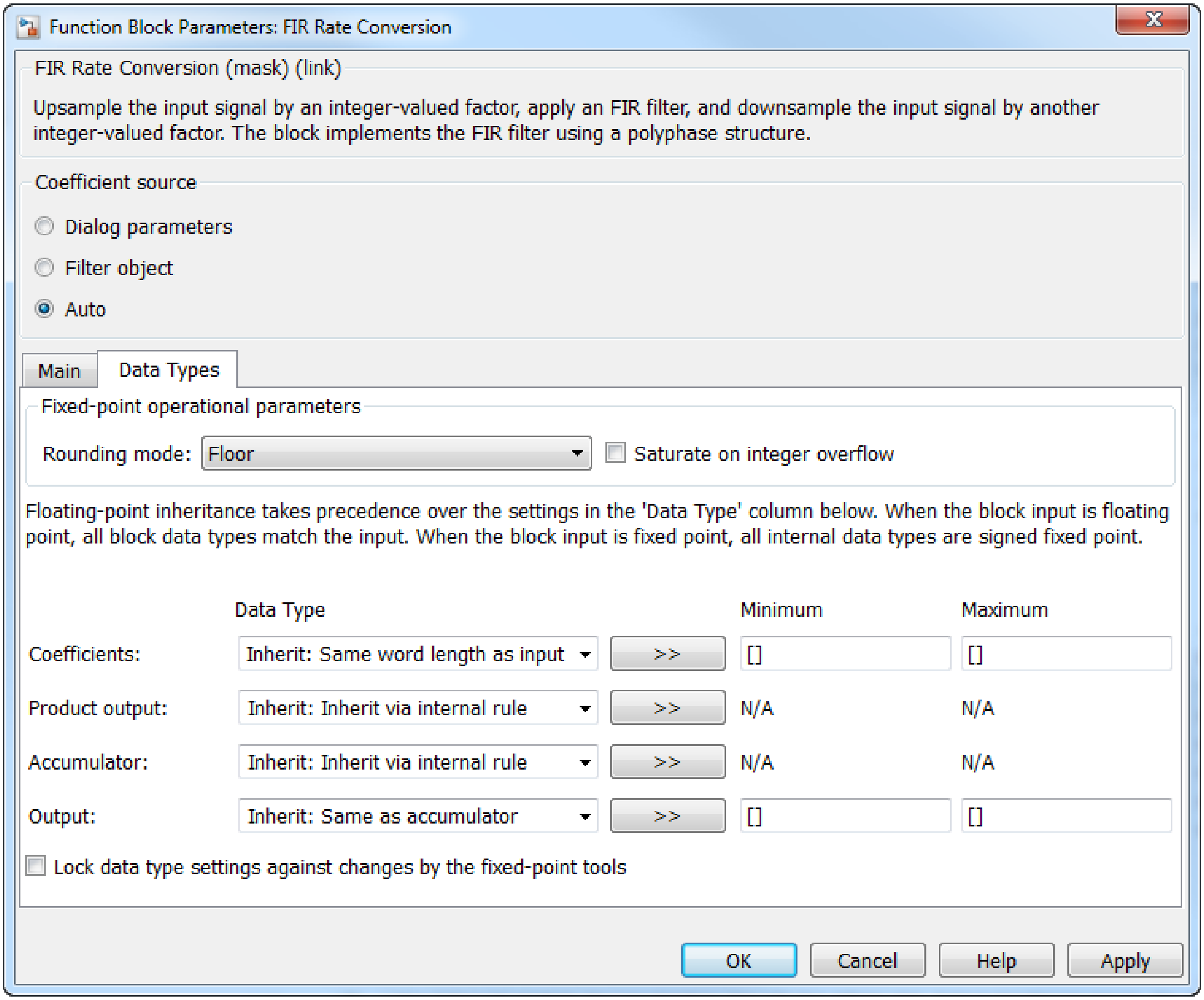
Decimation factor

Specify the integer factor, K, by which to downsample the signal after filtering. The default is 2.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the filter defined in the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

The **Data Types** pane of the FIR Rate Conversion block dialog box appears as follows when you select **Auto** in the **Coefficient source** group box.



Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Note: The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output data type** is `Inherit: Inherit via internal rule`
- **Accumulator data type** is `Inherit: Inherit via internal rule`
- **Output data type** is `Inherit: Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients data type

Specify the coefficients data type. See "Fixed-Point Data Types" on page 2-768 and "Multiplication Data Types" for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Coefficients data type** parameter.

See "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink) for more information.

Coefficients Minimum

Specify the minimum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Coefficients Maximum

Specify the maximum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-768 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-768 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-768 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

`Downsample`	DSP System Toolbox
`Upsample`	DSP System Toolbox
`FIR Decimation`	DSP System Toolbox
`FIR Interpolation`	DSP System Toolbox
`CIC Decimation`	DSP System Toolbox
`CIC Interpolation`	DSP System Toolbox
`FIR Halfband Interpolator`	DSP System Toolbox
`FIR Halfband Decimator`	DSP System Toolbox
`IIR Halfband Interpolator`	DSP System Toolbox
`IIR Halfband Decimator`	DSP System Toolbox
`CIC Compensation Interpolator`	DSP System Toolbox

`CIC Compensation Decimator`	DSP System Toolbox
dsp.CICCompensationDecimator	DSP System Toolbox
dsp.CICCompensationInterpolator	DSP System Toolbox
dsp.FIRHalfbandDecimator	DSP System Toolbox
dsp.FIRHalfbandInterpolator	DSP System Toolbox
dsp.FIRDecimator	DSP System Toolbox
dsp.FIRInterpolator	DSP System Toolbox
`firnyquist`	DSP System Toolbox
`firhalfband`	DSP System Toolbox
`firgr`	DSP System Toolbox
`firceqrip`	DSP System Toolbox

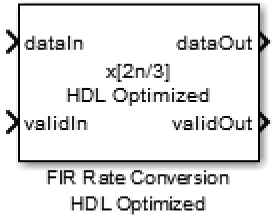
See the following sections for related information:

- "Convert Sample and Frame Rates in Simulink"
- "Multirate and Multistage Filters"

Introduced before R2006a

FIR Rate Conversion HDL Optimized

Upsample, filter, and downsample input signals—optimized for HDL code generation



Library

Filtering/Multirate Filters

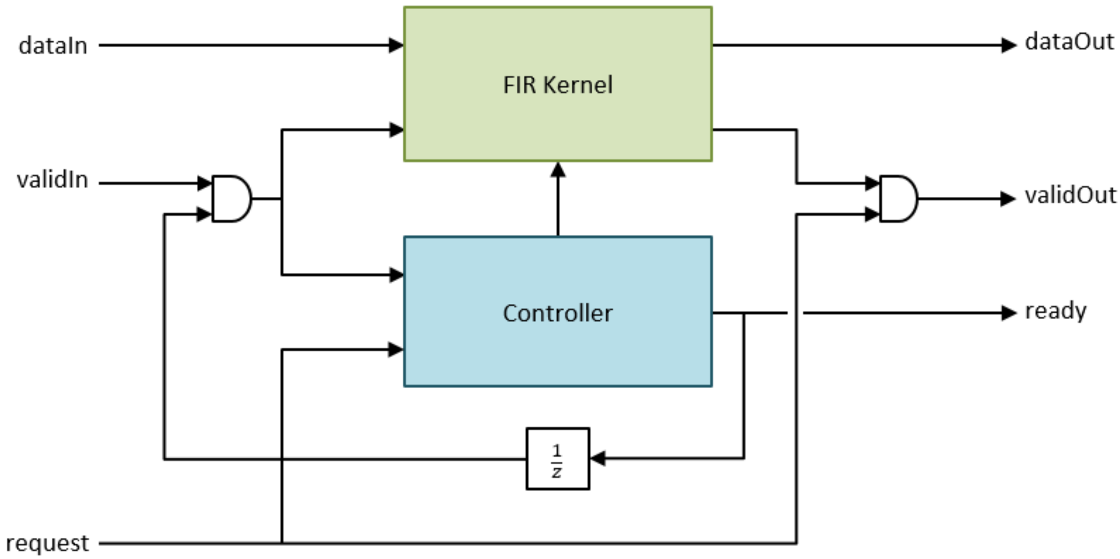
`dspmlti4`

Description

The FIR Rate Conversion HDL Optimized block upsamples, filters, and downsamples input signals. It is optimized for HDL code generation and operates on one sample of each channel at a time. The block implements an efficient polyphase architecture to avoid unnecessary arithmetic operations and high intermediate sample rates.

The block upsamples by an integer factor of L, applies an FIR filter, and downsamples by an integer factor of M.

The block has input and output control ports for pacing the flow of samples. In the default configuration, the block uses `validIn` and `validOut` control signals. For additional flow control, you can enable a `ready` output signal and a `request` input signal.



The `ready` output port indicates that the block can accept a new input data sample on the next time step. When $L \geq M$, you can use the `ready` signal to achieve continuous output data samples. If you apply a new input sample after each time the block returns `ready` = `true`, the block returns a data output sample with `validOut` = `true` on every time step.

When you do not enable the `ready` port, you can apply a valid data sample only every `ceil(L/M)` time steps. For example:

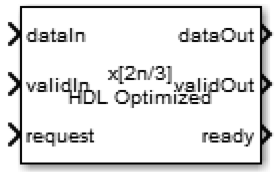
- L/M = 4/5 — You can apply a new input sample on every time step.
- L/M = 3/2 — You can apply a new input sample on every other time step.

When you enable the `request` input port, the block returns the next output sample when `request` is `true` and a valid output sample is available. When you do not use `request` , the block returns output samples when they are available. When no new data is available, the block returns `validOut` = `false`.

You can connect the `request` input port to the `ready` output port of a downstream block.

Signal Attributes

This icon shows all the optional ports of the FIR Rate Conversion HDL Optimized block.



FIR Rate Conversion
HDL Optimized

Port	Direction	Description	Data Type
`dataIn`	Input	Data sample, specified as a scalar, or as a row vector in which each element represents an independent channel. The block accepts real or complex data.	• `fixdt()` • `uint8/16/32`, `int8/16/32` • `double` and `single` are allowed for simulation but not for HDL code generation.
`validIn`	Input	Capture the value of `dataIn`, when `validIn` is `true`. You can apply a valid data sample every `ceil(L/M)` time steps.	`boolean`
`request`	Input (Optional)	Request for a new output sample.	`boolean`
`dataOut`	Output	Resampled and filtered data sample, returned as a scalar, or as a vector in which each element represents an independent channel.	Same as `dataIn`
`validOut`	Output	Qualification of the `dataOut` value. When `validOut` is `true`, `dataOut` is valid.	`boolean`
`ready`	Output (Optional)	Indicates that the block is ready for a new input sample, when `ready` is `true`.	`boolean`

Parameters

Main

Interpolation factor

Upsampling factor, *L*, specified as a scalar integer. The default is 3.

Decimation factor

Downsampling factor, *M*, specified as a scalar integer. The default is 2.

FIR filter coefficients

Filter coefficients, specified as a vector in descending powers of z^{-1}.

You can generate filter coefficients using the Signal Processing Toolbox filter design functions (such as `fir1`). Design a lowpass filter with normalized cutoff frequency no greater than `min(1/L,1/M)`. The block initializes internal filter states to zero. The default coefficients are `firpm(70,[0 .28 .32 1],[1 1 0 0])`.

Enable ready output port

Select this check box to enable a port that indicates when the block is able, on the next time step, to accept a new input data sample.

Enable request input port

Select this check box to enable a port that requests the block return an output sample. When the `request` port is `true`, and there is an output sample available, the block returns a new output sample and sets `validOut` to `true`. When `request` is `false`, or there is no new sample available, the block sets `validOut` to `false`.

Data Types

Rounding mode

The default rounding method for internal fixed point calculations is `Floor`. `Simplest` rounding mode is not supported.

Overflow mode

The default overflow action for internal fixed point calculations is `Wrap`.

Coefficients Data Type

Data type of the FIR filter coefficients, specified as a `fixdt(s,wl,fl)` object with `signedness`, `word length`, and `fractional length` properties. The default is `fixdt(1,16,16)`.

Output Data Type

Data type of the output data samples. You can choose `Inherit: Inherit via internal rule`, `Inherit: Full precision`, or specify a `fixdt(s,wl,fl)` object. The default is `Inherit: Same word length as input`.

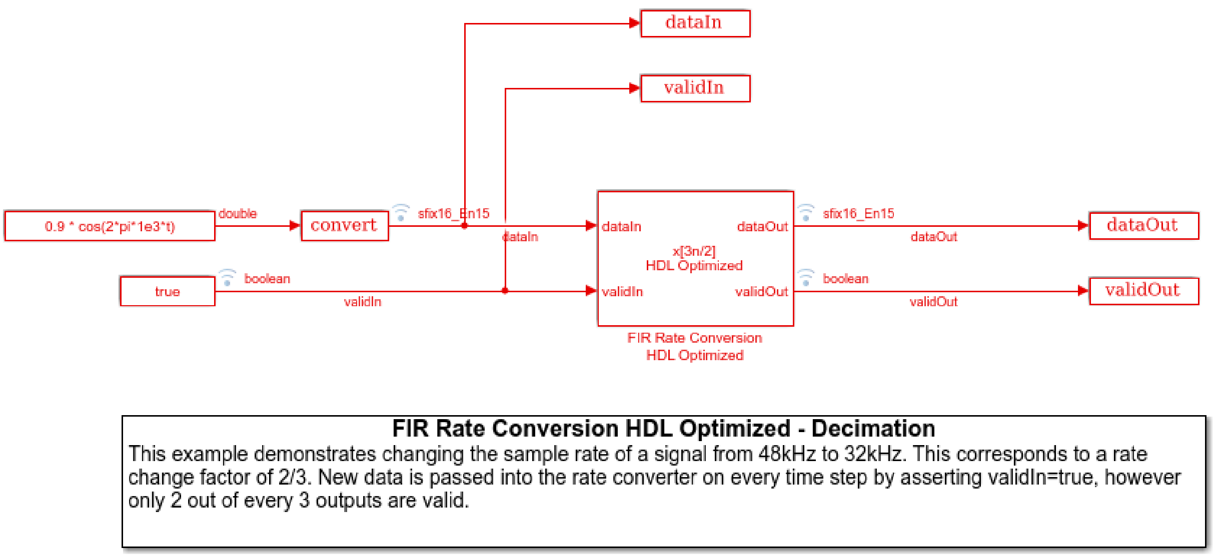
Examples

Downsample a Signal

Convert a signal from 48 kHz to 32 kHz using the FIR Rate Conversion HDL Optimized block.

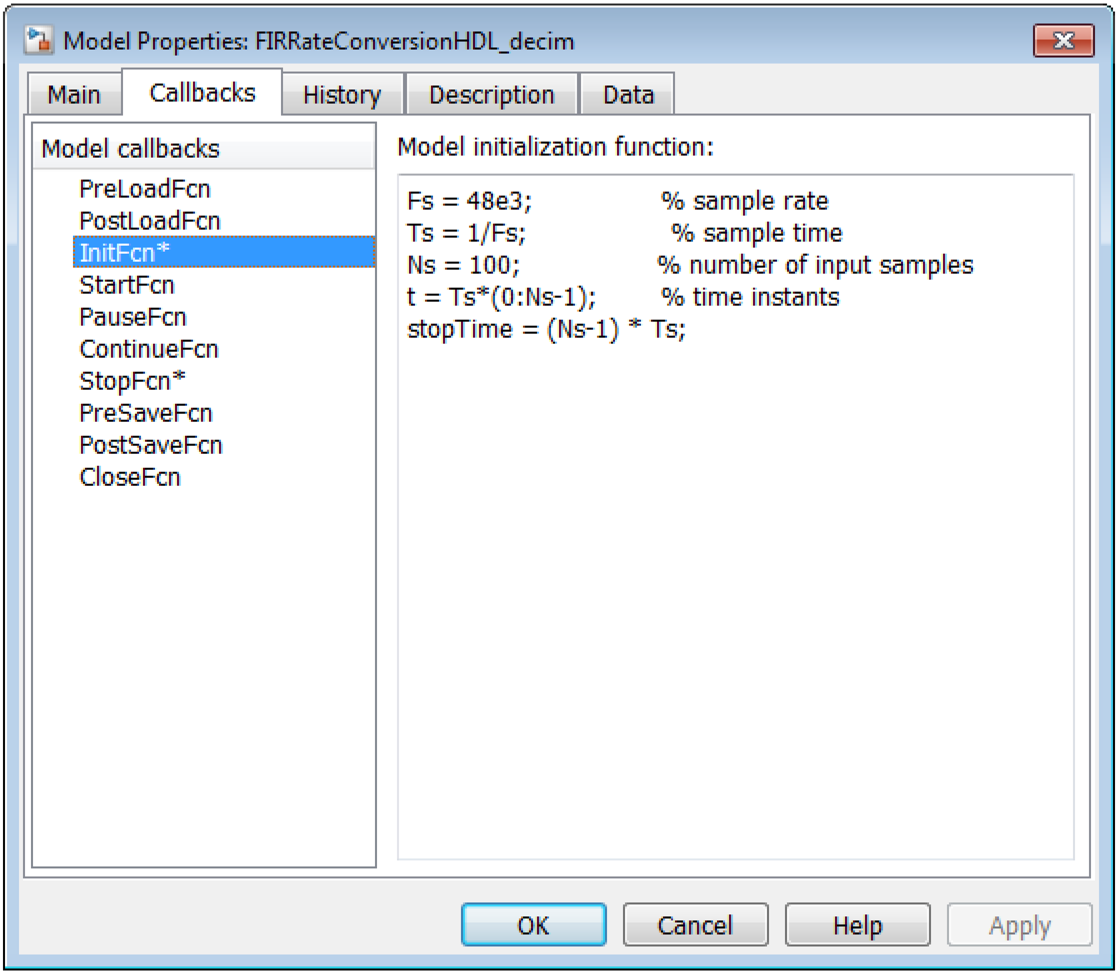
The source is a cosine input signal, sampled at 48kHz. The model passes a new data sample into the block on every time step by holding `validIn` = `true`. After resampling, the `validOut` signal is `true` on only 2/3 of the time steps.

Open the Model



Configure the Model

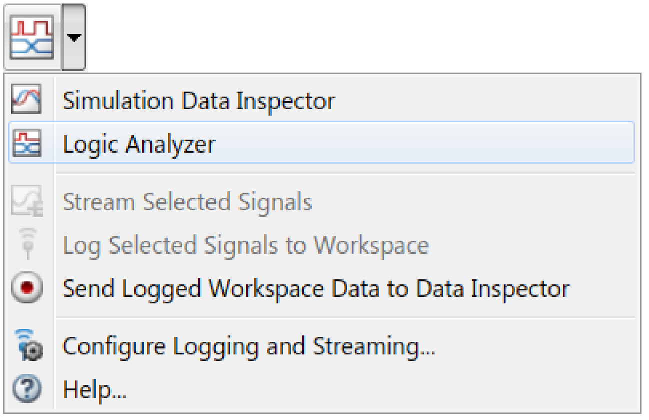
Define the data rate parameters in the `InitFcn` callback.



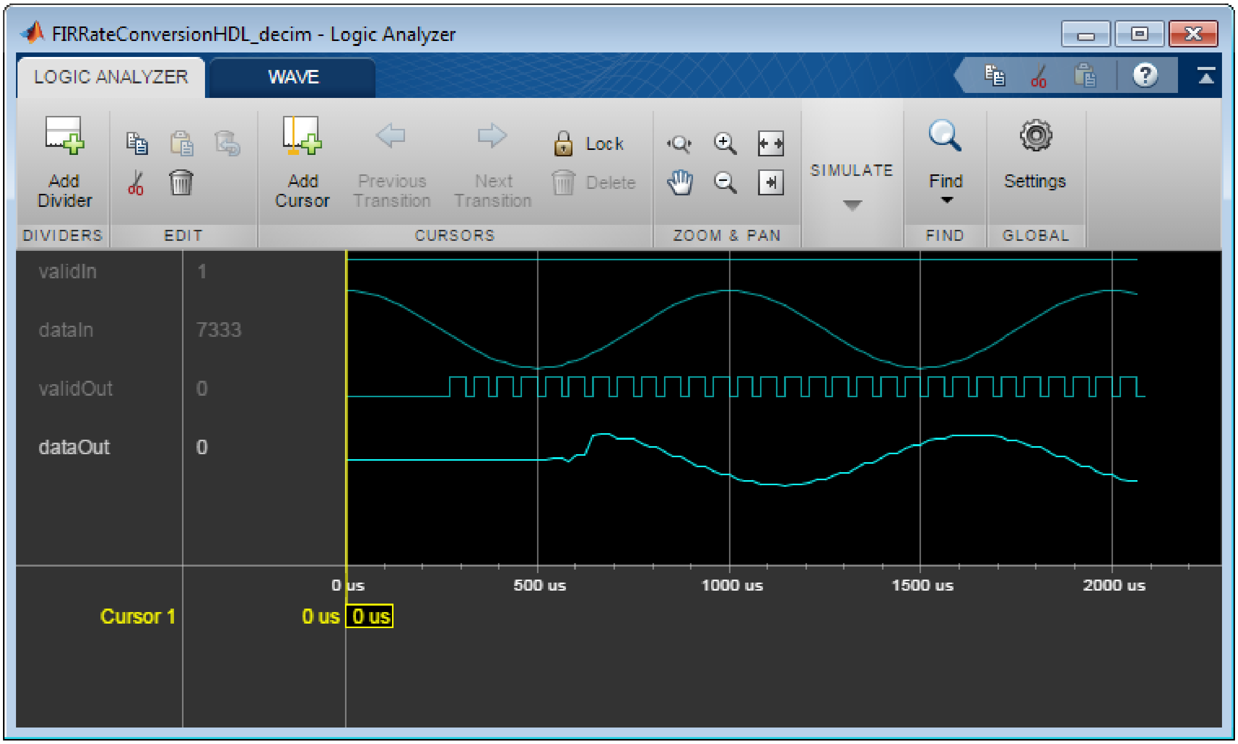
Configure the FIR Rate Conversion HDL Optimized block. Use the default interpolation factor of 2 and decimation factor of 3. Use the `firmpm` function to design an equiripple FIR filter. In the **Data Types** group, set the **Coefficients** data type to `fixdt(1,16,15)` to accommodate the filter you designed.

Run the Model and Display Results

Run the model. Use the **Logic Analyzer** to view the input and output signals of the block. The blue icon in the model indicates streamed signals. Launch the Logic Analyzer from the model's toolbar.



In the Logic Analyzer, note the pattern of `validIn` and the resulting `validOut` signal.



Generate HDL Code

To generate HDL code from the FIR Rate Converter HDL Optimized block, right-click the block and select **Create Subsystem from Selection**. Then right-click the subsystem and select **HDL Code** > **Generate HDL Code for Subsystem**.

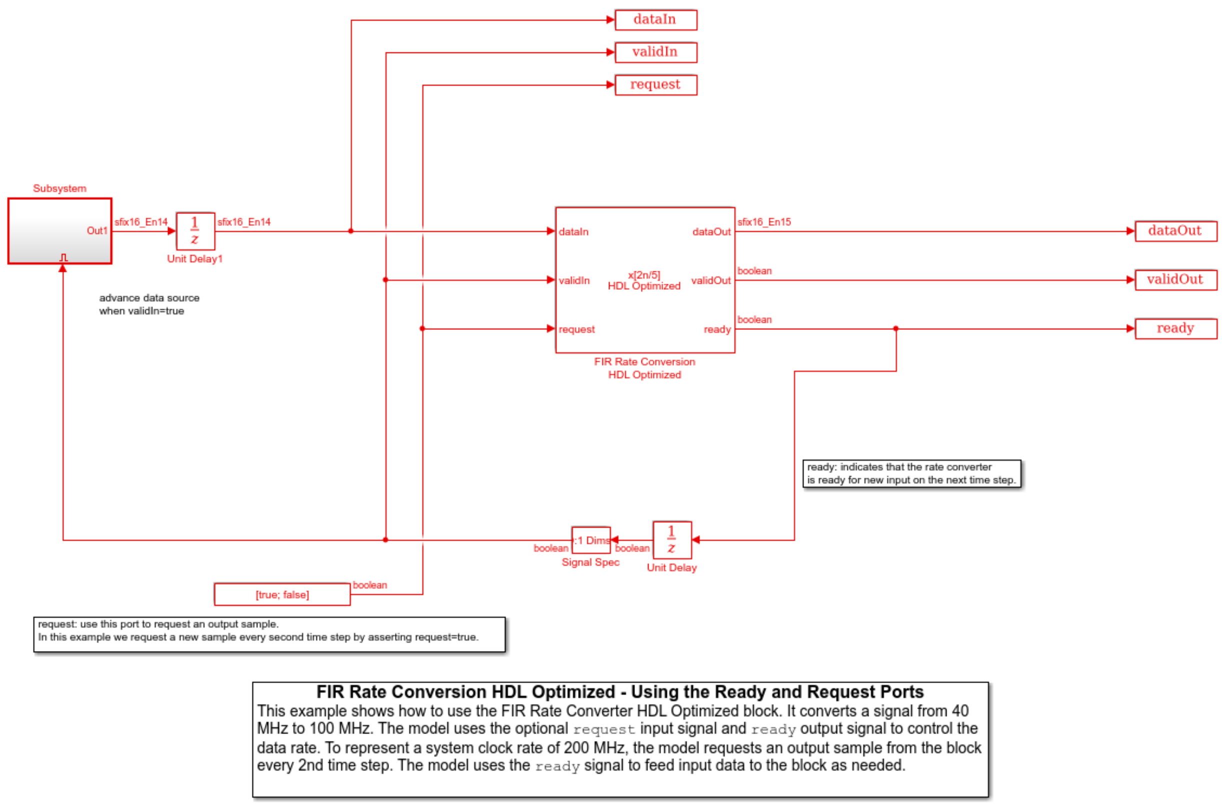
Control Data Rate Using the Ready and Request Ports

Convert a signal from 40 MHz to 100 MHz using the FIR Rate Converter HDL Optimized block. Uses the optional `request` input signal and `ready` output signal to control the data rate.

- To represent a system clock rate of 200MHz, the model connects a repeating true-false signal to the `request` port. This configuration generates output samples at 100 MHz, i.e. every second time step. Alternatively, you can connect this port to the `ready` port of a downstream block.

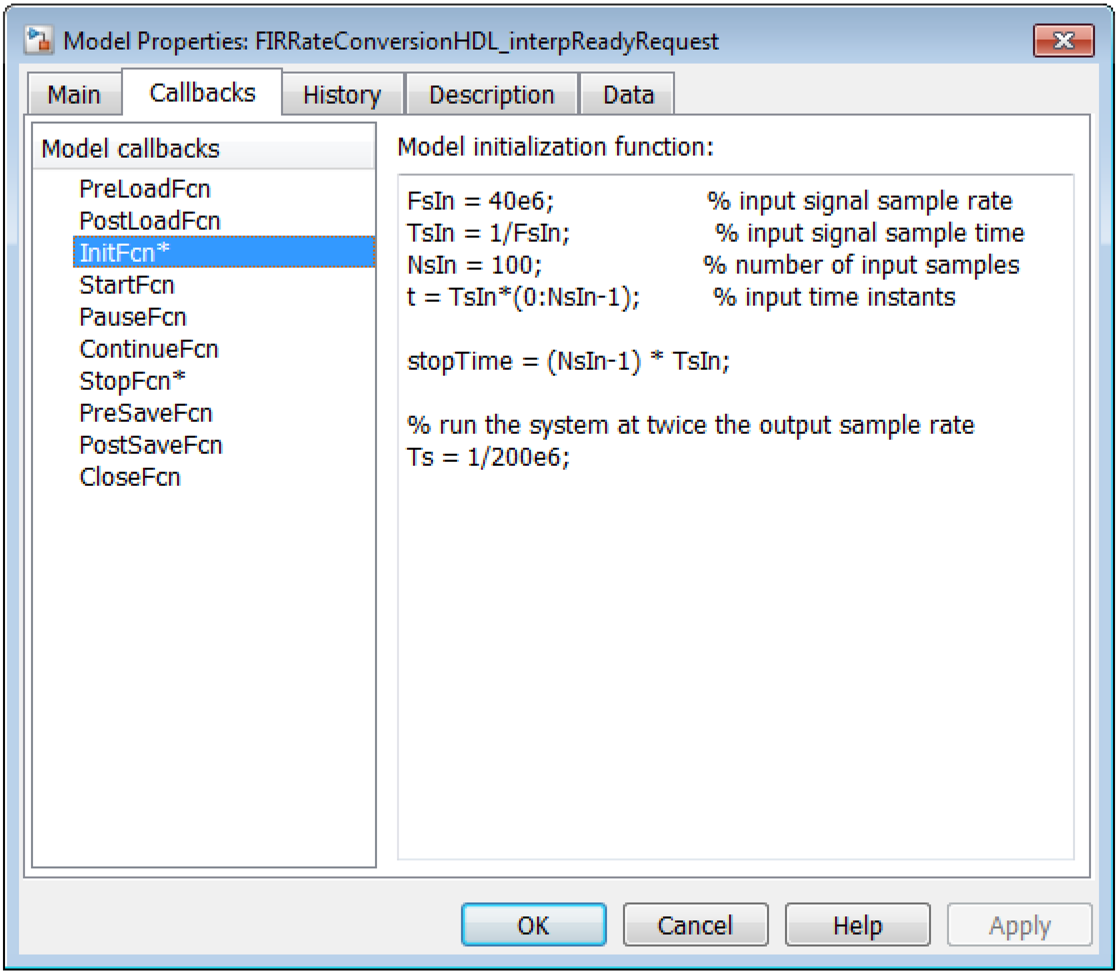
- When the block can accept a new input sample on the next time step, it sets the `ready` output signal to `true`. The model connects this signal to a waveform source that generates one sample at a time.

Open the Model



Configure the Model

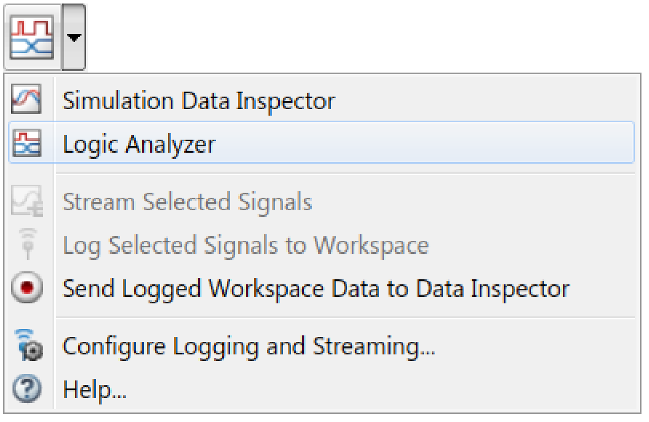
Define the data rate parameters in the `InitFcn` callback.



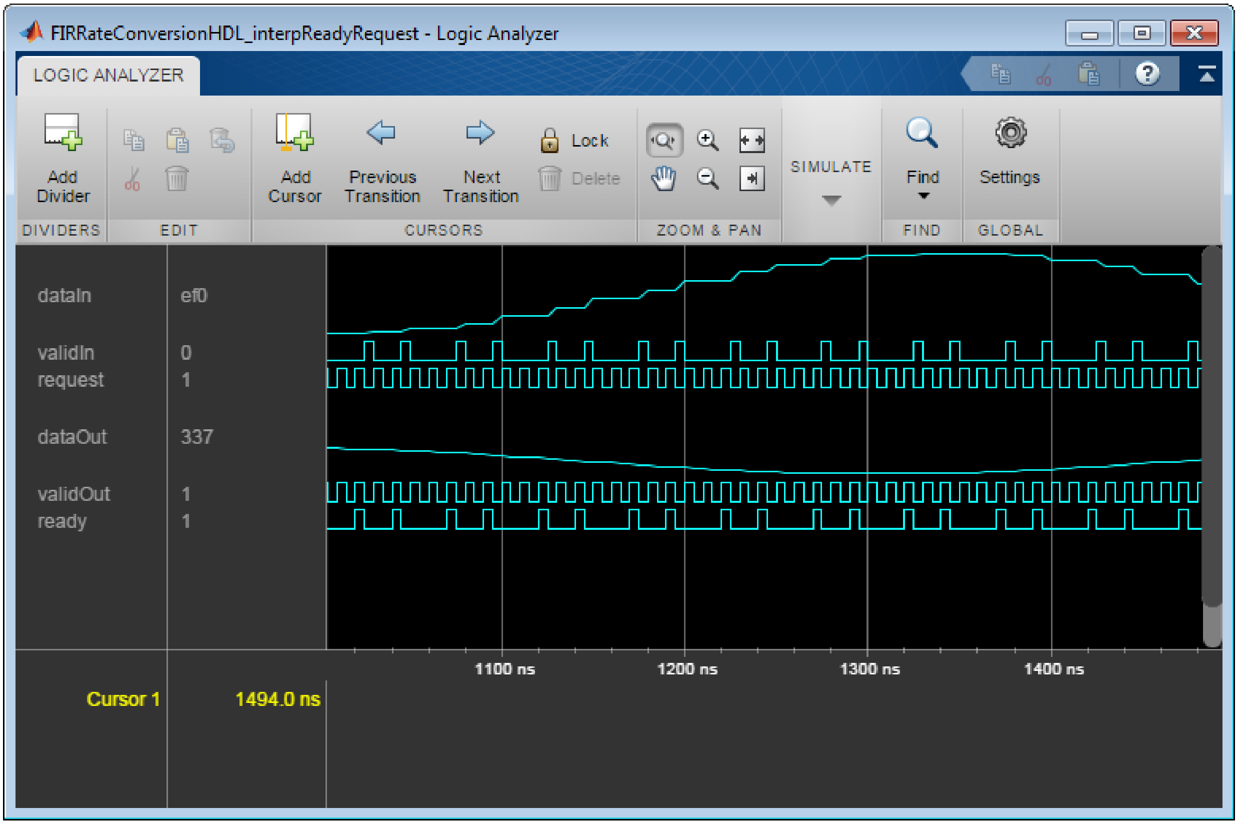
Configure the FIR Rate Conversion HDL Optimized block. Use an interpolation factor of 5 and a decimation factor of 2. Use the `firmpm` function to design an equiripple FIR filter. Select both check boxes to enable the `ready` and `request` ports. % In the **Data Types** group, set the **Coefficients** data type to `fixdt(1,16,15)` to accommodate your filter design.

Run the Model and Display Results

Run the model. Use the **Logic Analyzer** to view the input and output signals of the block. The blue icon in the model indicates streamed signals. Launch the Logic Analyzer from the model's toolbar.



In the **Logic Analyzer**, note the pattern of `request` and the resulting `validOut` signal, and the pattern of `ready` and the resulting `validIn` signal.

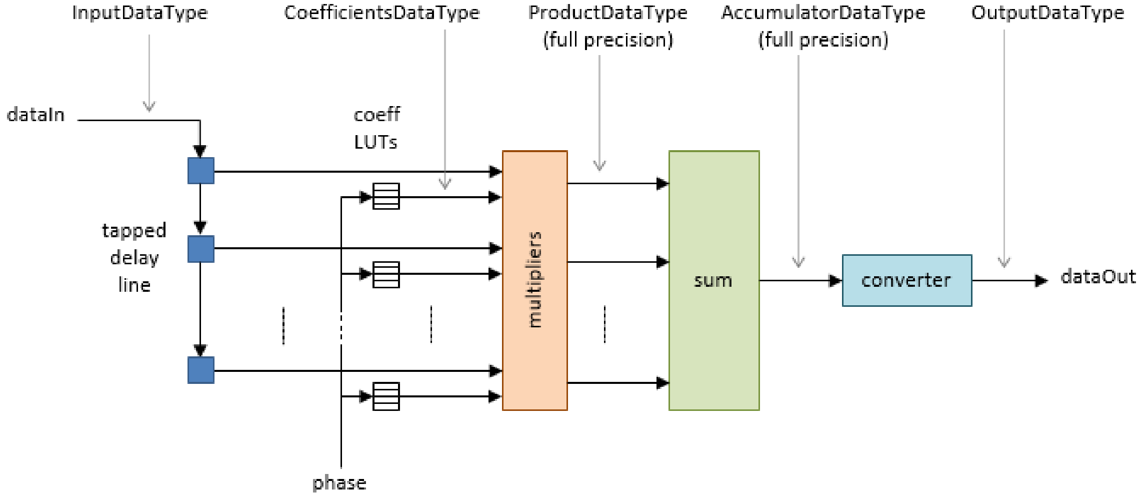


Generate HDL Code

To generate HDL code from the FIR Rate Converter HDL Optimized block, right-click the block and select **Create Subsystem from Selection**. Then right-click the subsystem and select **HDL Code** > **Generate HDL Code for Subsystem**.

Algorithm

The FIR Rate Conversion HDL Optimized block implements a fully parallel polyphase filter architecture. The diagram shows where the block casts the data types, according to your configuration.



Delay

The block models HDL pipeline latency, so there is an initial delay of several time steps before the object returns the first valid output sample. The latency depends on the filter coefficients and the resampling factors. To determine the latency from first sample in to first sample out, observe the `validOut` signal.

Performance

For a sample of design performance, generate HDL for the block as configured in the "Control Data Rate Using the Ready and Request Ports" on page 2-795 example. The example filter resamples at 5/2, and uses a symmetric 71-tap filter. The input samples and filter coefficients are 16 bits wide. The design was targeted to a Xilinx Virtex-6 FPGA, using Xilinx ISE synthesis and place and route tools.

After place and route, the design achieves 535 MHz clock frequency. It uses these resources.

LUT	592
FFS	979
Xilinx LogiCORE DSP48	15

Block RAM (16K)	0

Performance of the synthesized HDL code varies depending on your filter coefficients, FPGA target, and synthesis options.

See Also

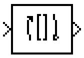
See Also

dsp.HDLFIRRateConverter | `FIR Rate Conversion`

Introduced in R2015b

Flip

Flip input vertically or horizontally



Library

Signal Management / Indexing

`dspindex`

Description

The Flip block vertically or horizontally reverses the M-by-N input matrix, u. The output always has the same dimensionality as the input.

When you set the **Flip along** parameter to `Columns`, the block flips the input *vertically* so the first row of the input becomes the last row of the output.

```
y = flipud(u)      % Equivalent MATLAB code
```

When flipping the input vertically, the block treats length-M unoriented vector inputs as M-by-1 column vectors.

When you set the **Flip along** parameter to `Rows`, the block flips the input *horizontally* so the first column of the input becomes the last column of the output.

```
y = fliplr(u)      % Equivalent MATLAB code
```

When flipping the input horizontally, the block treats length-N unoriented vector inputs as 1-by-N row vectors.

This block supports Simulink virtual buses.

Parameters

Flip along

Specify the dimension along which to flip the input. When you set this parameter to `Columns`, the block flips the input vertically. When you set this parameter to `Rows`, the block flips the input horizontally.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated

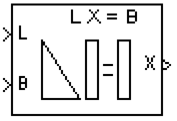
See Also

Selector	Simulink
Variable Selector	DSP System Toolbox
`flipud`	MATLAB
`fliplr`	MATLAB

Introduced before R2006a

Forward Substitution

Solve $LX=B$ for X when L is lower triangular matrix



Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

`dspsolvers`

Description

The Forward Substitution block solves the linear system $LX=B$ by simple forward substitution of variables, where:

- L is the lower triangular M-by-M matrix input to the L port.
- B is the M-by-N matrix input to the B port.

The M-by-N matrix output X is the solution of the equations. The block does not check the rank of the inputs.

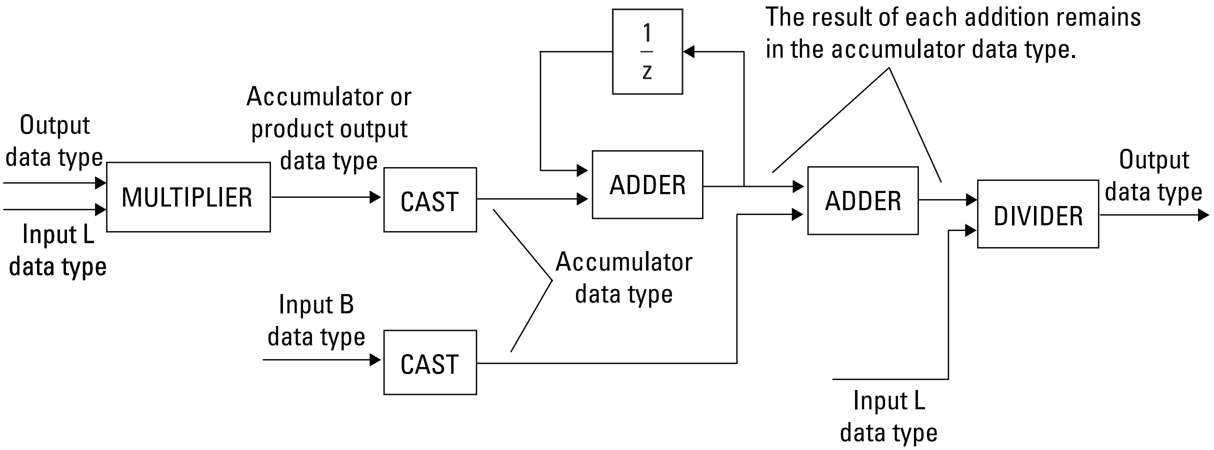
The block only uses the elements in the *lower triangle* of input L and ignores the upper elements. When you select **Input L is unit-lower triangular**, the block assumes the elements on the diagonal of L are 1s. This is useful when matrix L is the result of another operation, such as an LDL decomposition, that uses the diagonal elements to represent the D matrix.

The block treats a length-M vector input at port B as an M-by-1 matrix.

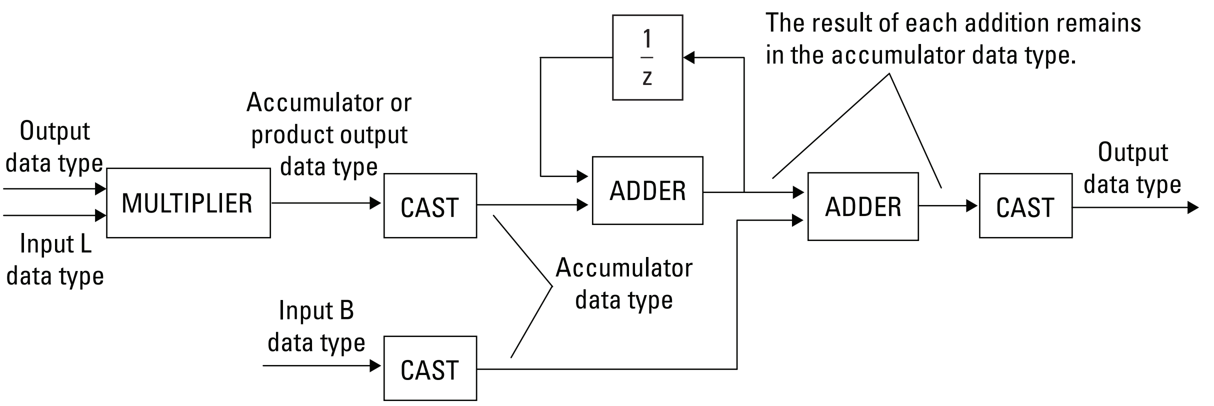
Fixed-Point Data Types

The following diagram shows the data types used within the Forward Substitution block for fixed-point signals.

When input L is not unit-lower triangular:



When input L is unit-lower triangular:

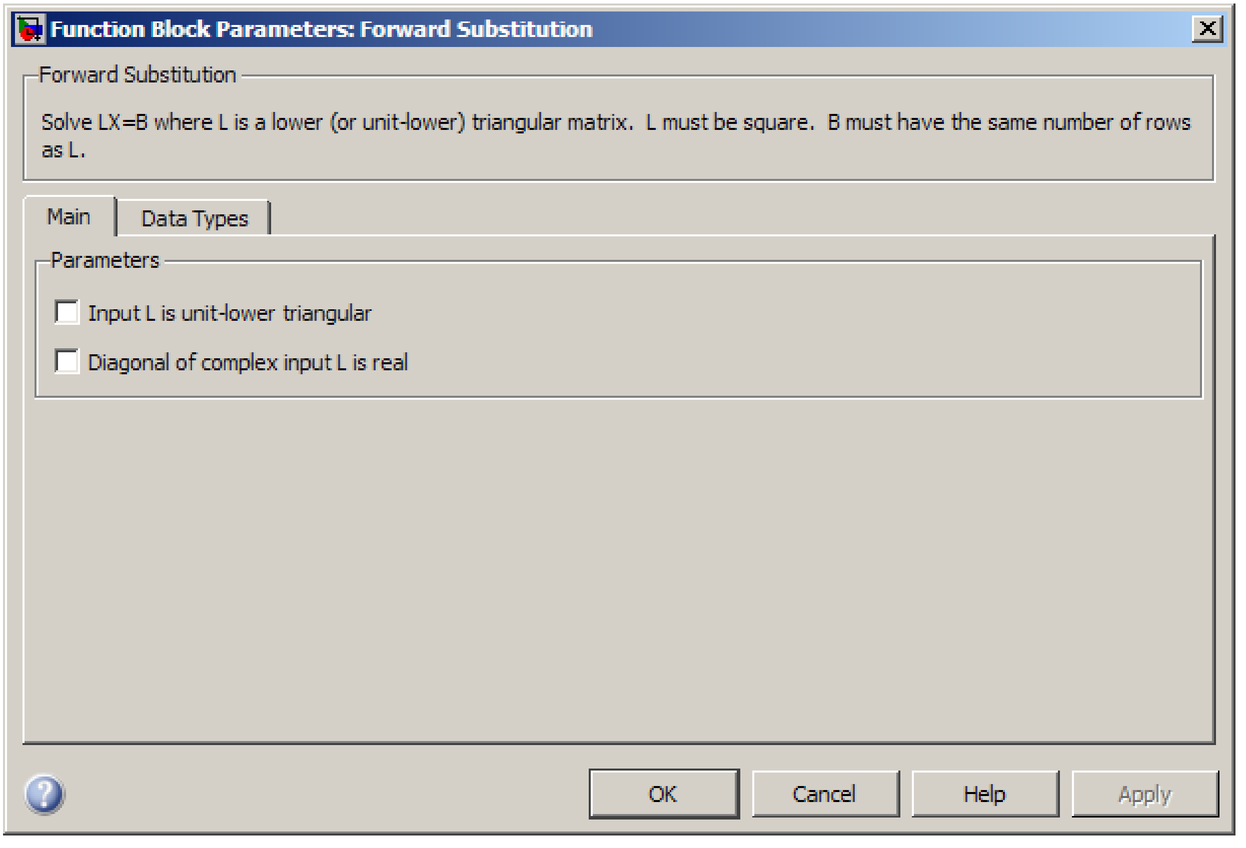


You can set the product output, accumulator, and output data types in the block dialog box, as discussed in the following section.

The output of the multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

Dialog Box

The **Main** pane of the Forward Substitution block dialog box appears as follows.



Input L is unit-lower triangular

Select this check box only when all elements on the diagonal of L have a value of 1. When you do so, the block optimizes its behavior by skipping an unnecessary divide operation.

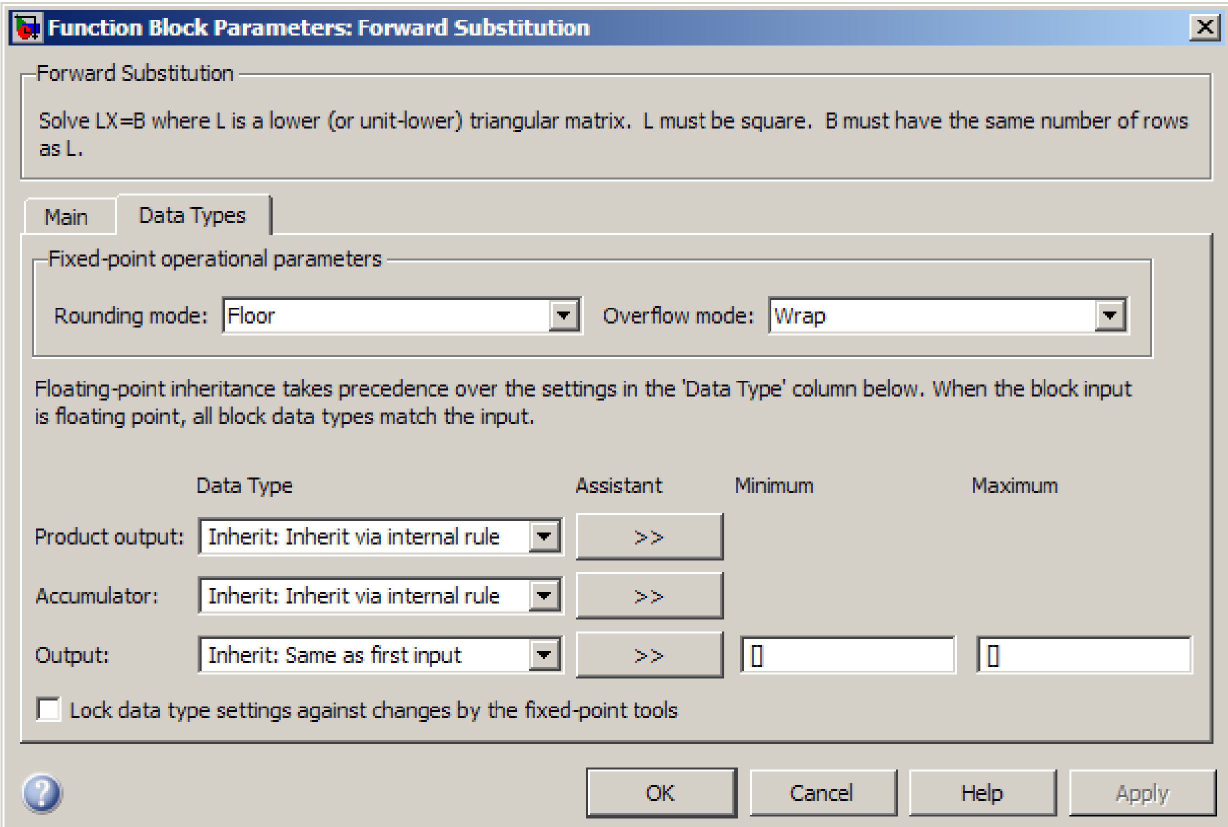
Do not select this check box if there are any elements on the diagonal of L that do not have a value of 1. When you clear the **Input L is unit-lower triangular** check box, the block always performs the necessary divide operation.

Diagonal of complex input L is real

Select to optimize simulation speed when the diagonal elements of complex input L are real. This parameter is only visible when **Input L is unit-upper triangular** is not selected.

Note: When L is a complex fixed-point signal, you must select either **Input L is unit-lower triangular** or **Diagonal of complex input L is real**. In such a case, the block ignores any imaginary part of the diagonal of L.

The **Data Types** pane of the Forward Substitution block dialog box appears as follows.



Note: Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-804 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-804 for illustrations depicting the use of the accumulator data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-804 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as first input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Minimum

Specify the minimum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
L	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
B	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
X	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

See Also

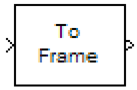
`Backward Substitution`	DSP System Toolbox
`Cholesky Solver`	DSP System Toolbox
`LDL Solver`	DSP System Toolbox
`Levinson-Durbin`	DSP System Toolbox
`LU Solver`	DSP System Toolbox
`QR Solver`	DSP System Toolbox

See “Linear System Solvers” for related information.

Introduced before R2006a

Frame Conversion

Specify sampling mode of output signal



Library

Signal Management / Signal Attributes

`dspsigattribs`

Description

The Frame Conversion block passes the input through to the output and sets the output sampling mode to the value of the **Sampling mode of output signal** parameter, which can be either `Frame-based` or `Sample-based`. The output sampling mode can also be inherited from the signal at the Ref (reference) input port, which you make visible by selecting the **Inherit output sampling mode from <Ref> input port** check box.

The Frame Conversion block does not make any changes to the input signal other than the sampling mode. In particular, the block does not rebuffer or resize 2-D inputs. Because 1-D vectors cannot be frame based, when the input is a length-M 1-D vector and the block is in `Frame-based` mode, the output is a frame-based M-by-1 matrix — that is, a single channel.

Parameters

Inherit output sampling mode from <Ref> input port

Select to enable the Ref port from which the block inherits the output sampling mode.

Sampling mode of output signal

Specify the sampling mode of the output signal, `Frame-based` or `Sample-based`.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Frame Conversion.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated
Ref	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated

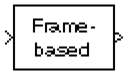
See Also

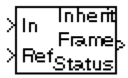
Buffer	DSP System Toolbox
Check Signal Attributes	DSP System Toolbox
Convert 1-D to 2-D	DSP System Toolbox
Convert 2-D to 1-D	DSP System Toolbox
Inherit Complexity	DSP System Toolbox
Unbuffer	DSP System Toolbox
Probe	Simulink
Reshape	Simulink
Signal Specification	Simulink

Introduced before R2006a

Frame Status Conversion (Obsolete)

Specify frame status of output as sample based or frame based





Library

`dspobslib`

Description

Note The Frame Status Conversion block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the `Frame Conversion` block.

The Frame Status Conversion block passes the input through to the output, and sets the output frame status to the **Output signal** parameter, which can be either `Frame-based` or `Sample-based`. The output frame status can also be inherited from the signal at the `Ref` (reference) input port, which is made visible by selecting the **Inherit output frame status from Ref input port** check box.

When the **Output signal** parameter setting or the inherited signal's frame status differs from the input frame status, the block changes the input frame status accordingly, but does not otherwise alter the signal. In particular, the block does not rebuffer or resize 2-D inputs. Because 1-D vectors cannot be frame based, when the input is a length-M 1-D vector, and the **Output signal** parameter is set to `Frame-based`, the output is a frame-based M-by-1 matrix (that is, a single channel).

When the **Output signal** parameter or the inherited signal's frame status matches the input frame status, the block passes the input through to the output unaltered.

Parameters

Inherit output frame status from Ref input port

When selected, enables the `Ref` input port from which the block inherits the output frame status.

Output signal

The output frame status, `Frame-based` or `Sample-based`.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Ref	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

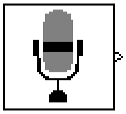
See Also

Check Signal Attributes	DSP System Toolbox
Convert 1-D to 2-D	DSP System Toolbox
Convert 2-D to 1-D	DSP System Toolbox
Inherit Complexity	DSP System Toolbox

Introduced in R2008b

From Audio Device

Read audio data from computer's audio device



Note: The From Audio Device block will be removed in a future release. Existing instances of the block continue to run. For new models, use the Audio Device Reader block from Audio System Toolbox instead.

Library

Sources

`dspsrcs4`

Description

The From Audio Device block reads audio data from an audio device in real time. This block is not supported for use with the Simulink Model block.

Use the **Device** parameter to specify the device from which to acquire audio. This parameter is automatically populated based on the audio devices installed on your system. If you plug or unplug an audio device from your system, type `clear mex` at the MATLAB command prompt to update this list.

Use the **Number of channels** parameter to specify the number of audio channels in the signal. For example:

- Enter 2 if the audio source is two channels (stereo).
- Enter 1 if the audio source is single channel (mono).
- Enter 6 if you are working with a 5.1 speaker system.

The block's output is an M-by-N matrix, where M is the number of consecutive samples and N is the number of audio channels.

Use the **Sample rate (Hz)** parameter to specify the number of samples per second in the signal. If the audio data is processed in uncompressed pulse code modulation (PCM) format, it should typically be sampled at one of the standard audio device rates: 8000, 11025, 22050, 44100, or 48000 Hz.

The range of supported audio device sample rates and data type formats, depend on both the sound card and the API which is chosen for the sound card.

Use the **Device data type** parameter to specify the data type of the audio data that the device is placing in the buffer. You can choose:

- `8-bit integer`
- `16-bit integer`
- `24-bit integer`
- `32-bit float`
- `Determine from output data type`

If you choose `Determine from output data type`, the following table summarizes the block's behavior.

Output Data Type	Device Data Type
Double-precision floating point or single-precision floating point	32-bit floating point
32-bit integer	24-bit integer
16-bit integer	16-bit integer
8-bit integer	8-bit integer

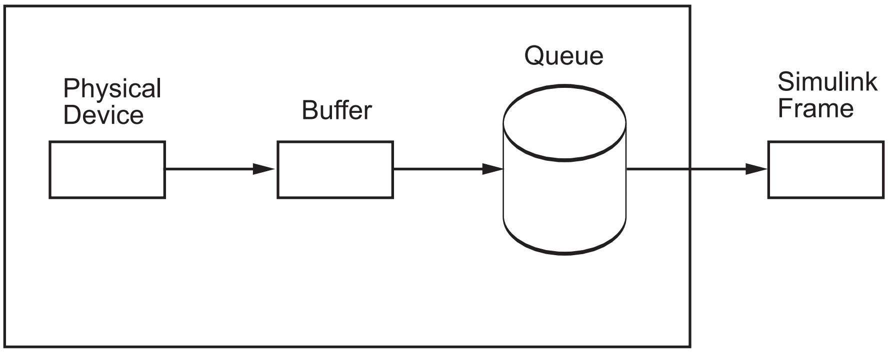
If you choose `Determine from output data type` and the device does not support a data type, the block uses the next lowest precision data type supported by the device.

Use the **Frame size (samples)** parameter to specify the number of samples in the block's output. Use the **Output data type** parameter to specify the data type of audio data output by the block.

The generated code for this block relies on prebuilt .dll files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra .dll files when doing so. The packNGo function creates a single zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

Buffering

The From Audio Device block buffers the data from the audio device using the process illustrated by the following figure.



From Audio Device Block

1. At the start of the simulation, the audio device begins writing the input data to a buffer. This data has the data type specified by the **Device data type** parameter.
2. When the buffer is full, the From Audio Device block writes the contents of the buffer to the queue. Specify the size of this queue using the **Queue duration (seconds)** parameter.
3. As the audio device appends audio data to the bottom of the queue, the From Audio Device block pulls data from the top of the queue to fill the Simulink frame. This data has the data type specified by the **Output data type** parameter.

Select the **Automatically determine buffer size** check box to allow the block to calculate a conservative buffer size using the following equation:

$$size = 2^{\left\lfloor \log_2 \frac{sr}{10} \right\rfloor}$$

In this equation, *size* is the buffer size, and *sr* is the sample rate. If you clear this check box, the **Buffer size (samples)** parameter appears on the block. Use this parameter to specify the buffer size in samples.

When the simulation throughput rate is lower than the hardware throughput rate, the queue, which is initially empty, fills up. If the queue is full, the block drops the incoming data from the audio device. You can monitor dropped samples using the optional `Overrun` output port. When the simulation throughput rate is higher than the hardware throughput rate, the From Audio Device block waits for new samples to become available.

Channel Mapping

The term *Channel Mapping* refers to a 1-to-1 mapping that associates channels on the selected audio device to channels of the data. When you record audio, channel mapping allows you to specify which channel of the audio data directs input to a specific channel of audio. You can specify channel mapping as a vector of audio channel indices corresponding to each channel of data being read. The default value in the **Device Input Channels** parameter is 1:MAXINPUTCHANNELS. If you do not select the default mapping, you must specify the **Device Input Channels** parameter in the dialog box.

Example: The selected input audio device contains 8 channels. You want to read data from only channels 2, 4, 6, and redirect the data as follows:

- Audio Device channel 2 to first data channel
- Audio Device channel 4 to second data channel
- Audio Device channel 6 to third data channel

Thus you would specify the **Device Input Channels** as [2 4 6].

Troubleshooting

Not Keeping Up in Real Time

When Simulink cannot keep up with an audio device that is operating in real time, the queue fills up and the block begins to lose audio data. Select the **Output number of samples by which the queue was overrun** check box to add an output port indicating when the queue was full. Here are several ways to deal with this situation:

- *Increase the queue duration.*

  The **Queue duration (seconds)** parameter specifies the duration of the signal, in seconds, that can be buffered during the simulation. This is the maximum length of time that the block's data demand can lag behind the hardware's data supply.
- *Increase the buffer size.*

The size of the buffer processed in each interrupt from the audio device affects the performance of your model. If the buffer is too small, a large portion of hardware resources are used to write data to the queue. If the buffer is too big, Simulink must wait for the device to fill the buffer before it moves the data to the queue, which introduces latency.

- *Increase the simulation throughput rate.*

  Two useful methods for improving simulation throughput rates are increasing the signal frame size and compiling the simulation into native code:

  - Increase frame sizes and convert sample-based signals to frame-based signals throughout the model to reduce the amount of block-to-block communication overhead. This can increase throughput rates in many cases. However, larger frame sizes generally result in greater model latency due to initial buffering operations.
  - Generate executable code with Simulink Coder code generation software. Native code runs much faster than Simulink and should provide rates adequate for real-time audio processing.

Other ways to improve throughput rates include simplifying the model and running the simulation on a faster PC processor. For other ideas on improving simulation performance, see "Delay and Latency" and "Performance" (Simulink).

Running an Executable Outside MATLAB

To run your generated standalone executable application in Shell, you need to set your environment to the following:

Platform	Command

Mac	`setenv DYLD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/maci64 (csh/tcsh)`
	
	`export DYLD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/maci64 (Bash)`

Linux	`setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/glnxa64 (csh/tcsh)`

Platform	Command
	`export LD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/ bin/glnxa64 (Bash)`
Windows	`set PATH = $MATLABROOT\bin\win64; %PATH%`

Audio Hardware API

In order to communicate with the audio hardware on a given computer, the To Audio Device and From Audio Device blocks use the open-source PortAudio library. The PortAudio library supports a range of APIs designed to communicate with the audio hardware on a given platform. The following API choices were made when building the PortAudio library for the DSP System Toolbox product:

- Windows®: DirectSound, WDM—KS, ASIO™
- Linux®: OSS, ALSA
- Mac: CoreAudio

For Windows, the default is DirectSound, for Linux, the default is ALSA, and for Mac there is only one choice.

To determine the audio hardware API currently selected, type the following command in the MATLAB command prompt.

```
getpref('dsp','portaudioHostApi')
```

The output is a scalar indicating the choice of the API.

- 1 — DirectSound
- 3 — ASIO
- 7 — OSS
- 8 — ALSA
- 11 — WDM-KS

To select a particular API, type the following command in the MATLAB command prompt.

```
setpref('dsp','portaudioHostApi',N)
```

where *N* is a scalar. Choose *N* based on the API choice above.

Parameters

Device

Specify the device from which to acquire audio data.

Use default mapping between Device Input Channels and Data

Select this check box to have the default mapping, where the data from the first channel of audio device is sent to the first channel of the input data, data from second channel of audio device is sent to second channel of data and so on. The maximum number of channels in the input data is determined by the **Number of channels** property.

Number of channels

Specify the number of audio channels. This parameter is visible when the **Use default mapping between Device Input Channels and Data** check box is enabled.

Device Input Channels

Specify the channel mapping. This parameter is visible when the **Use default mapping between Device Input Channels and Data** check box is disabled.

Sample rate (Hz)

Specify the number of samples per second in the signal.

Device data type

Specify the data type used by the device to acquire audio data.

Automatically determine buffer size

Select this check box to enable the block to use a conservative buffer size.

Buffer size (samples)

Specify the size of the buffer that the block uses to communicate with the audio device. This parameter is visible when the **Automatically determine buffer size** check box is cleared.

Queue duration (seconds)

Specify the size of the queue in seconds.

Output number of samples by which the queue was overrun

Select this check box to output the number of samples lost to queue overrun since the last transfer of a frame from the audio device. You can use this value to debug throughput problems and adjust the queues and buffers in your model. To learn how to improve throughput, see “Troubleshooting” on page 2-821.

Frame size (samples)

Specify the number of samples in the block's output signal.

Output data type

Select the data type of the block's output.

Supported Data Types

Port	Supported Data Types
Output	• Double-precision floating point • Single-precision floating point • 32-bit signed integers • 16-bit signed integers • 8-bit unsigned integers
Overrun	32-bit unsigned integer

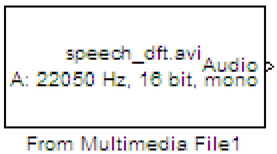
See Also

`From Multimedia File`	DSP System Toolbox
`To Audio Device`	DSP System Toolbox
`audiorecorder`	MATLAB
dsp.AudioRecorder	System Object

Introduced in R2007b

From Multimedia File

Stream from multimedia file



Library

Sources

`dspsrcs4`

Description

The From Multimedia File block reads audio samples, video frames, or both from a multimedia file. The block imports data from the file into a Simulink model.

Note: This block supports code generation for the host computer that has file I/O available. You cannot use this block with Simulink Desktop Real-Time™ software because that product does not support file I/O.

The generated code for this block relies on prebuilt library files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra library files when doing so. The packNGo function creates a single zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

To run an executable file that was generated from a model containing this block, you may need to add precompiled shared library files to your system path. See “Understanding C Code Generation” for details.

Supported Platforms and File Types

The supported file formats available to you depends on the codecs installed on your system.

Windows Platforms Supported File Formats

With the necessary Windows DirectShow codecs installed on your system, the From Multimedia File Block supports many video and audio file formats. This block performs best on platforms with Version 9.0 or later of DirectX® software.

The following table lists the most common file formats.

Multimedia Types	File Name Extensions
Image files	`.jpg`,`.bmp`,`.png`
Video files	`.qt`, `.mov`, `.avi`, `.asf`, `.asx`, `.wmv`, `.mpg`, `.mpeg`, `.mp2`, `.mp4`, `.m4v`
Audio files	`.wav`, `.wma`, `.avi`, `.aif`, `.aifc`, `.aiff`, `.mp3`, `.au`, `.snd, .mp4, .m4a, .flac, .ogg`

The default for image files is `.png`, , for video files is `.avi`, and for audio files is `.mp3`.

Windows XP x64 platform ships with a limited set of 64-bit video and audio codecs. If the From Multimedia File block cannot work on a compressed multimedia file, try one of the two alternatives:

- Run the 32-bit version of MATLAB on your Windows XP x64 platform. Windows XP x64 ships with many 32-bit codecs.
- Save the multimedia file to a file format supported by the From Multimedia File block.

If you use Windows, use Windows Media® Player Version 11 or later with this block for best results.

Non-Windows Platform Supported File Formats

The following table lists the most common file formats.

Multimedia Types	File Name Extensions
Video files	`.avi`, `.mj2`, `.mov`, `.mp4`,`.m4v`
Audio files	`.avi, .mp3, .mp4, .m4a, .wav, .flac, .ogg, .aif, .aifc, .aiff,`

The default for video files is `.avi`, and for audio files is `.mp3`.

Ports

The output ports of the From Multimedia File block change according to the content of the multimedia file. If the file contains only video frames, the **Image**, intensity **I**, or **R**,**G**,**B** ports appear on the block. If the file contains only audio samples, the **Audio** port appears on the block. If the file contains both audio and video, you can select the data to emit. The following table describes available ports.

Port	Description
Image	M-by-N-by-P color video signal where P is the number of color planes.
I	M-by-N matrix of intensity values.
R, **G**, **B**	Matrix that represents one plane of the RGB video stream. Outputs from the R, G, or B ports must have same dimensions.
Audio	Vector of audio data.
Y, Cb, Cr	Matrix that represents one frame of the YCbCr video stream. The Y, Cb, Cr ports produce the following outputs: Y: $M\ x\ N$ Cb: $M\ x\ \frac{N}{2}$ Cr: $M\ x\ \frac{N}{2}$

Sample Rates

The sample rate that the block uses depends on the audio and video sample rate. While the FMMF block operates at a single rate in Simulink, the underlying audio and video streams can produce different rates. In some cases, when the block outputs both audio and video, makes a small adjustment to the video rate.

Sample Time Calculations Used for Video and Audio Files

Sample time = $\dfrac{ceil(AudioSampleRate/FPS)}{AudioSampleRate}$.

When audio sample time, $\dfrac{AudioSampleRate}{FPS}$ is noninteger, the equation cannot reduce to $\dfrac{1}{FPS}$.

In this case, to prevent synchronization problems, the block drops the corresponding video frame when the audio stream leads the video stream by more than $\dfrac{1}{FPS}$.

In summary, the block outputs one video frame at each Simulink time step. To calculate the number of audio samples to output at each time step, the block divides the audio sample rate by the video frame rate (fps). If the audio sample rate does not divide evenly by the number of video frames per second, the block rounds the number of audio samples up to the nearest whole number. If necessary, the block periodically drops a video frame to maintain synchronization for large files.

Dialog Box

Main Tab

File name

Specify the name of the multimedia file from which to read. The block determines the type of file (audio and video, audio only, or video only) and provides the associated parameters.

If the location of the file does not appear on your MATLAB path, use the **Browse** button to specify the full path. Otherwise, if the location of this file appears on your MATLAB path, enter only the file name. On Windows platforms, this parameter supports URLs that point to MMS (Microsoft Media Server) streams.

Inherit sample time from file

Select the **Inherit sample time from file** check box if you want the block sample time to be the same as the multimedia file. If you clear this check box, enter the block sample time in the **Desired sample time** parameter field. The file that the From Multimedia File block references, determines the block default sample time. You can also set the sample time for this block manually. If you do not know the intended sample rate of the video, let the block inherit the sample rate from the multimedia file.

Desired sample time

Specify the block sample time. This parameter becomes available if you clear the **Inherit sample time from file** check box.

Number of times to play file

Enter a positive integer or `inf` to represent the number of times to play the file.

Output end-of-file indicator

Use this check box to determine whether the output is the last video frame or audio sample in the multimedia file. When you select this check box, a Boolean output port labeled EOF appears on the block. The output from the EOF port defaults to 1 when the last video frame or audio sample is output from the block. Otherwise, the output from the EOF port defaults to 0.

Multimedia outputs

Specify `Video and audio`, `Video only`, or `Audio only` output file type. This parameter becomes available only when a video signal has both audio and video.

Samples per audio channel

Specify number of samples per audio channel. This parameter becomes available for files containing audio.

Output color format

Specify whether you want the block to output `RGB`, `Intensity`, or `YCbCr 4:2:2` video frames. This parameter becomes available only for a signal that contains video. If you select `RGB`, use the **Image signal** parameter to specify how to output a color signal.

Image signal

Specify how to output a color video signal. If you select `One multidimensional signal`, the block outputs an M-by-N-by-P color video signal, where P is the number of color planes, at one port. If you select `Separate color signals`, additional ports appear on the block. Each port outputs one M-by-N plane of an RGB video stream. This parameter becomes available only if you set the **Image color space** parameter to `RGB` and the signal contains video.

Audio output sampling mode

Select Sample based or Frame based output. If the input to the block has 3 or more dimensions, you must select Sample based output. This parameter appears when you specify a file containing audio for the **File name** parameter.

Data Types Tab

Audio output data type

Set the data type of the audio samples output at the Audio port. This parameter becomes available only if the multimedia file contains audio. You can choose `double`, `single`, `int16`, or `uint8` types.

Video output data type

Set the data type of the video frames output at the **R**, **G**, **B**, or **Image** ports. This parameter becomes available only if the multimedia file contains video. You can choose `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, or `Inherit from file` types.

Troubleshooting

Running an Executable Outside MATLAB

To run your generated standalone executable application in Shell, you need to set your environment to the following:

Platform	Command

Mac	`setenv DYLD_LIBRARY_PATH "${DYLD_LIBRARY_PATH}:$MATLABROOT/bin/maci64" (csh/tcsh)`
	
	`export DYLD_LIBRARY_PATH=$LD_LIBRARY_PATH:$MATLABROOT/bin/maci64 (Bash)`

Linux	`setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:$MATLABROOT/bin/glnxa64 (csh/tcsh)`
	
	`export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$MATLABROOT/bin/glnxa64 (Bash)`

Windows	`set PATH=%PATH%;%MATLABROOT%\bin\win64`

Supported Data Types

For source blocks to display video data properly, double- and single-precision floating-point pixel values must be between `0` and `1`. For other data types, the pixel values must be between the minimum and maximum values supported by their data type.

Port	Supported Data Types	Supports Complex Values?
Image	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers	No
R, G, B	Same as the Image port	No
Audio	• Double-precision floating point • Single-precision floating point • 16-bit signed integers • 8-bit unsigned integers	No
Y, Cb,Cr	Same as the Image port	No

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

The executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

See Also

See Also

Blocks

`To Multimedia File`

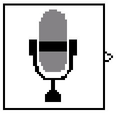
Topics

"Sample Time" (Simulink)
"How To Run a Generated Executable Outside MATLAB"

Introduced before R2006a

From Wave Device (Obsolete)

Read audio data from standard audio device in real-time (32-bit Windows operating systems only)



Library

`dspwin32`

Description

Note The From Wave Device block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the `From Audio Device` block.

The From Wave Device block reads audio data from a standard Windows audio device in real-time. It is compatible with most popular Windows hardware, including Sound Blaster cards. (Models that contain both this block and the To Wave Device block require a *duplex-capable* sound card.)

The **Use default audio device** parameter allows the block to detect and use the system's default audio hardware. This option should be selected on systems that have a single sound device installed, or when the default sound device on a multiple-device system is the desired source. In cases when the default sound device is not the desired input source, clear **Use default audio device**, and select the desired device in the **Audio device menu** parameter.

When the audio source contains two channels (stereo), the **Stereo** check box should be selected. When the audio source contains a single channel (mono), the **Stereo** check box should be cleared. For stereo input, the block's output is an M-by-2 matrix containing one frame (M consecutive samples) of audio data from each of the two channels. For mono input, the block's output is an M-by-1 matrix containing one frame (M consecutive samples) of audio data from the mono input. The frame size, M, is specified by the **Samples per frame** parameter. For M=1, the output is sample based; otherwise, the output is frame based.

The audio data is processed in uncompressed pulse code modulation (PCM) format, and should typically be sampled at one of the standard Windows audio device rates: 8000, 11025, 22050, or 44100 Hz. You can select one of these rates from the **Sample rate** parameter. To specify a different rate, select the **User-defined** option and enter a value in the **User-defined sample rate** parameter.

The **Sample Width (bits)** parameter specifies the number of bits used to represent the signal samples read by the audio device. The following settings are available:

- 8 — allocates 8 bits to each sample, allowing a resolution of 256 levels
- 16 — allocates 16 bits to each sample, allowing a resolution of 65536 levels
- 24 — allocates 24 bits to each sample, allowing a resolution of 16777216 levels (only for use with 24-bit audio devices)

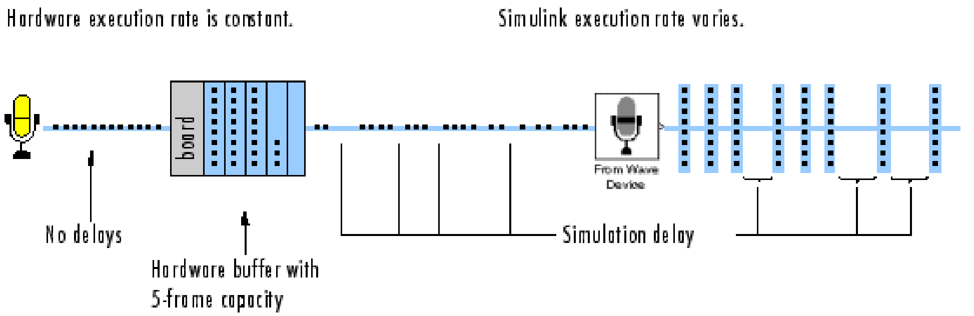
Higher sample width settings require more memory but yield better fidelity. The output from the block is independent of the **Sample width (bits)** setting. The output data type is determined by the **Data type** parameter setting.

Buffering

Since the audio device accepts real-time audio input, Simulink software must read a continuous stream of data from the device throughout the simulation. Delays in reading data from the audio hardware can result in hardware errors or distortion of the signal. This means that the From Wave Device block must read data from the audio hardware as quickly as the hardware itself acquires the signal. However, the block often *cannot* match the throughput rate of the audio hardware, especially when the simulation is running from within Simulink rather than as generated code. (Simulink operations are generally slower than comparable hardware operations, and execution speed routinely varies during the simulation as the host operating system services other processes.) The block must therefore rely on a buffering strategy to ensure that signal data can be read on schedule without losing samples.

At the start of the simulation, the audio device begins writing the input data to a (hardware) buffer with a capacity of T_b seconds. The From Wave Device block immediately begins pulling the earliest samples off the buffer (first in, first out) and collecting them in length-M frames for output. As the audio device continues to append inputs to the bottom of the buffer, the From Wave Device block continues to pull inputs off the top of the buffer at the best possible rate.

The following figure shows an audio signal being acquired and output with a frame size of 8 samples. The buffer of the sound board is approaching its five-frame capacity at the instant shown, which means that the hardware is adding samples to the buffer more rapidly than the block is pulling them off. (If the signal sample rate was 8 kHz, this small buffer could hold approximately 0.005 second of data.



When the simulation throughput rate is higher than the hardware throughput rate, the buffer remains empty throughout the simulation. If necessary, the From Wave Device block simply waits for new samples to become available on the buffer (the block does not interpolate between samples). More typically, the simulation throughput rate is lower than the hardware throughput rate, and the buffer tends to fill over the duration of the simulation.

Troubleshooting

When the buffer size is too small in relation to the simulation throughput rate, the buffer might fill before the entire length of signal is processed. This usually results in a device error or undesired device output. When this problem occurs, you can choose to either increase the buffer size or the simulation throughput rate:

- *Increase the buffer size*

  The **Queue duration** parameter specifies the duration of signal, T_b (in real-time seconds), that can be buffered in hardware during the simulation. Equivalently, this is the maximum length of time that the block's data acquisition can lag the hardware's data acquisition. The number of frames buffered is approximately

$$\frac{T_b F_s}{M}$$

where F_s is the sample rate of the signal and M is the number of samples per frame. The required buffer size for a given signal depends on the signal length, the frame size, and the speed of the simulation. Note that increasing the buffer size might increase model latency.

- *Increase the simulation throughput rate*

  Two useful methods for improving simulation throughput rates are increasing the signal frame size and compiling the simulation into native code:

  - Increase frame sizes (and convert sample-based signals to frame-based signals) throughout the model to reduce the amount of block-to-block communication overhead. This can drastically increase throughput rates in many cases. However, larger frame sizes generally result in greater model latency due to initial buffering operations.
  - Generate executable code with Simulink Coder. Native code runs much faster than Simulink, and should provide rates adequate for real-time audio processing.

More general ways to improve throughput rates include simplifying the model, and running the simulation on a faster PC processor. See “Delay and Latency” and “Performance” (Simulink) for other ideas on improving simulation performance.

Parameters

Sample rate (Hz)

The sample rate of the audio data to be acquired. Select one of the standard Windows rates or the `User-defined` option.

User-defined sample rate (Hz)

The (nonstandard) sample rate of the audio data to be acquired.

Sample width (bits)

The number of bits used to represent each signal sample.

Stereo

Specifies stereo (two-channel) inputs when selected, mono (one-channel) inputs when cleared. Stereo output is M-by-2; mono output is M-by-1.

Samples per frame

The number of audio samples in each successive output frame, *M*. When the value of this parameter is 1, the block outputs a sample-based signal.

Queue duration (seconds)

The length of signal (in seconds) to buffer to the hardware at the start of the simulation.

Use default audio device

Reads audio input from the system's default audio device when selected. Clear to enable the **Audio device ID** parameter and select a device.

Audio device

The name of the audio device from which to read the audio output (lists the names of the installed audio device drivers). Select **Use default audio device** when the system has only a single audio card installed.

Data type

The data type of the output: double-precision, single-precision, signed 16-bit integer, or unsigned 8-bit integer.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- 16-bit signed integer
- 8-bit unsigned integer

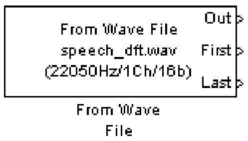
See Also

From Wave File (Obsolete)	DSP System Toolbox
To Wave Device (Obsolete)	DSP System Toolbox
`audiorecorder`	MATLAB

Introduced in R2008b

From Wave File (Obsolete)

Read audio data from Microsoft Wave (.wav) file



Library

dspwin32

Description

Note: The From Wave File block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the `From Multimedia File` block.

The From Wave File block streams audio data from a Microsoft® Wave (.wav) file and generates a signal with one of the data types and amplitude ranges in the following table.

Note: AVI files are the only supported file type for non-Windows platforms.

Output Data Type	Output Amplitude Range
double	±1
single	±1
int16	-32768 to 32767 (-2^{15} to $2^{15} - 1$)
uint8	0 to 255

The audio data must be in uncompressed pulse code modulation (PCM) format.

```
y = wavread('filename')      % Equivalent MATLAB code
```

The block supports 8-, 16-, 24-, and 32-bit Microsoft Wave (`.wav`) files.

The **File name** parameter can specify an absolute or relative path to the file. When the file is on the MATLAB path or in the current folder (the folder returned by typing `pwd` at the MATLAB command line), you need only specify the file name. You do not need to specify the `.wav` extension.

Note: The From Wave File block does not support .wav file names that contain a `+` character. To read .wav files that have a `+` character in the file name, use the `From Multimedia File` block.

For an audio file containing C channels, the block's output is an M-by-C matrix containing one frame (M consecutive samples) of audio data from each channel. The frame size, M, is specified by the **Samples per output frame** parameter. For M=1, the output is sample based; otherwise, the output is frame based.

The output frame period, T_{fo}, is

$$T_{fo} = \frac{M}{F_s}$$

where F_s is the data sample rate in Hz.

To reduce the required number of file accesses, the block acquires L consecutive samples from the file during each access, where L is specified by the **Minimum number of samples for each read from file** parameter ($L \geq M$). For $L < M$, the block instead acquires M consecutive samples during each access. Larger values of L result in fewer file accesses, which reduces run-time overhead.

Use the **Data type** parameter to specify the data type of the block's output. Your choices are `double`, `single`, `uint8`, or `int16`.

Select the **Loop** check box if you want to play the file more than once. Then, enter the number of times to play the file. The number you enter must be a positive integer or `inf`.

Use the **Number of times to play file** parameter to enter the number of times to play the file. The number you enter must be a positive integer or `inf`, to play the file until you stop the simulation.

The **Samples restart** parameter determines whether the samples from the audio file repeat immediately or repeat at the beginning of the next frame output from the output port. When you select `immediately after last sample`, the samples repeat immediately. When you select `at beginning of next frame`, the frame containing the last sample value from the audio file is zero padded until the frame is filled. The block then places the first sample of the audio file in the first position of the next output frame.

Use the **Output start-of-file indicator** parameter to determine when the first audio sample in the file is output from the block. When you select this check box, a Boolean output port labeled SOF appears on the block. The output from the SOF port is 1 when the first audio sample in the file is output from the block. Otherwise, the output from the SOF port is 0.

Use the **Output end-of-file indicator** parameter to determine when the last audio sample in the file is output from the block. When you select this check box, a Boolean output port labeled EOF appears on the block. The output from the EOF port is 1 when the last audio sample in the file is output from the block. Otherwise, the output from the EOF port is 0.

The block icon shows the name, sample rate (in Hz), number of channels (1 or 2), and sample width (in bits) of the data in the specified audio file. All sample rates are supported; the sample width must be either 8, 16, 24, or 32 bits.

Parameters

File name

Enter the path and name of the file to read. Paths can be relative or absolute.

Note: The From Wave File block does not support .wav file names that contain a + character. To read .wav files that have a + character in the file name, use the `From Multimedia File` block.

Samples per output frame

Enter the number of samples in each output frame, *M*. When the value of this parameter is `1`, the block outputs a sample-based signal.

Minimum number of samples for each read from file

Enter the number of consecutive samples to acquire from the file with each file access, *L*.

Data type

Select the output data type: `double`, `single`, `uint8`, or `int16`. The data type setting determines the output's amplitude range.

Loop

Select this check box if you want to play the file more than once.

Number of times to play file

Enter the number of times you want to play the file.

Samples restart

Select `immediately after last sample` to repeat the audio file immediately. Select `at beginning of next frame` to place the first sample of the audio file in the first position of the next output frame.

Output start-of-file indicator

Use this check box to determine whether the output contains the first audio sample in the file.

Output end-of-file indicator

Use this check box to determine whether the output contains the last audio sample in the file.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- 16-bit signed integer
- 8-bit unsigned integer

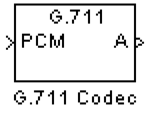
See Also

`From Audio Device`	DSP System Toolbox
`Signal From Workspace`	DSP System Toolbox
`To Multimedia File`	DSP System Toolbox

Introduced in R2010a

G711 Codec

Quantize narrowband speech input signals



Library

Quantizers

`dspquant2`

Description

The G711 Codec block is a logarithmic scalar quantizer designed for narrowband speech. Narrowband speech is defined as a voice signal with an analog bandwidth of 4 kHz and a Nyquist sampling frequency of 8 kHz. The block quantizes a narrowband speech input signal so that it can be transmitted using only 8-bits. The G711 Codec block has three modes of operation: encoding, decoding, and conversion. You can choose the block's mode of operation by setting the **Mode** parameter.

If, for the **Mode** parameter, you choose `Encode PCM to A-law`, the block assumes that the linear PCM input signal has a dynamic range of 13 bits. Because the block always operates in saturation mode, it assigns any input value above $2^{12}-1$ to $2^{12}-1$ and any input value below -2^{12} to -2^{12}. The block implements an A-law quantizer on the input signal and outputs A-law index values. When you choose `Encode PCM to mu-law`, the block assumes that the linear PCM input signal has a dynamic range of 14 bits. Because the block always operates in saturation mode, it assigns any input value above $2^{13}-1$ to $2^{13}-1$ and any input value below -2^{13} to -2^{13}. The block implements a mu-law quantizer on the input signal and outputs mu-law index values.

If, for the **Mode** parameter, you choose `Decode A-law to PCM`, the block decodes the input A-law index values into quantized output values using an A-law lookup table.

When you choose `Decode mu-law to PCM`, the block decodes the input mu-law index values into quantized output values using a mu-law lookup table.

If, for the **Mode** parameter, you choose `Convert A-law to mu-law`, the block converts the input A-law index values to mu-law index values. When you choose `Convert mu-law to A-law`, the block converts the input mu-law index values to A-law index values.

Note Set the **Mode** parameter to `Convert A-law to mu-law` or `Convert mu-law to A-law` only when the input to the block is A-law or mu-law index values.

If, for the **Mode** parameter, you choose `Encode PCM to A-law` or `Encode PCM to mu-law`, the **Overflow diagnostic** parameter appears on the block parameters dialog box. Use this parameter to determine the behavior of the block when overflow occurs. The following options are available:

- `Ignore` — Proceed with the computation and do not issue a warning message.
- `Warning` — Display a warning message in the MATLAB Command Window, and continue the simulation.
- `Error` — Display an error dialog box and terminate the simulation.

Note Like all diagnostic parameters on the Configuration Parameters dialog box, **Overflow diagnostic** parameter is set to `Ignore` in the code generated for this block by Simulink Coder code generation software.

Parameters

Mode

- When you choose `Encode PCM to A-law`, the block implements an A-law encoder.
- When you choose `Encode PCM to mu-law`, the block implements a mu-law encoder.
- When you choose `Decode A-law to PCM`, the block decodes the input index values into quantized output values using an A-law lookup table.
- When you choose `Decode mu-law to PCM`, the block decodes the input index values into quantized output values using a mu-law lookup table.

- When you choose `Convert A-law to mu-law`, the block converts the input A-law index values to mu-law index values.
- When you choose `Convert mu-law to A-law`, the block converts the input mu-law index values to A-law index values.

Overflow diagnostic

Use this parameter to determine the behavior of the block when overflow occurs.

- Select `Ignore` to proceed with the computation without a warning message.
- Select `Warning` to display a warning message in the MATLAB Command Window and continue the simulation.
- Select `Error` to display an error dialog box and terminate the simulation.

This parameter is only visible if, for the **Mode** parameter, you select `Encode PCM to A-law` or `Encode PCM to mu-law`.

References

ITU-T Recommendation G.711, "Pulse Code Modulation (PCM) of Voice Frequencies," *General Aspects of Digital Transmission Systems*; *Terminal Equipments*, International Telecommunication Union (ITU), 1993.

Supported Data Types

Port	Supported Data Types
PCM	• 16-bit signed integers
A	• 8-bit unsigned integers
mu	• 8-bit unsigned integers

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox

Scalar Quantizer Design	DSP System Toolbox
Uniform Decoder	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
Vector Quantizer Decoder	DSP System Toolbox
Vector Quantizer Design	DSP System Toolbox
Vector Quantizer Encoder	DSP System Toolbox

Introduced before R2006a

Halfband Filter

Design halfband filter

Note: The Halfband Filter block has been removed from the DSP System Toolbox block library. Existing instances of the Halfband Filter block will continue to operate. To model FIR halfband decimators and interpolators, use the `FIR Halfband Decimator` and `FIR Halfband Interpolator` blocks. These blocks replace the functionality of the Halfband Filter block, when **Impulse response** is set to `FIR`, and **FilterType** is set to `Decimator` and `Interpolator`. To model IIR halfband decimators and interpolators, use the `IIR Halfband Decimator` and `IIR Halfband Interpolator` blocks. These blocks replace the functionality of the Halfband Filter block, when **Impulse response** is set to `IIR`, and **FilterType** is set to `Decimator` and `Interpolator`.

Library

Filtering / Filter Designs

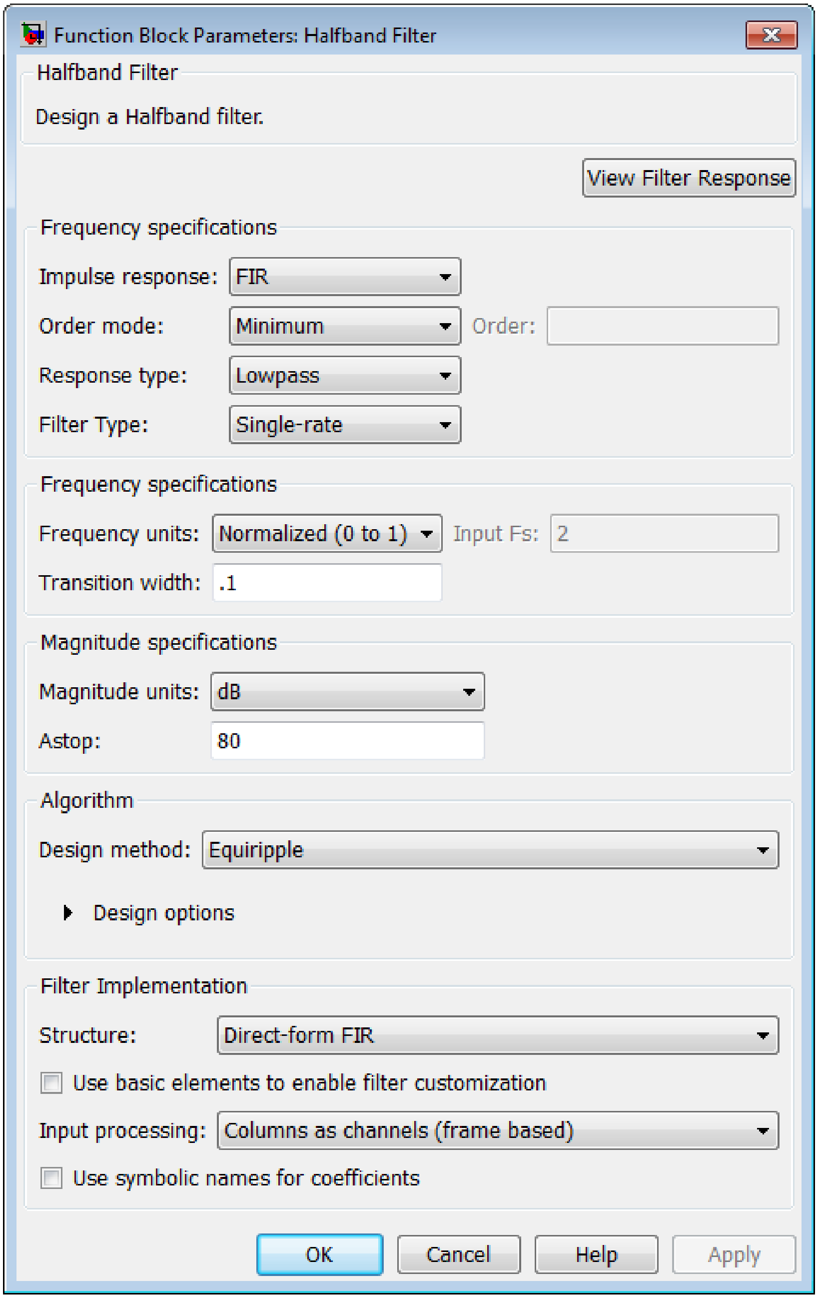
`dspfdesign`

Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See “Halfband Filter Design — Main Pane” on page 5-749 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.


Function Block Parameters: Halfband Filter
Halfband Filter
Design a Halfband filter.
View Filter Response
Frequency specifications
Impulse response:
FIR
Order mode:
Minimum
Order:
Response type:
Lowpass
Filter Type:
Single-rate
Frequency specifications
Frequency units:
Normalized (0 to 1)
Input Fs:
2
Transition width:
.1
Magnitude specifications
Magnitude units:
dB
Astop:
80
Algorithm
Design method:
Equiripple
Design options
Filter Implementation
Structure:
Direct-form FIR
Use basic elements to enable filter customization
Input processing:
Columns as channels (frame based)
Use symbolic names for coefficients
OK
Cancel
Help
Apply

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter type and order.

Impulse response

Select either `FIR` or `IIR` from the drop-down list. `FIR` is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note: The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Response type

Specify the filter response as `Lowpass` (the default) or `Highpass`.

Filter type

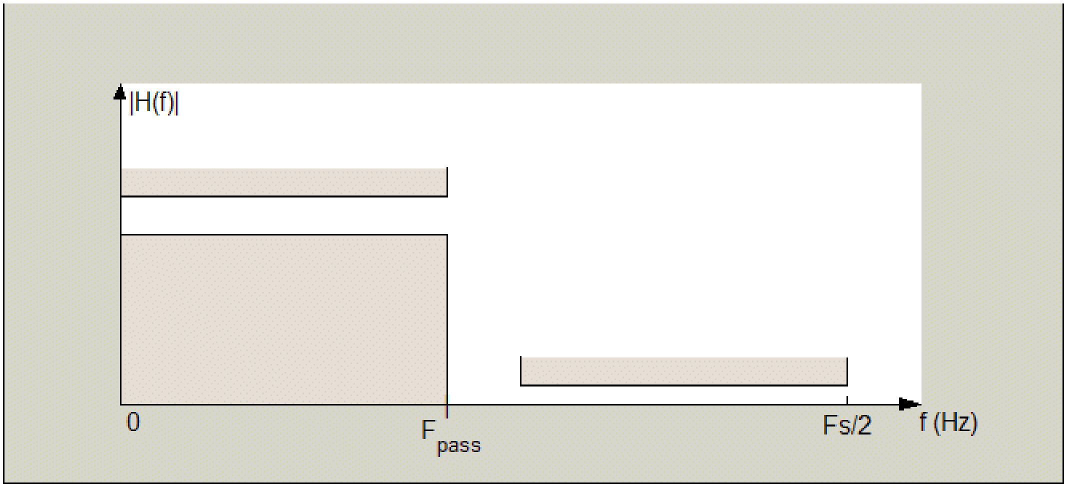
Select `Single-rate`, `Decimator`, or `Interpolator`. By default, the block specifies a single-rate filter.

Order

Enter the filter order. This option is enabled only when the **Filter order mode** is set to `Specify`.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications for a halfband lowpass filter look similar to those shown in the following figure.



In the figure, the transition region lies between the end of the passband and the start of the stopband. The width is defined explicitly by the value of **Transition width**.

Frequency constraints

When **Order mode** is `Specify`, set this parameter to `Unconstrained` or `Transition width`.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is

available when you select one of the frequency options from the **Frequency units** list.

Transition width

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude constraints

Specify `Unconstrained` (the default), or select `Stopband attenuation` to constrain the response in the stopband explicitly.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).

Astop

When **Magnitude units** is `Stopband attenuation`, enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. For FIR halfband filters, the available design options are `equiripple`, and `Kaiser window`. For IIR halfband filters, the available design options are `Butterworth`, `elliptic`, and `IIR quasi-linear phase`.

Design Options

The following design options are available for FIR halfband filters when the user specifies an equiripple design:

Minimum phase

Select the checkbox to specify a minimum-phase design.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. The block applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note: The `Inherited (this choice will be removed — see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

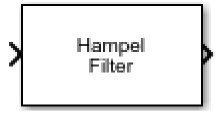
Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2006b

Hampel Filter

Filter outliers using Hampel identifier
Library: DSP System Toolbox / Filtering / Filter Designs



Description

The Hampel Filter block detects and removes the outliers of the input signal by using the Hampel identifier. The Hampel identifier is a variation of the three-sigma rule of statistics that is robust against outliers. For each sample of the input signal, the block computes the median of a window composed of the current sample and $\frac{Len-1}{2}$ adjacent samples on each side of current sample. *Len* is the window length you specify through the **Window length** parameter. The block also estimates the standard deviation of each sample about its window median by using the median absolute deviation. If a sample differs from the median by more than the threshold multiplied by the standard deviation, the filter replaces the sample with the median. For more information, see "Algorithms" on page 2-859.

Ports

Input

`Port_1` — Data input
vector | matrix

The block accepts multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$, and $n \geq 1$. m is the number of samples in each frame (channel), and n is the number of channels. The block also accepts variable-size inputs. That is, you can change the size of each input channel during simulation. However, the number of channels cannot change.

Data Types: `single` | `double`

Output

y — Filtered output

vector | matrix

The size and data type of this output matches the size and data type of the input.

This port is unnamed until you select the **Output outlier status** check box.

Data Types: `single` | `double`

`outlier` — Output the outlier status

vector | matrix

A value of `1` in this output indicates that the corresponding element in the input is an outlier. This output has the same size as the input.

This port is unnamed until you select the **Output outlier status** check box.

Dependencies

To enable this port, select the **Output outlier status** check box.

Data Types: `Boolean`

Parameters

`Window length` — Length of the sliding window

`11` (default) | positive odd scalar integer

Length of the sliding window, specified as a positive odd scalar integer. The window of finite length slides over the data, and the block computes the median and median absolute deviation of the data in the window.

`Threshold for outlier detection (standard deviations)` — Threshold

3 (default) | positive real scalar

Threshold for outlier detection, specified as a positive real scalar. For information on how this parameter is used to detect the outlier, see “Algorithms” on page 2-859.

Tunable: Yes

`Output outlier status` — Flag to output the outlier status
off (default) | on

Select this parameter to output a matrix of Boolean values that has the same size as the input. Each element in this matrix indicates whether the corresponding element in the input is an outlier. A value of `1` indicates an outlier.

`Simulate using` — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time and provides faster simulation speed than `Code generation`.
- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time and has slower simulation speed than `Interpreted execution`.

Definitions

Hampel Identifier

The Hampel identifier is a variation of the three-sigma rule of statistics that is robust against outliers.

Given a sequence $x_1, x_2, x_3, \ldots, x_n$ and a sliding window of length k, define point-to-point median and standard-deviation estimates using:

- Local median — $m_i = \text{median}\left(x_{i-k}, x_{i-k+1}, x_{i-k+2}, \ldots, x_i, \ldots, x_{i+k-2}, x_{i+k-1}, x_{i+k}\right)$
- Standard deviation — $\sigma_i = \kappa\,\text{median}\left(\left|x_{i-k} - m_i\right|, \ldots, \left|x_{i+k} - m_i\right|\right)$, where

  $$\kappa = \frac{1}{\sqrt{2}\,\text{erfc}^{-1}\,1/2} \approx 1.4826$$

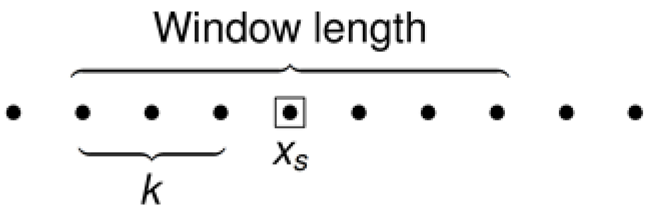
The quantity σ_i / κ is known as the median absolute deviation (MAD).

If a sample x_i is such that

$$|x_i - m_i| > n_\sigma \sigma_i$$

for a given threshold n_σ, then the Hampel identifier declares x_i an outlier and replaces it with m_i. If n_σ is 0, then the Hampel filter behaves as a regular median filter.

Algorithms



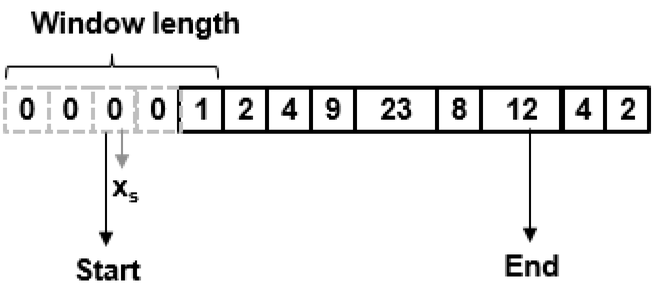
For a given sample of data, x_s, the algorithm:

- Centers the window of odd length at the current sample.
- Computes the local median, m_i, and standard deviation, σ_i, over the current window of data.
- Compares the current sample with $n_\sigma \times \sigma_i$, where n_σ is the threshold value. If $|x_s - m_i| > n_\sigma \times \sigma_i$, the filter identifies the current sample, x_s, as an outlier and replaces it with the median value, m_i.

Consider a frame of data that is passed into the Hampel filter.

1	2	4	9	23	8	12	4	2

In this example, the Hampel filter slides a window of length 5 (Len) over the data. The filter has a threshold value of 2 (n_σ). To have a complete window at the beginning of the frame, the filter algorithm prepends the frame with $Len - 1$ zeros. To compute the first sample of the output, the window centers on the $\left[\frac{Len-1}{2}+1\right]^{th}$ sample in the appended frame, the third zero in this case. The filter computes the median, median absolute deviation, and the standard deviation over the data in the local window.



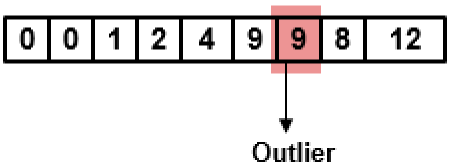
- Current sample: $x_s = 0$.
- Window of data: $win = [0\ 0\ 0\ 0\ 1]$.
- Local median: $m_i = \text{median}([0\ 0\ 0\ 0\ 1]) = 0$.
- Median absolute deviation: $mad_i = \text{median}\left(|x_{i-k} - m_i|, \ldots, |x_{i+k} - m_i|\right)$. For this window of data, $mad = \text{median}\left(|0-0|, \ldots, |1-0|\right) = 0$.
- Standard deviation: $\sigma_i = \kappa \times mad_i = 0$, where $\kappa = \frac{1}{\sqrt{2}\,\text{erfc}^{-1} 1/2} \approx 1.4826$.
- The current sample, $x_s = 0$, does not obey the relation for outlier detection.

  $$\left[|x_s - m_i| = 0\right] > \left[(n_\sigma \times \sigma_i) = 0\right]$$

  Therefore, the Hampel filter outputs the current input sample, $x_s = 0$.

Repeat this procedure for every succeeding sample until the algorithm centers the window on the $\left[End - \frac{Len-1}{2}\right]^{th}$ sample, marked as End. Because the window centered on the last $\frac{Len-1}{2}$ samples cannot be full, these samples are processed with the next frame of input data.

Here is the first output frame the Hampel filter generates:



The seventh sample of the appended input frame, 23, is an outlier. The Hampel filter replaces this sample with the median over the local window [4 9 23 8 12].

References

[1] Bodenham, Dean. “Adaptive Filtering and Change Detection for Streaming Data.” PH.D. Thesis. Imperial College, London, 2012.

[2] Liu, Hancong, Sirish Shah, and Wei Jiang. “On-line outlier detection and data cleaning.” *Computers and Chemical Engineering*. Vol. 28, March 2004, pp. 1635–1647.

[3] Suomela, Jukka. Median Filtering Is Equivalent to Sorting, 2014.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Functions

`hampel`

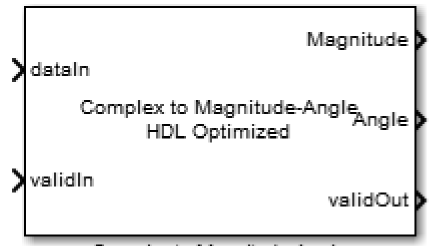
System Objects
dsp.HampelFilter | dsp.MedianFilter | dsp.MovingAverage

Blocks
`Median Filter` | `Moving Average`

Introduced in R2017a

Complex to Magnitude-Angle HDL Optimized

Compute magnitude and/or phase angle of complex signal—optimized for HDL code generation using the CORDIC algorithm



Complex to Magnitude-Angle
HDL Optimized

Library

Math Operations

`dspmathops`

Description

The Complex to Magnitude-Angle HDL Optimized block computes the magnitude and/or phase angle of a complex signal. It provides hardware-friendly control signals. The block uses a pipelined Coordinate Rotation Digital Computer (CORDIC) algorithm to achieve an efficient HDL implementation.

Signal Attributes

Port	Direction	Description	Data Type
`dataIn`	Input	Complex scalar input data.	• `fixdt()` • `int32/16/8` • `uint32/16/8`

Port	Direction	Description	Data Type
			double/single are allowed for simulation but not for HDL code generation.
validIn	Input	Indicates that the input data is valid. When validIn is high, the block captures the dataIn value.	boolean
magnitude	Output	Scalar output data.	Same as dataIn
angle	Output	Scalar output data. Optional.	Same as dataIn
validOut	Output	Indicates that the output data is valid. When the magnitude or angle output is ready, the block sets validOut high .	boolean

Parameters

Number of iterations source

Specifies the source of **Number of iterations** for the CORDIC algorithm. Select Auto to set the number of iterations to the input word length − 1. If the input is double or single, Auto sets the number of iterations to 16. Select Property to set the number of iterations from **Number of iterations**. The default is Auto.

Number of iterations

Specifies the number of CORDIC iterations the block executes. This parameter is visible only when **Number of iterations source** is set to Property. The number of iterations must be less than or equal to the input data word length − 1.

The latency of the block depends on the number of iterations performed. See “Latency” on page 2-869.

Output format

Specifies which output ports are active. You can select Magnitude, Angle, or Magnitude and angle. The default is Magnitude and angle.

Angle format

Specifies the format of the angle output. You can select Normalized or Radians. Select Normalized to return output in a fixed-point format that normalizes the angles in the range [–1,1]. For more information see “Normalized Angle Format” on page 2-869. Select `Radians` to return output as a fixed-point value between π and −π. The default format is `Normalized`.

Scale output

Scales output by the inverse of the CORDIC gain factor. The default value is selected.

Note: If you turn off output scaling, and apply the CORDIC gain elsewhere in your design, you must exclude the π/4 term. The quadrant mapping algorithm replaces the first CORDIC iteration by mapping inputs onto the angle range [0,π/4]. Therefore, the initial rotation does not contribute a gain term.

Troubleshooting

If the input is 0+0i, the output angle is undefined. The block does not implement correction logic to force the output to 0. You can ignore this output angle.

Algorithm

CORDIC Algorithm

The CORDIC algorithm is a hardware-friendly method for performing trigonometric functions. It is an iterative algorithm that approximates the solution by converging toward the ideal point. The block uses CORDIC vectoring mode to iteratively rotate the input onto the real axis.

The Givens method for rotating a complex number $x+iy$ by an angle θ is as follows. The direction of rotation, d, is +1 for counterclockwise and −1 for clockwise.

$$x_r = x\cos\theta - dy\sin\theta$$
$$y_r = y\cos\theta + dx\sin\theta$$

For a hardware implementation, factor out the $\cos\theta$ to leave a $\tan\theta$ term.

$$x_r = \cos\theta\left(x - dy\tan\theta\right)$$
$$y_r = \cos\theta\left(y + dx\tan\theta\right)$$

To rotate the vector onto the real axis, choose a series of rotations of θ_n so that $\tan\theta_n = 2^{-n}$. Remove the $\cos\theta$ term so each iterative rotation uses only shift and add operations.

$$Rx_n = x_{n-1} - d_n y_{n-1} 2^{-n}$$
$$Ry_n = y_{n-1} + d_n x_{n-1} 2^{-n}$$

Combine the missing $\cos\theta$ terms from each iteration into a constant, and apply it with a single multiplier to the result of the final rotation. The output magnitude is the scaled final value of x. The output angle, z, is the sum of the rotation angles.

$$x_r = (\cos\theta_0 \cos\theta_1 \ldots \cos\theta_n) Rx_N$$
$$z = \sum_{0}^{N} d_n \theta_n$$

Modified CORDIC Algorithm

The convergence region for the standard CORDIC rotation is ≈±99.7°. To work around this limitation, before doing any rotation, the block maps the input into the [0,π/4] range using the following algorithm.

```
if abs(x) > abs(y)
  input_mapped = [abs(x), abs(y)];
else
  input_mapped = [abs(y), abs(x)];
end
```

At each iteration, the block rotates the vector towards the real axis. The rotation is counterclockwise when y is negative, and clockwise when y is positive.

Quadrant mapping saves hardware resources and reduces latency by reducing the number of CORDIC pipeline stages by one. The CORDIC gain factor, Kn, therefore does not include the n=0, or cos(π/4) term.

$$K_n = \cos\theta_1 \ldots \cos\theta_n = \cos(26.565) \cdot \cos(14.036) \cdot \cos(7.125) \cdot \cos(3.576)$$

After the CORDIC iterations are complete, the block corrects the angle back to its original location. First it adjusts the angle to the correct side of π/4.

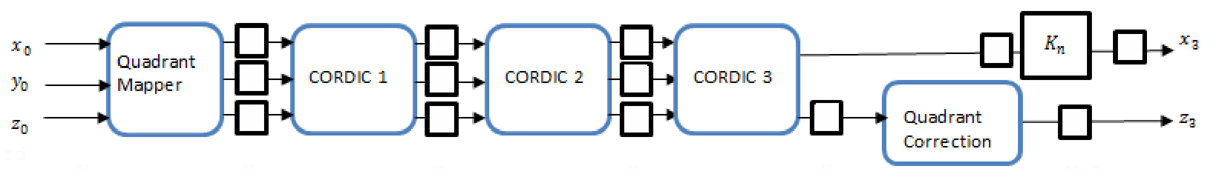
```
if abs(x) > abs(y)
  angle_unmapped = CORDIC_out;
else
  angle_unmapped = (pi/2) - CORDIC_out;
end
```

Then it flips the angle to the original quadrant.

```
if (x < 0)
  if (y < 0)
    output_angle = - pi + angle_unmapped;
  else
    output_angle = pi - angle_unmapped;
else
  if (y<0)
    output_angle = -angle_unmapped;
```

Architecture

The block generates a pipelined HDL architecture to maximize throughput. Each CORDIC iteration is done in one pipeline stage. The gain multiplier, if enabled, is implemented with Canonical Signed Digit (CSD) logic.

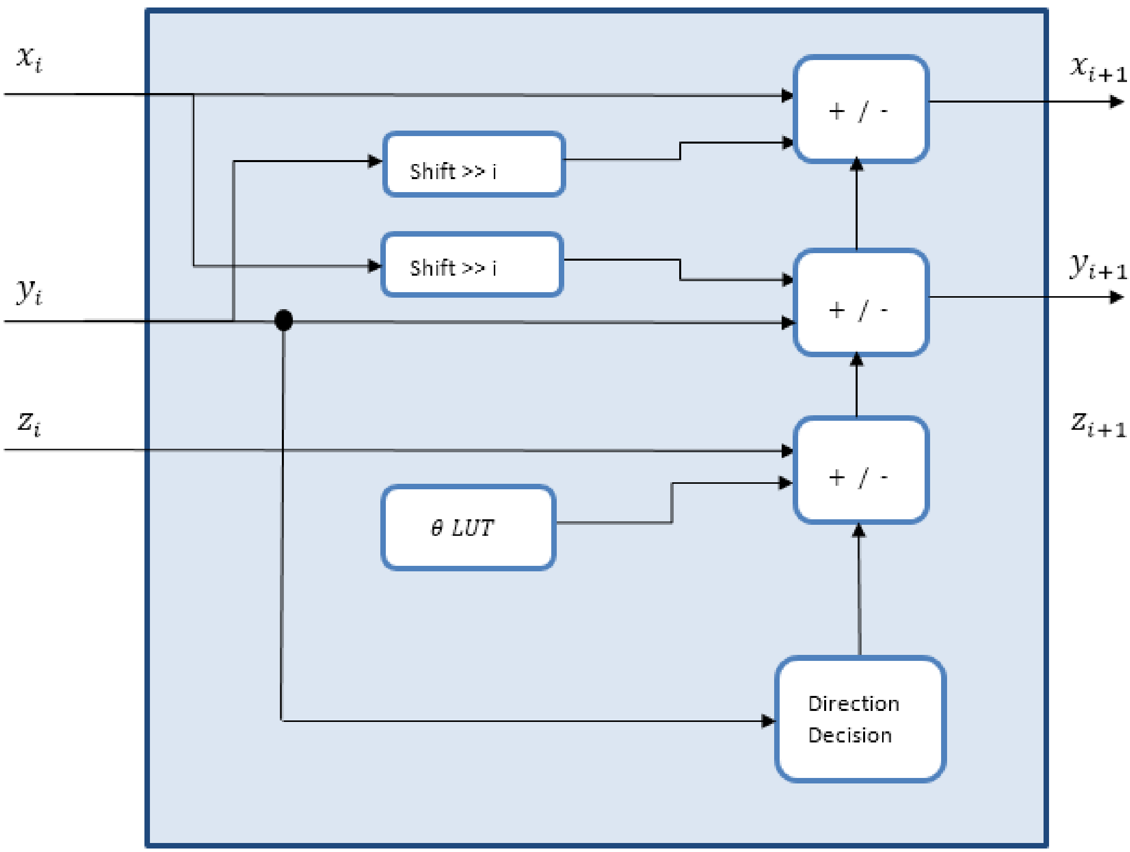


Input Word Length	Output Magnitude Word Length
fixdt(0,WL,FL)	fixdt(0,WL+2,FL)
fixdt(1,WL,FL)	fixdt(1,WL+1,FL)

Input Word Length	Output Angle Word Length	
fixdt([ ],WL,FL)	Radians	fixdt(1,WL+3,WL)

Input Word Length	Output Angle Word Length	
	Normalized	fixdt(1,WL+3,WL+2)

The CORDIC logic at each pipeline stage implements one iteration. For each pipeline stage, the shift and angle rotation are constants.

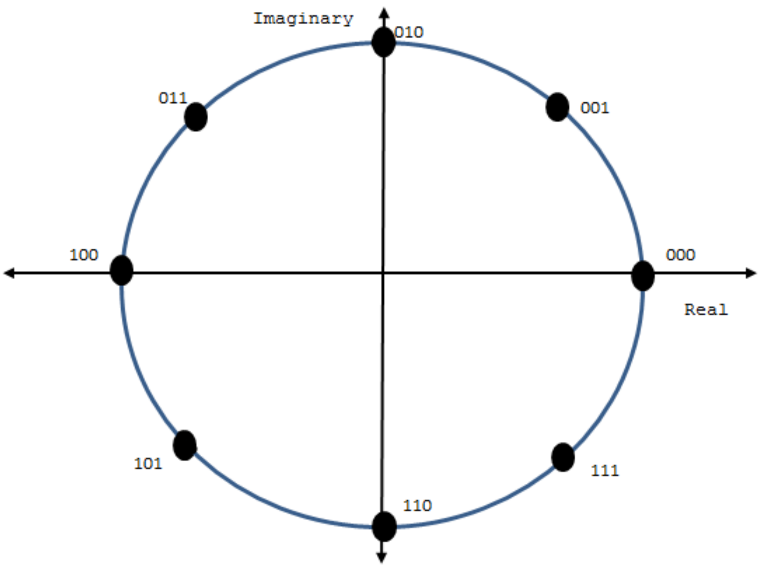


When you set **Output format** to `Magnitude`, the block does not generate HDL code for the angle accumulation and quadrant correction logic.

Normalized Angle Format

This format normalizes the fixed-point radian angle values around the unit circle. This is a more efficient use of bits than a range of [0,2π] radians. Normalized angle format also enables wraparound at 0/2π without additional detect and correct logic.

For example, representing the angle with 3 bits results in the following normalized values.



Using the mapping described in "Modified CORDIC Algorithm" on page 2-866, the block normalizes the angles across [0,π/4] and maps them to the correct octant at the end of the calculation.

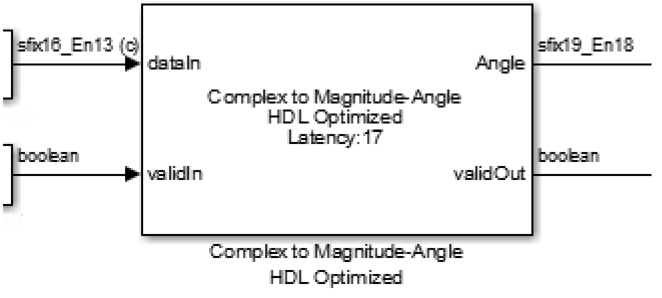
Control Signals

The `validIn` signal qualifies the input data. When the output data calculated from a valid input reaches the end of the pipeline, the block asserts the `validOut` signal.

Latency

The output is valid **Number of iterations** + 2 cycles after valid input.

When you set **Number of iterations source** to `Property`, the block shows the latency immediately. When you set **Number of iterations source** to `Auto`, the block calculates the latency based on the input port data type, and displays it when you update the model.



When you set **Number of iterations source** to `Auto`, the number of iterations is input word length − 1, and the latency is input word length + 1. If the input is `double` or `single` type, the number of iterations is 16, and the latency is 18.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Complex to Magnitude-Angle HDL Optimized in the HDL Coder documentation.

Performance

When generated HDL code for the default configuration, with output scaling disabled and `fixdt(1,16,12)` input, is synthesized into a Xilinx Virtex–6 (XC6VLX240T-1FFG1156) FPGA, the design achieves 260 MHz clock frequency. It uses the following resources.

Resource	Uses
LUT	882

Resource	Uses
FFS	792
Xilinx LogiCORE DSP48	0
Block RAM (16K)	0

Performance of the synthesized HDL code varies depending on your target and synthesis options.

See Also

See Also

Complex to Magnitude-Angle | dsp.HDLComplexToMagnitudeAngle

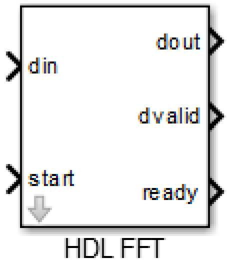
Topics

“HDL Optimized QPSK Receiver with Captured Data” (Communications System Toolbox)
“HDL Optimized QAM Transmitter and Receiver” (Communications System Toolbox)
“IEEE 802.11 WLAN - HDL Optimized Beacon Frame Receiver with Captured Data” (HDL Coder)

Introduced in R2014b

HDL Minimum Resource FFT

FFT— optimized for HDL code generation using minimum hardware resources



Library

Obsolete

`dspobs`

Description

The HDL Minimum Resource FFT block implements an FFT architecture that uses minimal hardware resources. The HDL Minimum Resource FFT block supports the Radix-2 with decimation-in-time (DIT) algorithm for FFT computation. See the FFT block for more information about this algorithm.

The results returned by the HDL Minimum Resource FFT block are bit-for-bit compatible with results returned by the FFT block. The operation of the HDL Minimum Resource FFT block differs from the FFT block, due to the requirements of hardware realization. The HDL Minimum Resource FFT block:

- Requires serial input
- Generates serial output

- Operates in burst I/O mode

The HDL Minimum Resource FFT block provides handshaking signals to support these features.

- “Block Inputs and Outputs” on page 2-873
- “Parameters” on page 2-875
- “Signal Processing with the HDL FFT Block” on page 2-877

Block Inputs and Outputs

As shown in the following figure, the HDL Minimum Resource FFT block has two input ports and three output ports. Two of these ports are for data input and output signals. The other ports are for control signals.

The input ports are:

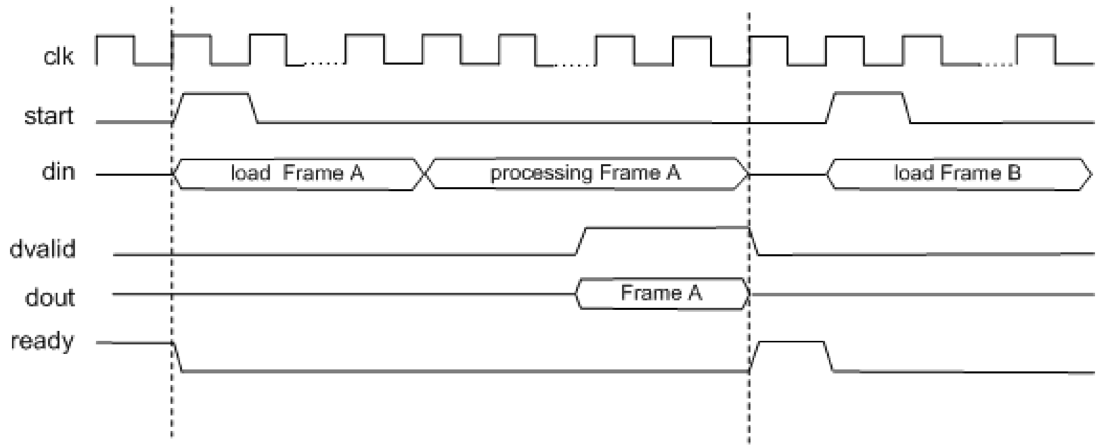
- `din`: The input data signal. A complex signal is required.
- `start`: Boolean control signal. When this signal is asserted true (1), the HDL Minimum Resource FFT block initiates processing of a data frame.

The output ports are:

- `dout`: Data output signal. The Radix-2 with DIT algorithm produces output with linear ordering.
- `dvalid`: Boolean control signal. The HDL Minimum Resource FFT block asserts this signal true (1) when a burst of valid output data is available at the `dout` port.
- `ready`: Boolean control signal. The HDL Minimum Resource FFT block asserts this signal true (1) to indicate that it is ready to process a new frame.

Configuring Control Signals

For efficient hardware deployment of the HDL Minimum Resource FFT block, the timing of the block's input and output data streams must be considered carefully. The following figure shows the timing relationships between the system clock and the `start`, `ready`, and `dvalid` signals.



When ready is asserted, the start signal (active high) triggers the block. The high cycle period of the start signal does not affect the behavior of the block.

One clock cycle after the start trigger, the block begins to load data and the ready signal is deasserted. During the interval when the block is loading, processing, and outputting data, ready is low and the start signal is ignored.

The dvalid signal is asserted high for N clock cycles (where N is the FFT length) after processing is complete. ready is asserted again after the N-point FFT outputs are sent out.

The expression Tcycle denotes the total number of clock cycles required by the HDL Minimum Resource FFT block to complete an FFT of length N. Tcycle is defined as follows:

- Where N >8

  ```
  Tcycle =  3N/2-2 + log2(N)*(N/2+3);
  ```
- Where N = 8

  ```
  Tcycle =  3N/2-1 +log2(N)*(N/2+3);
  ```

Given Tcycle, you can then define the period between assertions of the HDL Minimum Resource FFT start signal in a way that is suitable to your application. In the “Using the Minimum Resource HDL FFT” example, this period is computed and assigned to the variable startLen, as follows:

```
if (N<=8)
startLen = (ceil(Tcycle/N)+1)*N;
else
startLen = ceil(Tcycle/N)*N;
end
```

In the example model, `startLen` determines the period of a Pulse Generator that drives the HDL Minimum Resource FFT block's `start` input. These values are computed in the model's initialization function (`InitFcn`), which is defined in the **Callbacks** pane of the Simulink Model Explorer.

The HDL Minimum Resource FFT block asserts and deasserts the `ready` and `dvalid` signals automatically. These signals are routed to the model components that write to and read from the HDL Minimum Resource FFT block.

Parameters

FFT Length

Default: 8

The FFT length must be a power of 2, in the range 2^3 .. 2^{16}.

Rounding mode

Default: `Floor`

The HDL Minimum Resource FFT block supports all rounding modes of the FFT block. See also the FFT block reference section.

Overflow mode

Default: `Saturate`

The HDL Minimum Resource FFT block supports all overflow modes of the FFT block. See also the FFT block reference section.

Sine table

Default: `Same word length as input`

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values is equal to the word length minus one.

- When you select `Same word length as input`, the word length of the sine table values match that of the input to the block.
- When you select `Specify word length`, you can enter the word length of the sine table values, in bits, in the **Sine table word length** field. The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; they always saturate and round to `Nearest`.

Product output

Default: `Same as input`

Use this parameter to specify how you want to designate the product output word and fraction lengths:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the product output, in bits, in the **Product word length** and **Product fraction length** fields.

Accumulator

Default: `Same as input`

Use this parameter to specify how you want to designate the accumulator word and fraction lengths:

When you select `Same as product output`, these characteristics match those of the product output.

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the accumulator, in bits, in the **Accumulator word length** and **Accumulator fraction length** fields.

Output

Default: `Same as input`

Choose how you specify the output word length and fraction length:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the output, in bits, in the **Output word length** and **Output fraction length** fields.

Note: The HDL FFT block skips the divide-by-two operation on butterfly outputs for fixed-point signals.

Signal Processing with the HDL FFT Block

To get started with the HDL Minimum Resource FFT block, run the "Using the Minimum Resource HDL FFT" example, which is located in the HDL Coder/Signal Processing example library.

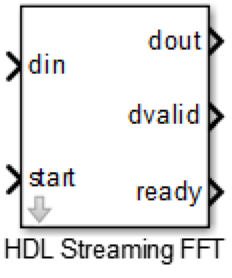
The example illustrates the use of the HDL Minimum Resource FFT block in simulation. The model includes buffering and control logic that handles serial input and output. In the example, a complex source signal is stored as a series of samples in a FIFO. Samples from the FIFO are processed serially by the HDL Minimum Resource FFT block, which emits a stream of scalar FFT data.

For comparison, the same source signal is also processed by the frame-based FFT block. The output frames from the FFT block are buffered into a FIFO and compared to the output of the HDL Minimum Resource FFT block. Examination of the results shows the outputs to be identical.

Introduced in R2014b

HDL Streaming FFT

Radix-2 FFT with decimation-in-frequency (DIF) — optimized for HDL code generation



Library

Obsolete

`dspobs`

Description

Note: The HDL Streaming FFT block will be deprecated in future releases. Use the FFT HDL Optimized block instead.

The HDL Streaming FFT block returns results identical to results returned by the Radix-2 DIF algorithm of the FFT block.

- "Block Inputs and Outputs" on page 2-879
- "Timing Description" on page 2-879
- "Parameters" on page 2-882

Block Inputs and Outputs

As shown in the following figure, the HDL Streaming FFT block has two input ports and three output ports. Two of these ports are for data input and output signals. The other ports are for control signals.

The block has the following input ports:

- `din`: The input data signal. The coder requires a complex fixed-point signal.
- `start`: Boolean control signal. When `start` asserts true (1), the HDL Streaming FFT block initiates processing of a data frame.

The block has the following output ports:

- `dout`: Data output signal.
- `dvalid`: Boolean control signal. The HDL Streaming FFT block asserts this signal true (1) when a stream of valid output data is available at the `dout` port.
- `ready`: Boolean control signal. The HDL Streaming FFT block asserts this signal true (1) to indicate that it is ready to process a new frame.

Timing Description

The HDL Streaming FFT block operates in one of two modes:

- *Continuous data streaming* mode: In this mode, the HDL Streaming FFT block expects to receive a continuous stream of data at `din`. After an initial delay, the block produces a continuous stream of data at `dout`.
- *Non-continuous data streaming* mode: In this mode, the HDL Streaming FFT block receives non-continuous bursts of streaming data at `din`. After an initial delay, the block produces non-continuous bursts of streaming data at `dout`.

The behavior of the control signals determines the timing mode of the block.

Continuous Data Streaming Timing

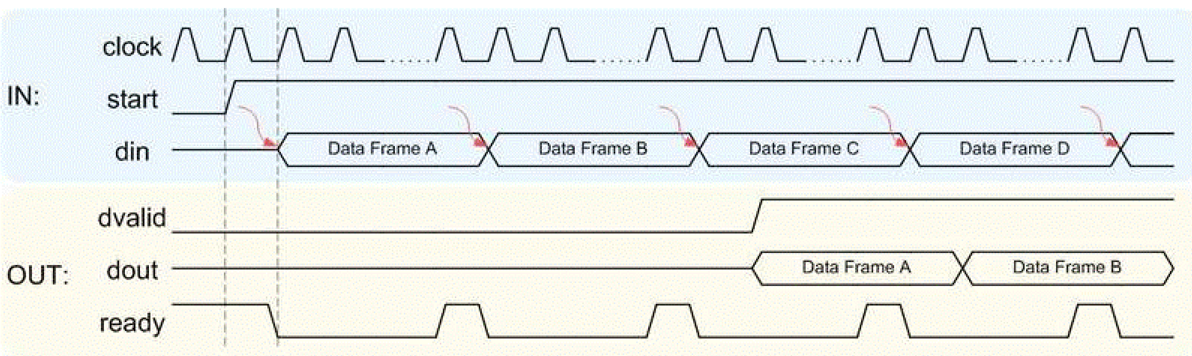
Assertion of the `start` signal (active high) triggers processing by the HDL Streaming FFT block. To initiate continuous data stream processing, assert the `start` signal in one of the following ways:

- Hold the start signal high (as shown in figure "Continuous Data Streaming With Start Signal Held High").
- Pulse the start signal every N clock cycles, where N is the FFT length (as shown in figure "Continuous Data Streaming With Pulsed Start Signal").

One clock cycle after the `start` trigger, the block begins to load data at `din`. After the first frame of streaming data, the block starts to receive the next frame of streaming data.

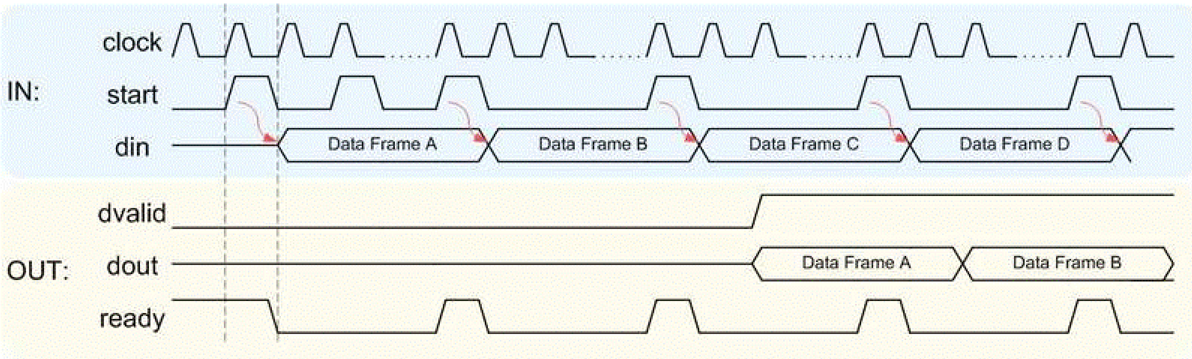
Meanwhile, the block performs the FFT calculation on the incoming data frames and outputs the results continuously at `dout`. The HDL Streaming FFT block asserts and deasserts the `ready` and `dvalid` signals automatically. The block asserts `dvalid` high whenever the output data stream is valid. The block asserts `ready` high to indicate that the block is ready to load a new data frame. When `ready` is low, the block ignores the `start` signal.

The following figures illustrate continuous data streaming. Each data frame corresponds to a stream of N input data values, where N is the FFT length.



Continuous Data Streaming With Start Signal Held High

Note: The `start` signal can be a single cycle pulse; it need not be held high for the entire data frame. When processing for a frame begins, further pulses on `start` do not affect processing of that frame. However, a `start` pulse must occur at the beginning of each data frame.



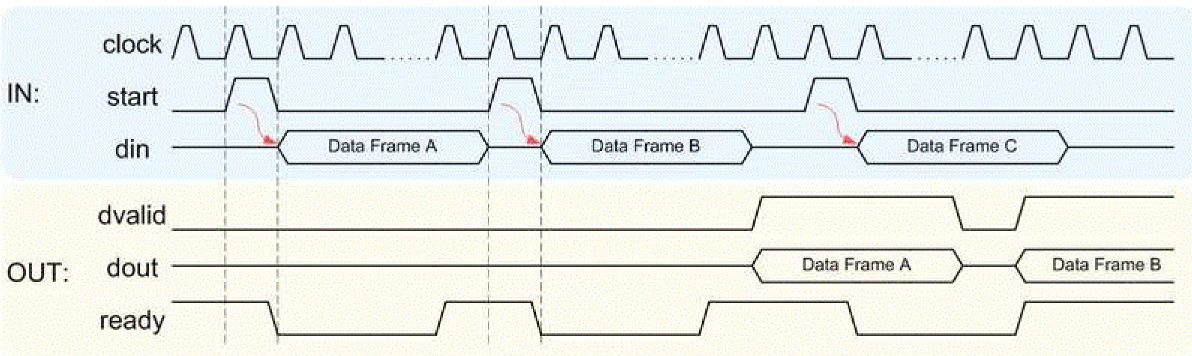
Continuous Data Streaming With Pulsed Start Signal

Non-Continuous Data Streaming Timing

In this mode, the HDL Streaming FFT block receives continuous bursts of streaming data at `din`. After an initial delay, the block produces non-continuous bursts of streaming data at `dout`. Breaks occur between data frames when the following condition exist:

- The `start` signal does not assert every N clock cycles (where N is the FFT length)
- The `start` signal is not continuously held high.

Non-continuous data streaming mode allows you more flexibility in determining the intervals between input data streams.

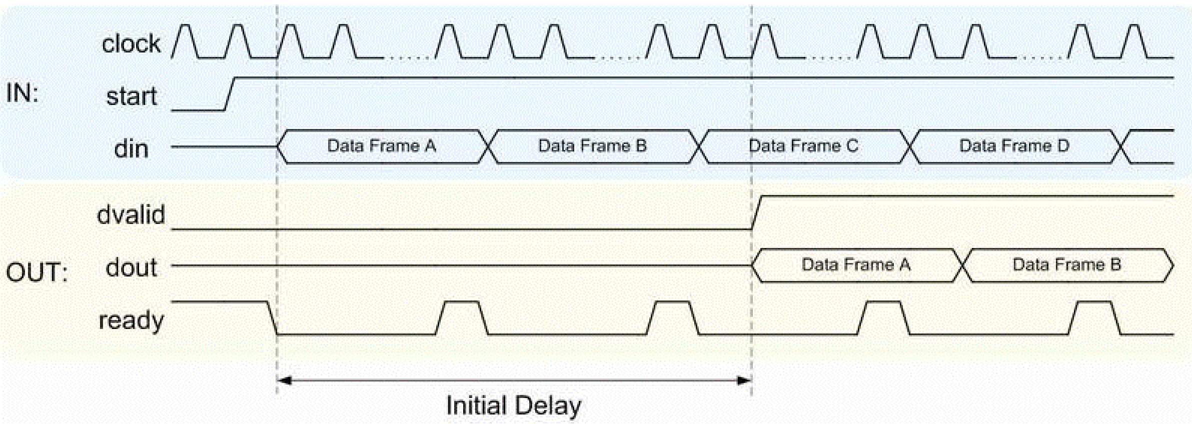


Initial Delay

The initial delay of the HDL Streaming FFT block is the interval between the following times:

- The time the block begins to receive the first frame of input data
- The time the block asserts `dvalid` and produces the first valid output data.

The initial delay represents the time the block uses to load a data frame, calculate the FFT, and output the beginning of the first output frame. The following figure illustrates the initial delay.



If you select the block option **Display computed initial delay on mask**, the block icon displays the initial delay. The display represents the delay time as Z^{-n}, where n is the delay time in samples.

Parameters

FFT Length

Default: 1024

The FFT length must be a power of 2, in the range 2^3 to 2^{16}.

Rounding mode

Default: `Floor`

The HDL Streaming FFT block supports all rounding modes of the FFT block. See also the FFT block reference.

Overflow mode

Default: `Wrap`

The HDL Streaming FFT block supports all overflow modes of the FFT block. See also the FFT block reference.

Sine table

Default: `Same word length as input`

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values is equal to the word length minus one.

- When you select `Same word length as input`, the word lengths of the sine table values match the word lengths of the block inputs.
- When you select `Specify word length`, you can enter the word length of the sine table values, in bits, in the **Sine table word length** field. The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters. They always saturate and round to `Nearest`.

Product output

Default: `Same as input`

Use this parameter to specify how you want to designate the product output word and fraction lengths:

- When you select `Same as input`, these characteristics match the characteristics of the input to the block.
- `Binary point scaling`: Enter the word length and the fraction length of the product output, in bits, in the **Product word length** and **Product fraction length** fields.

Accumulator

Default: `Same as input`

Use this parameter to specify how you want to designate the accumulator word and fraction lengths:

When you select Same as product output, these characteristics match the characteristics of the product output.

- When you select Same as input, these characteristics match the characteristics of the input to the block.
- Binary point scaling: Enter the word length and the fraction length of the accumulator, in bits, in the **Accumulator word length** and **Accumulator fraction length** fields.

Output

Default: Same as input

Choose how you specify the output word length and fraction length:

- Same as input: these characteristics match the characteristics of the input to the block.
- Binary point scaling: lets you enter the word length and fraction length of the output, in bits, in the **Output word length** and **Output fraction length** fields.

Output in bit-reversed order

Default: Off

- On: The output data stream is in bit-reversed order.
- Off: The output data stream is in natural order.

For more information about the effects of bit reversal, see "Linear and Bit-Reversed Output Order".

Display computed initial delay on mask

Default: Off

- On: The block icon displays the initial delay as Z^{-n}, where n is the delay time in samples.
- Off: The block icon does not display the initial delay.

Note: **Sine table**, **Product output**, **Accumulator**, and **Output** do not support:

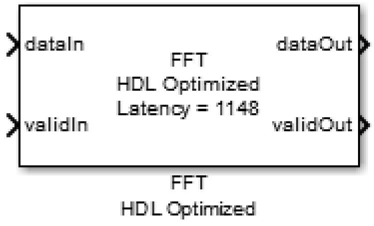
- Inherit via internal rule

- `Slope and bias scaling`

Introduced in R2014b

FFT HDL Optimized

Fast Fourier transform—optimized for HDL code generation



Library

Transforms

`dspxfrm3`

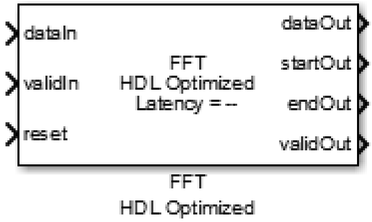
Description

The FFT HDL Optimized block implements a pipelined Radix 2^2 FFT algorithm that provides hardware speed and area optimization for streaming data applications. The block accepts scalar or vector input of real or complex data, provides hardware-friendly control signals, and has optional output frame control signals. You can achieve giga-sample-per-second (GSPS) throughput using vector input.

The FFT HDL Optimized block replaces the `HDL Streaming FFT` block.

Signal Attributes

This FFT HDL Optimized block icon shows all optional ports.



Port	Direction	Description	Data Type
`dataIn`	Input	Scalar or column vector of real or complex input data. The vector size must be a power of 2 between 1 and 64, that is not greater than the FFT length.	• `fixdt()` • `int64/32/16/8` • `uint64/32/16/8` `double/single` are allowed for simulation but not for HDL code generation.
`validIn`	Input	Indicates that the input data is valid. When `validIn` is `true`, the block captures the value on `dataIn`.	`boolean`
`reset`	Input	Optional. Reset internal state. When `reset` is `true`, the block stops the current calculation and clears all internal state. The block begins fresh calculations when `reset` is `false` and `validIn` starts a new frame.	`boolean`
`dataOut`	Output	Frequency channel output data. The output order is bit reversed by default.	Same as `dataIn`. If scaling is disabled, the word length grows by 2 bits per stage.
`validOut`	Output	Indicates that the output data is valid. The block sets `validOut` to `true` with each valid sample on `dataOut`.	`boolean`
`startOut`	Output	Optional. When this port is enabled, the block sets `startOut`	`boolean`

Port	Direction	Description	Data Type
		to `true`during the first valid cycle of a frame of output data.	
`endOut`	Output	Optional. When this port is enabled, the block sets `endOut` to `true` during the last valid cycle of a frame of output data.	`boolean`

Parameters

Main

Architecture

`Streaming Radix 2^2` (default) — An efficient architecture that has lower latency and uses fewer resources than the previous versions.

`Streaming Radix 2` — Versions before R2016a use this architecture. Use only for backward-compatibility purposes.

Complex Multiplication

Select the HDL implementation of complex multipliers. Each multiplication is implemented with either 3 multipliers and 5 adders, or 4 multipliers and 2 adders. Which option is faster or smaller depends on your synthesis tool and target device. This option applies only when you set **Architecture** to `Streaming Radix 2^2`.

Output in bit-reversed order

When you select this check box, the output elements are bit reversed relative to the input order. Clear the check box to output elements in linear order. By default, the check box is selected. The FFT algorithm calculates output in the reverse order to the input, and performs an extra reversal operation when required to provide output in the same order as the input. For vector data, input and output data must be in opposite orders so check only one of **Output in bit-reversed order** or **Input in bit-reversed order**. For more information, see “Linear and Bit-Reversed Output Order”.

Input in bit-reversed order

When you select this check box, the block expects input data in bit-reversed order. By default, the check box is clear and input is expected in linear order. The FFT algorithm calculates output in the reverse order to the input, and performs an extra reversal operation when required to provide output in the same order as the input. For vector data, input and output data must be in opposite orders so check only one of **Output in bit-reversed order** or **Input in bit-reversed order**. For more information, see “Linear and Bit-Reversed Output Order”.

Divide butterfly outputs by two

When you select this check box, the block implements an overall $1/N$ scale factor by scaling the result of each pipeline stage by 2. This adjustment keeps the output of the FFT in the same amplitude range as its input. If scaling is disabled, the block avoids overflow by increasing the word length by 1 bit at each stage. By default, the check box is not selected.

FFT length

Specify the number of data points used for one FFT calculation. The default value is 1024. The FFT length must be a power of 2 between 2^3 and 2^{16} for HDL code generation.

Data Types

Rounding Method

The default rounding method for internal fixed point calculations is `Floor`. The FFT block uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input is single or double type. Each stage rounds after the twiddle factor multiplication but before the butterflies.

Control Ports

Enable reset input port

When you select this check box, the `reset` port is present on the block icon. When `reset` is `true`, the block stops the current calculation and clears all internal state. The block begins fresh calculations when `reset` is `false` and `validIn` starts a new frame. By default, the check box is not selected.

Enable start output port

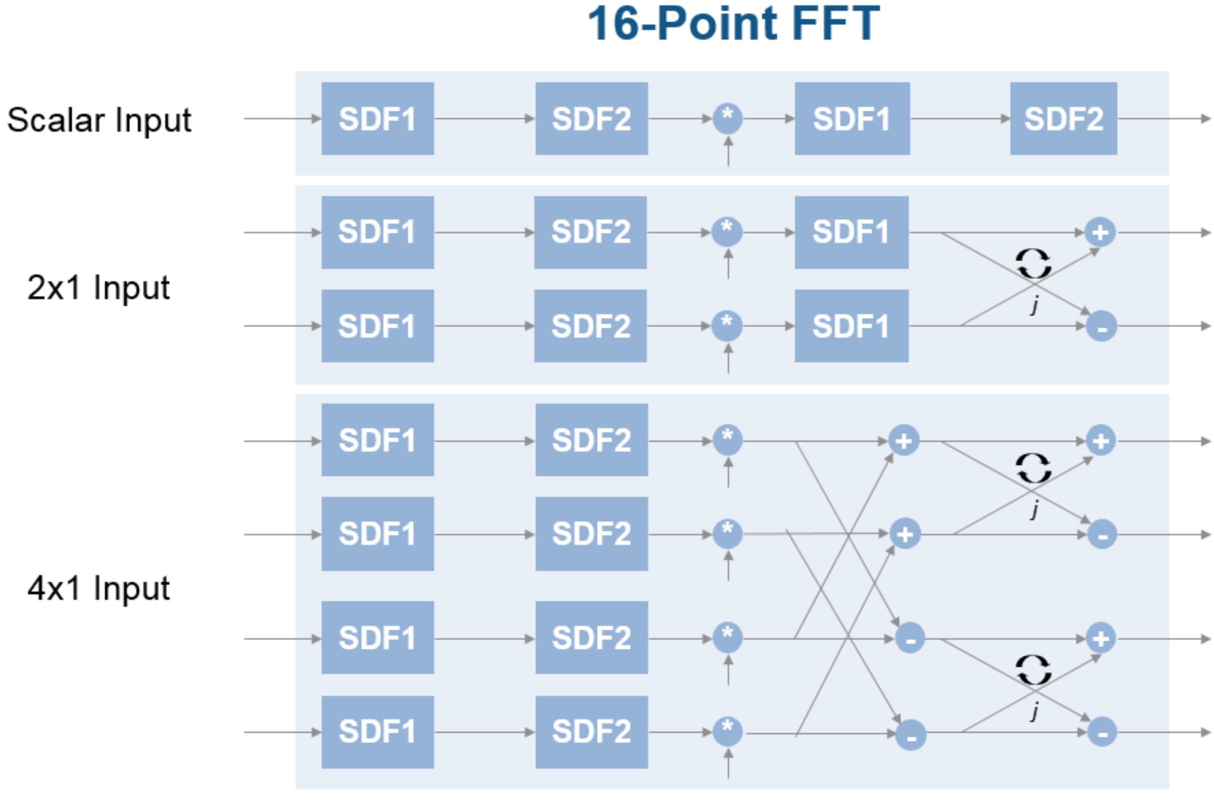
When you select this check box, the `startOut` port is present on the block icon, and this output signal is asserted (`true`) for the first cycle of an output frame. By default, the check box is not selected.

Enable end output port

When you select this check box, the `endOut` port is present on the block icon, and this output signal is asserted (`true`) for the last cycle of an output frame. By default, the check box is not selected.

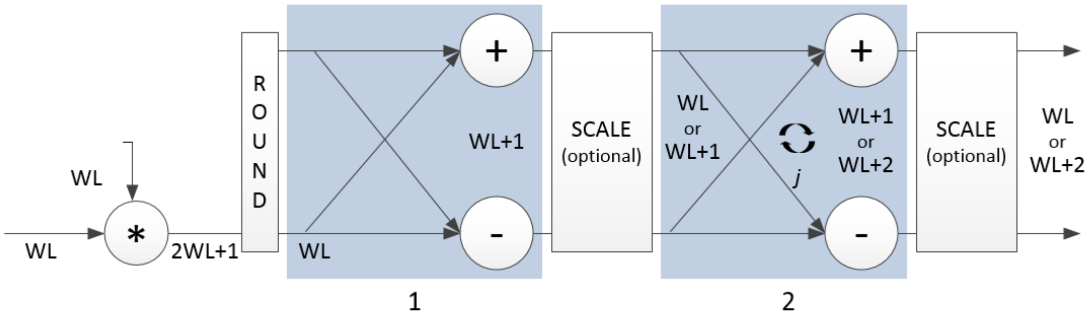
Algorithm

The Radix 2^2 architecture saves resources by factoring and grouping the FFT equation. The architecture has $\log_4(N)$ stages. Each stage contains two single-path delay feedback (SDF) butterflies with memory controllers. If you use vector input, the results of twiddle factor multiplication are shared across multiple data points, without memory accesses.



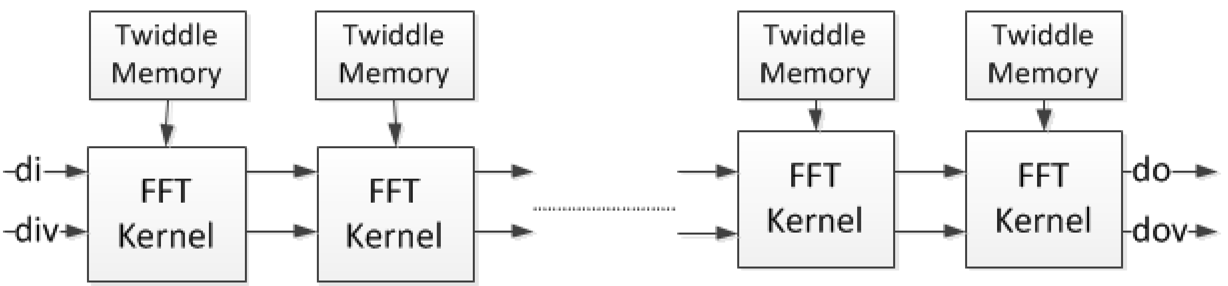
The first SDF stage is a regular butterfly, and the second stage multiplies by –*j* by swapping the real and imaginary parts of the input, and swapping the imaginary parts of the output. Each stage rounds the result of the twiddle factor multiplication to the input word length. This result is stored in memory for use by later stages. The twiddle factors have the same bit width as the input data. They use 2 integer bits and the remainder are fractional bits.

If you select **Divide butterfly outputs by two**, the block scales the result of each butterfly stage by 2. Scaling at each stage avoids overflow, keeps the word length the same as the input, and results in an overall scale factor of $1/N$. If scaling is disabled, the block avoids overflow by increasing the word length by 1 bit at each stage. The diagram shows the butterflies and internal word lengths of each stage, not including the memory.

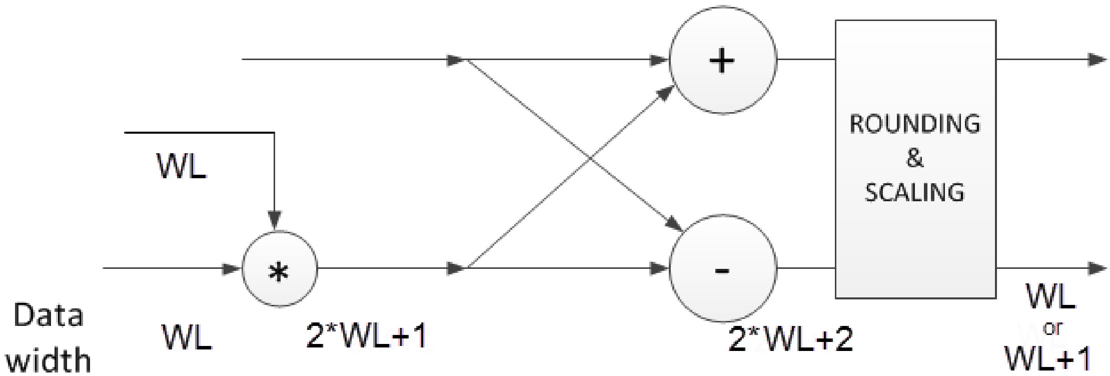


Radix 2 Architecture

In the Radix 2 architecture has $\log_2(N)$ pipeline stages. Each pipeline stage, or kernel, contains memory, a controller, and a complex Radix-2 butterfly.



Using decimation in time, each kernel multiplies by the twiddle factor and then adds samples in a butterfly. The kernel architecture minimizes the number of multipliers and adders. Within each kernel, data is processed in full precision. The kernel rounds to the output data width after the butterfly sum.



Control Signals

Input data is processed only when `validIn` is high. Output data is only valid when `validOut` is high.

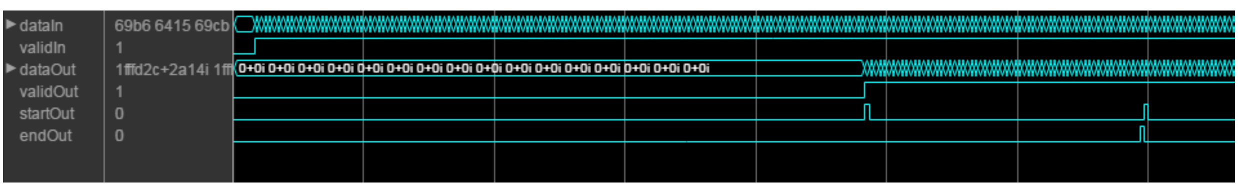
The block provides an optional reset port. When `reset` is high, the block stops the current calculation and clears all internal state. The block begins fresh calculations when `reset` is low and `validIn` starts a new frame.

Timing Diagram

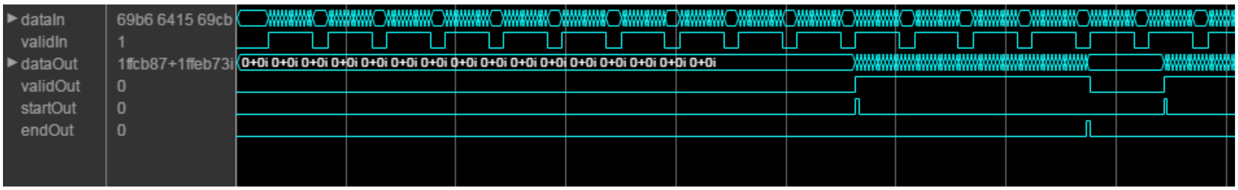
This diagram shows `validIn` and `validOut` signals for contiguous scalar input data and an FFT length of 1024, and a vector size of 16.

The diagram also shows the optional `startOut` and `endOut` signals that indicate frame boundaries. If you enable `startOut`, it pulses for one cycle with the first `validOut` of the frame. If you enable `endOut`, it pulses for one cycle with the last `validOut` of the frame.

If you apply continuous input frames, the output will also be continuous, after the initial latency.



The `validIn` signal can be noncontiguous. Data accompanied by a `validIn` signal is stored until a frame is filled, and output in a contiguous frame of *N* (FFT length) cycles. This diagram shows noncontiguous input and contiguous output for an FFT length of 512 and a vector size of 16.



Latency

The latency varies with the FFT length and input vector size. After you update the model, the latency is displayed on the block icon. The displayed latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see FFT HDL Optimized in the HDL Coder documentation.

Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a Xilinx Virtex-6 (XC6VLX75T-1FF484) FPGA. The three examples in the tables have this configuration:

- FFT length (default) — 1024
- Complex multiplication — 4 multipliers, 2 adders
- Output scaling — Enabled
- 16-bit complex input data
- Minimize clock enables (HDL Coder parameter)

Performance of the synthesized HDL code varies with your target and synthesis options. For instance, natural order output uses more RAM than bit-reversed output, and real input uses less RAM than complex input.

For a scalar input Radix 2^2 configuration, the design achieves 326 MHz clock frequency. The latency is 1116 cycles. The design uses these resources.

Resource	Uses
LUT	4597
FFS	5353
Xilinx LogiCORE DSP48	12
Block RAM (16K)	6

When you vectorize the same Radix 2^2 implementation to process two 16-bit input samples in parallel, the design achieves 316 MHz clock frequency. The latency is 600 cycles. The design uses these resources.

Resource	Uses
LUT	7653
FFS	9322
Xilinx LogiCORE DSP48	24
Block RAM (16K)	8

The Radix 2 implementation is supported with scalar input data only. The Radix 2 design achieves 295 MHz clock frequency. The latency is 1148 cycles. The design uses these resources.

Resource	Uses
LUT	4060
FFS	5160
Xilinx LogiCORE DSP48	16
Block RAM (16K)	6

References

[1] Algnabi, Y.S, F.A. Aldaamee, R. Teymourzadeh, M. Othman, and M.S. Islam. “Novel architecture of pipeline Radix 2^2 SDF FFT Based on digit-slicing technique.”

10th IEEE International Conference on Semiconductor Electronics (ICSE). 2012, pp. 470–474.

See Also

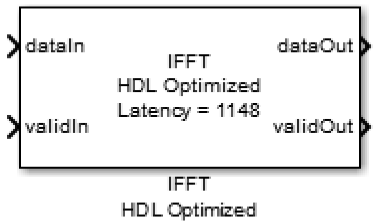
See Also

dsp.HDLFFT | `Channelizer HDL Optimized` | `FFT` | `IFFT HDL Optimized`

Introduced in R2014a

IFFT HDL Optimized

Inverse fast Fourier transform—optimized for HDL code generation



Library

Transforms

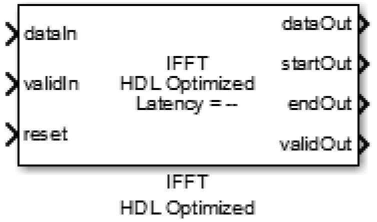
`dspxfrm3`

Description

The IFFT HDL Optimized block implements a pipelined Radix 2^2 IFFT algorithm that provides hardware speed and area optimization for streaming data applications. The block accepts scalar or vector input of real or complex data, provides hardware-friendly control signals, and has optional output frame control signals. You can achieve giga-sample-per-second (GSPS) throughput using vector input.

Signal Attributes

This IFFT HDL Optimized block icon shows all optional ports available.



Port	Direction	Description	Data Type
`dataIn`	Input	Scalar or column vector of real or complex input data. The vector size must be a power of 2 between 1 and 64, that is not greater than the FFT length.	• `fixdt()` • `int64/32/16/8` • `uint64/32/16/8` `double/single` are allowed for simulation but not for HDL code generation.
`validIn`	Input	Indicates that the input data is valid. When `validIn` is `true`, the block captures the value on `dataIn`.	`boolean`
`reset`	Input	Optional. Reset internal state. When `reset` is `true`, the block stops the current calculation and clears all internal state. The block begins fresh calculations when `reset` is `false` and `validIn` starts a new frame.	`boolean`
`dataOut`	Output	Frequency channel output data. The output order is bit reversed by default.	Same as `dataIn`. If scaling is disabled, the word length grows by 2 bits per stage.
`validOut`	Output	Indicates that the output data is valid. The block sets `validOut` to `true` with each valid sample on `dataOut`.	`boolean`
`startOut`	Output	Optional. When this port is enabled, the block sets `startOut`	`boolean`

Port	Direction	Description	Data Type
		to `true`during the first valid cycle of a frame of output data.	
`endOut`	Output	Optional. When this port is enabled, the block sets `endOut` to `true` during the last valid cycle of a frame of output data.	`boolean`

Parameters

Main

Architecture

`Streaming Radix 2^2` (default) — An efficient architecture that has lower latency and uses fewer resources than the previous versions.

`Streaming Radix 2` — Versions before R2016a use this architecture. Use only for backward-compatibility purposes.

Complex Multiplication

Select the HDL implementation of complex multipliers. Each multiplication is implemented with either 3 multipliers and 5 adders, or 4 multipliers and 2 adders. Which option is faster or smaller depends on your synthesis tool and target device. This option applies only if you set **Architecture** to `Streaming Radix 2^2`.

Output in bit-reversed order

When you select this check box, the output elements are bit reversed relative to the input order. Clear the check box to output elements in linear order. By default, the check box is selected. The FFT algorithm calculates output in the reverse order to the input, and performs an extra reversal operation when required to provide output in the same order as the input. For vector data, input and output data must be in opposite orders so check only one of **Output in bit-reversed order** or **Input in bit-reversed order**. For more information, see "Linear and Bit-Reversed Output Order".

Input in bit-reversed order

When you select this check box, the block expects input data in bit-reversed order. By default, the check box is clear and input is expected in linear order. The FFT algorithm calculates output in the reverse order to the input, and performs an extra reversal operation when required to provide output in the same order as the input. For vector data, input and output data must be in opposite orders so check only one of **Output in bit-reversed order** or **Input in bit-reversed order**. For more information, see "Linear and Bit-Reversed Output Order".

Divide butterfly outputs by two

When you select this check box, the block implements an overall $1/N$ scale factor by scaling the result of each pipeline stage by 2. This adjustment keeps the output of the IFFT in the same amplitude range as its input. If scaling is disabled, the block avoids overflow by increasing the word length by 1 bit at each stage. By default, the check box is selected.

FFT length

Specify the number of data points used for one FFT calculation. The default value is 1024. The FFT length must be a power of 2 between 2^3 and 2^{16} for HDL code generation.

Data Types

Rounding Method

The default rounding method for internal fixed point calculations is `Floor`. The IFFT block uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input is single or double type. Each stage rounds after the twiddle factor multiplication but before the butterflies.

Control Ports

Enable reset input port

When you select this check box, the `reset` port is present on the block icon. When `reset` is `true`, the block stops the current calculation and clears all internal state. The block begins fresh calculations when `reset` is `false` and `validIn` starts a new frame. By default, the check box is not selected.

Enable start output port

When you select this check box, the `startOut` port is present on the block icon, and this output signal is asserted (`true`) for the first cycle of an output frame. By default, the check box is not selected.

Enable end output port

When you select this check box, the endOut port is present on the block icon, and this output signal is asserted (true) for the last cycle of an output frame. By default, the check box is not selected.

Algorithm

The IFFT HDL Optimized block implements a pipelined Radix 2^2 (or Radix 2) algorithm. It implements the reverse function of the FFT HDL Optimized block. For more information, refer to the FFT HDL Optimized block page.

Control Signals

Input data is processed only when validIn is high. Output data is only valid when validOut is high.

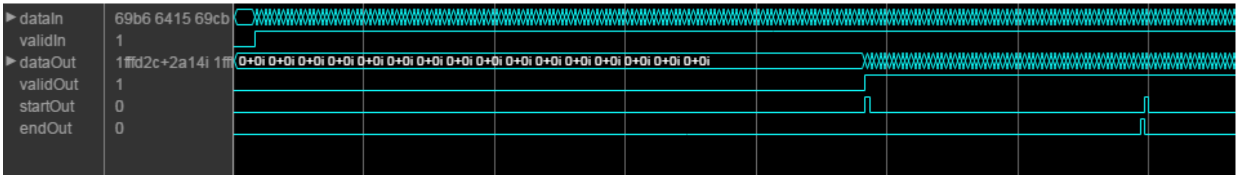
The block provides an optional reset port. When reset is high, the block stops the current calculation and clears all internal state. The block begins fresh calculations when reset is low and validIn starts a new frame.

Timing Diagram

This diagram shows validIn and validOut signals for contiguous scalar input data and an FFT length of 1024, and a vector size of 16.

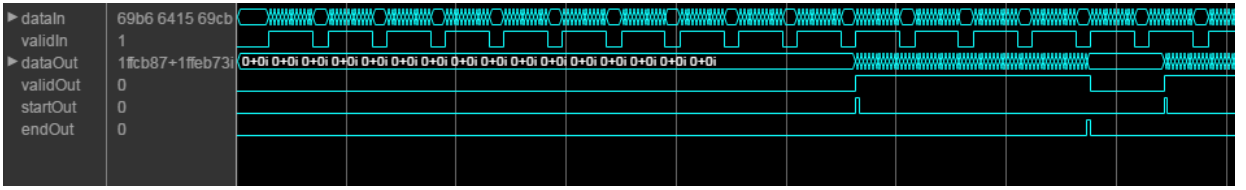
The diagram also shows the optional startOut and endOut signals that indicate frame boundaries. If you enable startOut, it pulses for one cycle with the first validOut of the frame. If you enable endOut, it pulses for one cycle with the last validOut of the frame.

If you apply continuous input frames, the output will also be continuous, after the initial latency.



The validIn signal can be noncontiguous. Data accompanied by a validIn signal is stored until a frame is filled, and output in a contiguous frame of N (FFT length) cycles.

This diagram shows noncontiguous input and contiguous output for an FFT length of 512 and a vector size of 16.



Latency

The latency varies with the FFT length and input vector size. After you update the model, the latency is displayed on the block icon. The displayed latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see IFFT HDL Optimized in the HDL Coder documentation.

Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a Xilinx Virtex-6 (XC6VLX75T-1FF484) FPGA. The three examples in the tables have this configuration:

- FFT length (default) — 1024
- Complex multiplication — 4 multipliers, 2 adders
- Output scaling — Enabled
- 16-bit complex input data
- Minimize clock enables (HDL Coder parameter)

Performance of the synthesized HDL code varies with your target and synthesis options. For instance, natural order output uses more RAM than bit-reversed output, and real input uses less RAM than complex input.

For a scalar input Radix 2^2 configuration, the design achieves 326 MHz clock frequency. The latency is 1116 cycles. The design uses these resources.

Resource	Uses
LUT	4597
FFS	5353
Xilinx LogiCORE DSP48	12
Block RAM (16K)	6

When you vectorize the same Radix 2^2 implementation to process two 16-bit input samples in parallel, the design achieves 316 MHz clock frequency. The latency is 600 cycles. The design uses these resources.

Resource	Uses
LUT	7653
FFS	9322
Xilinx LogiCORE DSP48	24
Block RAM (16K)	8

The Radix 2 implementation is supported with scalar input data only. The Radix 2 design achieves 295 MHz clock frequency. The latency is 1148 cycles. The design uses these resources.

Resource	Uses
LUT	4060
FFS	5160
Xilinx LogiCORE DSP48	16
Block RAM (16K)	6

See Also

See Also

dsp.HDLIFFT | `FFT HDL Optimized` | `IFFT`

Introduced in R2014a

Highpass Filter

Design FIR or IIR highpass filter

Library

Filtering / Filter Designs

`dspfdesign`

Description

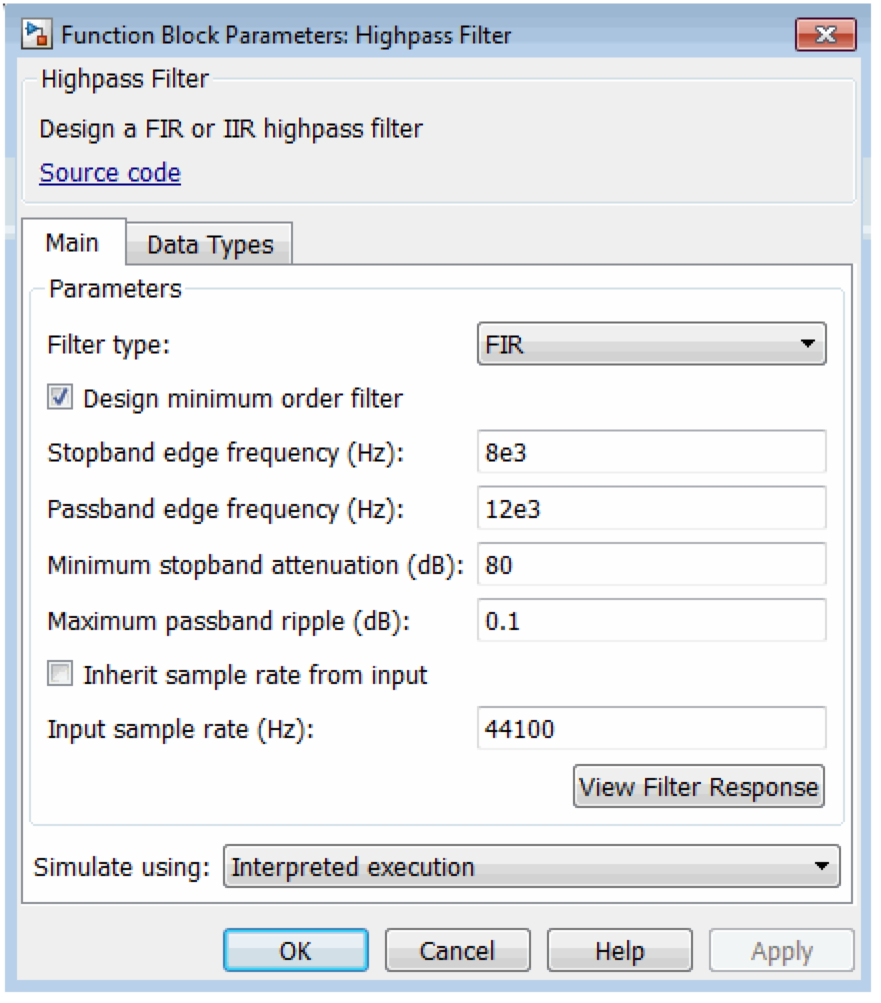
The Highpass Filter block independently filters each channel of the input signal over time using the given design specifications.

You can set the **Filter type** parameter of the block to design to implement the block as an FIR or IIR highpass filter. The block designs the filter based on the parameters specified in the block dialog box.

The input signal can be a real- or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The block supports fixed-point operations, HDL code generation, and ARM Cortex code generation. To learn more about ARM Cortex code generation, see “ARM Cortex-M and ARM Cortex-A Optimization”

Dialog Box

Main Tab



Filter type

Type of highpass filter.

- `FIR` (default) — Design an FIR highpass filter.
- `IIR` — Design an IIR highpass filter.

Design minimum order filter

When you select this check box, the block designs a filter with the minimum order that the specified passband, stopband frequency, passband ripple, and stopband attenuation. Set these specifications using the corresponding parameters. When you clear this check box, specify the order of the filter in **Filter order**.

By default, this check box is selected.

Filter order

Filter order of highpass filter, specified as a positive scalar integer. You can specify a filter order only when the **Design minimum order filter** check box is not cleared. The default is `50`.

Stopband edge frequency (Hz)

Stopband edge frequency of the highpass filter, specified as a real positive scalar in Hz. The value of the stopband edge frequency in Hz must be less than the passband frequency. You can specify the stopband edge frequency only when the **Design minimum order filter** check box is selected. The default is `8e3`.

Passband edge frequency (Hz)

Passband edge frequency of the highpass filter, specified as a real positive scalar in Hz. The passband edge frequency must be less than half the value of the **Input sample rate (Hz)** and greater than the value of the **Stopband edge frequency (Hz)**. The default is `12e3`.

Minimum stopband attenuation (dB)

Minimum attenuation in the stopband, specified as a real positive scalar in dB. The default is `80`.

Maximum passband ripple (dB)

Maximum ripple of the filter response in the passband, specified as a real positive scalar in dB. The default is `0.1`.

Inherit sample rate from input

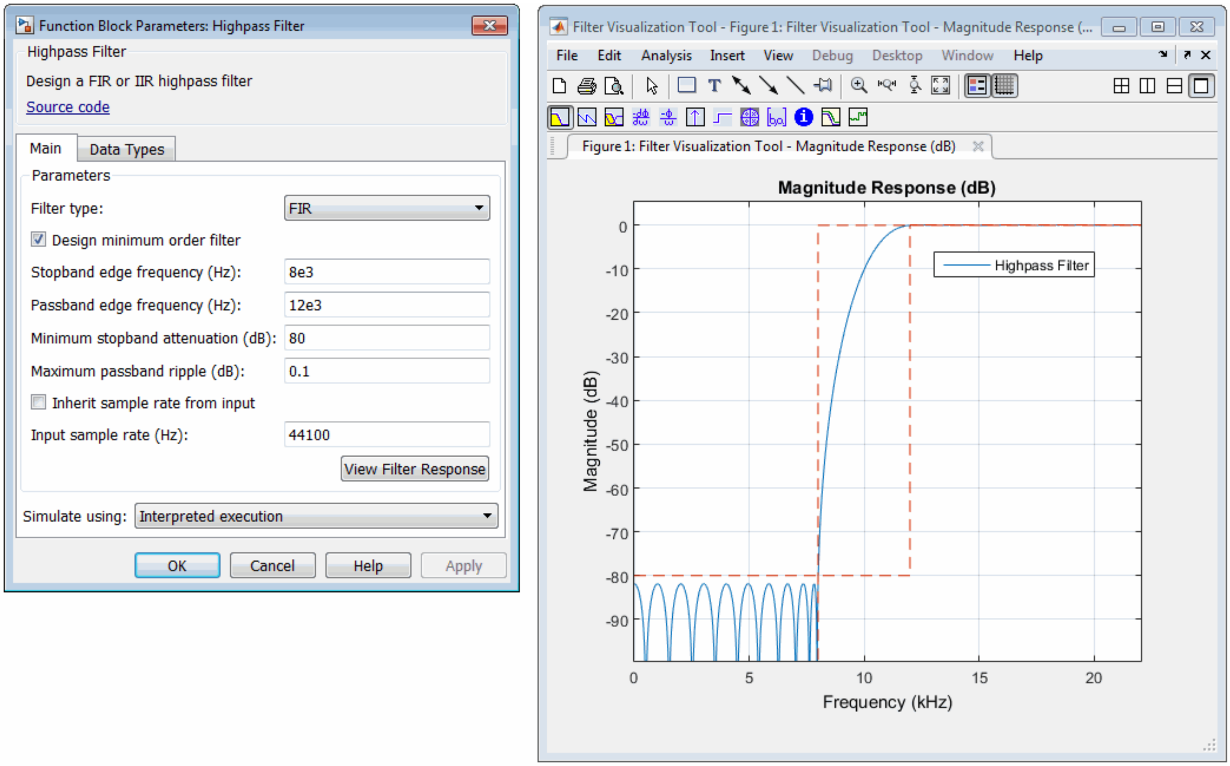
When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**.

Input sample rate (Hz)

Input sample rate, specified as a scalar in Hz. The default is `44100`.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the Highpass Filter. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

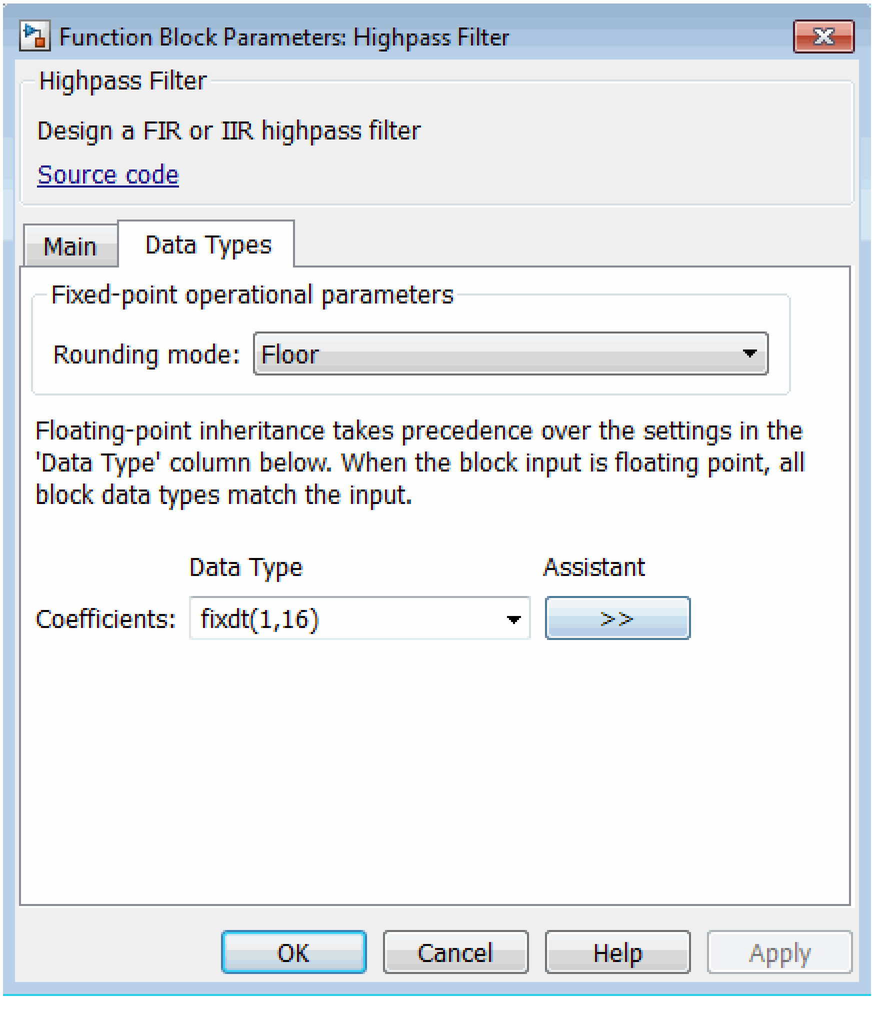
- `Interpreted execution` (default)

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

- `Code generation`

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

Data Types Tab



Rounding mode

Rounding method for the output fixed-point operations. The rounding methods are `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, and `Zero`. The default is `Floor`.

Coefficients

Fixed-point data type of the coefficients, specified as one of the following:

- `fixdt(1,16)` (default) — Signed fixed-point data type of word length `16`, with binary point scaling. The block determines the fraction length automatically from the coefficient values in such a way that the coefficients occupy maximum representable range without overflowing.
- `fixdt(1,16,0)` — Signed fixed-point data type of word length `16` and fraction length, `0`. You can change the fraction length to any other integer value.
- `<data type expression>` — Specify the data type using an expression that evaluates to a data type object, for example, numeric type (`fixdt([ ],16, 15)`). Specify the sign mode of this data type as `[ ]` or `true`.
- `Refresh Data Type` — Refresh to the default data type.

Click the **Show data type assistant** button >> to display the data type assistant, which helps you set the stage input parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed or unsigned)
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only)

See Also

dsp.HighpassFilter	DSP System Toolbox
dsp.LowpassFilter	DSP System Toolbox
`Lowpass Filter`	DSP System Toolbox

Algorithm

This block brings the capabilities of the dsp.HighpassFilter System object to the Simulink environment.

For information on the algorithms used by this block, see the Algorithms on page 4-959 section of `dsp.HighpassFilter`.

Introduced in R2015b

Highpass Filter (Obsolete)

Design highpass filter

Note: The Highpass Filter (Obsolete) block has been replaced by the `Highpass Filter` block. Existing instances of the Highpass Filter (Obsolete) block will continue to operate. For new models, use the Highpass Filter block.

Library

Filtering / Filter Designs

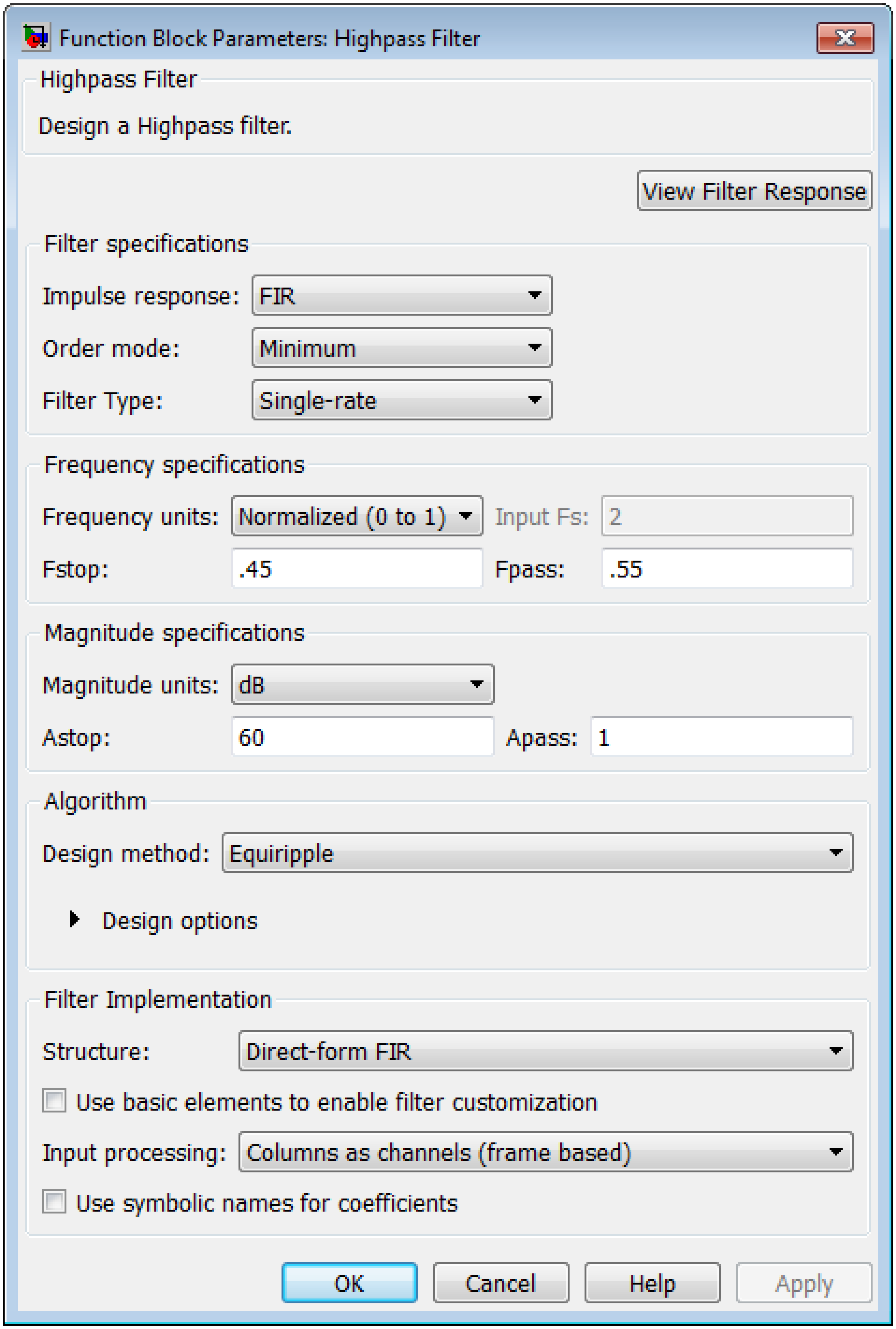
`dspfdesign`

Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See “Highpass Filter Design — Main Pane” on page 5-754 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.


Function Block Parameters: Highpass Filter
Highpass Filter
Design a Highpass filter.
View Filter Response
Filter specifications
Impulse response: FIR
Order mode: Minimum
Filter Type: Single-rate
Frequency specifications
Frequency units: Normalized (0 to 1)
Input Fs: 2
Fstop: .45
Fpass: .55
Magnitude specifications
Magnitude units: dB
Astop: 60
Apass: 1
Algorithm
Design method: Equiripple
Design options
Filter Implementation
Structure: Direct-form FIR
Use basic elements to enable filter customization
Input processing: Columns as channels (frame based)
Use symbolic names for coefficients
OK
Cancel
Help
Apply

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop-down list. `FIR` is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note: The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select `Minimum` (the default) or `Specify`. Selecting `Specify` enables the **Order** option so you can enter the filter order. When you set the **Impulse response** to `IIR`, you can specify different numerator and denominator orders. To specify a different denominator order, you must select the **Denominator order** check box.

Order

Enter the filter order. This option is enabled only if you set the **Order mode** to `Specify`.

Denominator order

Select this check box to specify a different denominator order. This option is enabled only if you set the **Impulse response** to `IIR` and the **Order mode** to `Specify`.

Filter type

Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting Sample-rate converter activates both factors.

Decimation Factor

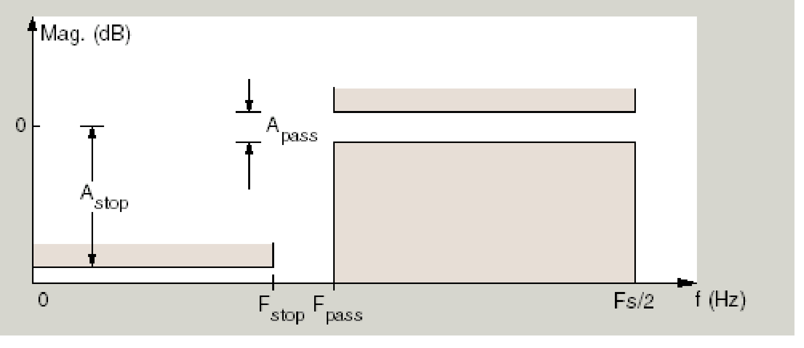
Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to Interpolator or Sample-rate converter. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, the region between specification values Fstop and Fpass represents the transition region where the filter response is not constrained.

Frequency constraints

When **Order mode** is `Specify`, select the filter features that the block uses to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Stopband edge and passband edge` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- `Passband edge` — Define the filter by specifying the edge of the passband.
- `Stopband edge` — Define the filter by specifying the edge of the stopband.
- `Stopband edge and 3 dB point` — For IIR filters, define the filter by specifying the frequency of the 3 dB point in the filter response and the edge of the stopband.
- `3 dB point` — Define the filter response by specifying the location of the 3 dB point. The 3 dB point is the frequency for the point three decibels below the passband value.
- `3 dB point and passband edge` — For IIR filters, define the filter by specifying the frequency of the 3 dB point in the filter response and the edge of the passband.
- `6 dB point` — For FIR filters, define the filter response by specifying the location of the 6 dB point. The 6 dB point is the frequency for the point six decibels below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fstop

Enter the frequency at the edge of the end of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fpass

Enter the frequency at the edge of the start of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

F3dB

When **Frequency constraints** is `3 dB point`, `Stopband edge and 3 dB point`, or `3 dB point and passband edge`, specify the frequency of the 3 dB point. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

F6dB

When **Frequency constraints** is `6 dB point`, specify the frequency of the 6 dB point. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude constraints

This option is only available when you specify the order of your filter design. Depending on the setting of the **Frequency constraints** parameter, some combination of the following options will be available for the **Magnitude constraints** parameter: `Unconstrained`, `Passband ripple`, `Passband ripple and stopband attenuation` or `Stopband attenuation`.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is `Equiripple`.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Specify the phase constraint of the filter as `Linear`, `Maximum`, or `Minimum`.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note: The `Inherited (this choice will be removed — see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2006b

Hilbert Filter

Design Hilbert filter

Library

Filtering / Filter Designs

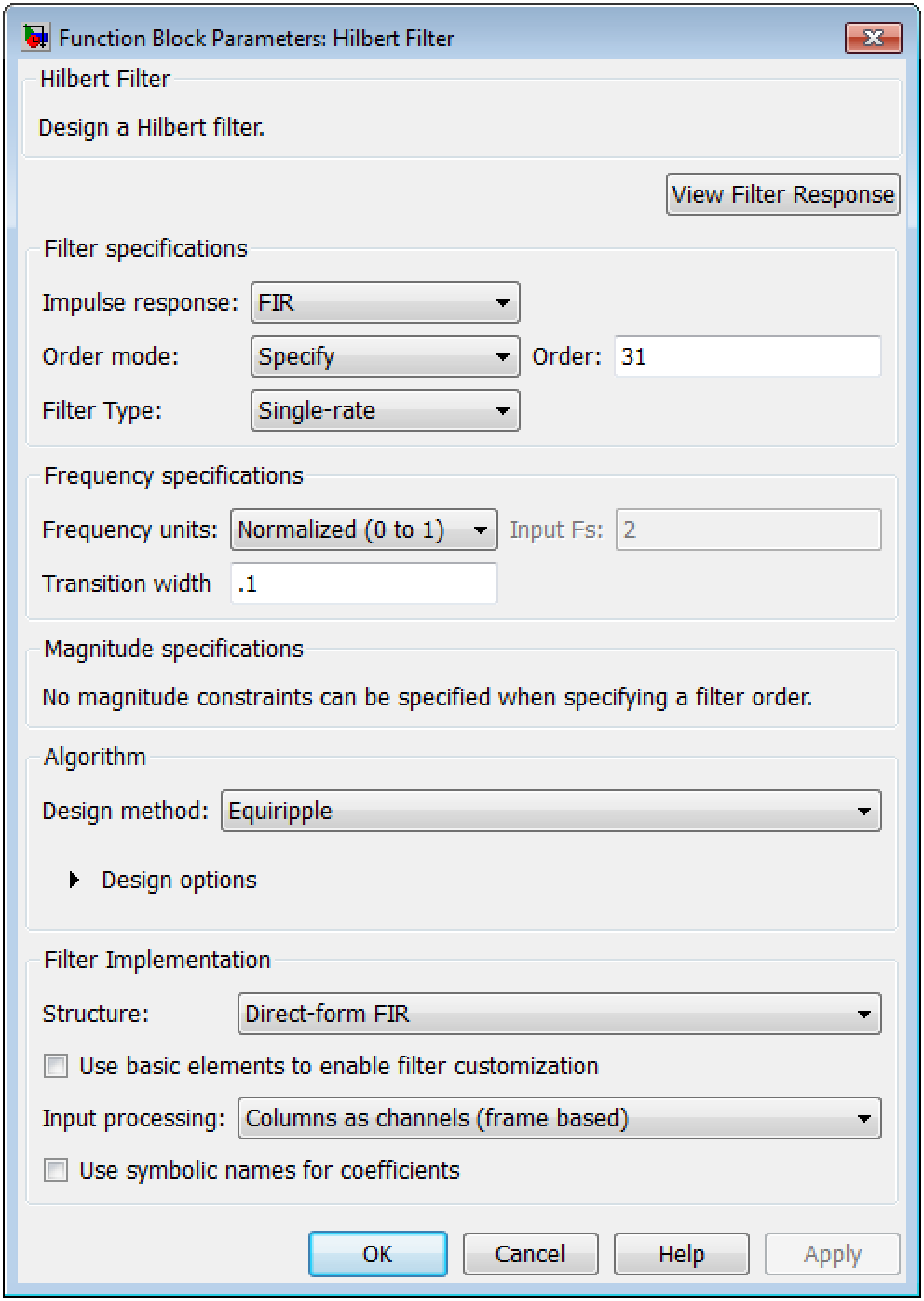
`dspfdesign`

Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See “Hilbert Filter Design — Main Pane” on page 5-761 for more information about the parameters of this block. The **Data Types** and **Code** panes are not available for blocks in the DSP System Toolbox Filter Designs library.


Function Block Parameters: Hilbert Filter
Hilbert Filter
Design a Hilbert filter.
View Filter Response
Filter specifications
Impulse response:
FIR
Order mode:
Specify
Order:
31
Filter Type:
Single-rate
Frequency specifications
Frequency units:
Normalized (0 to 1)
Input Fs:
2
Transition width
.1
Magnitude specifications
No magnitude constraints can be specified when specifying a filter order.
Algorithm
Design method:
Equiripple
Design options
Filter Implementation
Structure:
Direct-form FIR
Use basic elements to enable filter customization
Input processing:
Columns as channels (frame based)
Use symbolic names for coefficients
OK
Cancel
Help
Apply

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop-down list. `FIR` is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note: The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting `Sample-rate converter` activates both factors.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Transition width

Specify the width of the transitions at the ends of the passband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default)
- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default FIR method is `Equiripple`.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

FIR Type

Specify whether to design a type 3 or a type 4 FIR filter. The filter type is defined as follows:

- Type 3 — FIR filter with even order antisymmetric coefficients
- Type 4 — FIR filter with odd order antisymmetric coefficients

Select either `3` or `4` from the drop-down list.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use `Direct-form FIR`, and IIR filters use `Cascade minimum-multiplier allpass`.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

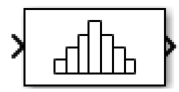
Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2006b

Histogram

Histogram of input or sequence of inputs

Library: DSP System Toolbox / Statistics



Description

The Histogram block computes the frequency distribution of the input elements along each column or along the entire input. You can specify the dimension using the **Find the histogram over** parameter. The block distributes the input elements based on their value into a number of discrete bins, which you specify through the **Number of bins** parameter. Specify the highest input value that the histogram can bin through the **Upper limit of histogram** parameter, B_M. Specify the lowest value through the **Lower limit of histogram** parameter, B_m. The width of each bin is equal and is given by:

$$\Delta = \frac{B_M - B_m}{n}$$

Each bin is centered at the following location:

$$B_m + \left(k + \frac{1}{2}\right)\Delta \qquad\qquad k = 0,1,2,\ldots,n-1$$

When the input data is real, the bin boundaries are cast into the double data type. When the input data is complex, bin boundaries for double-precision inputs are cast into the data type double and bin boundaries for integer inputs are cast into data type double and squared.

To track the frequency distribution of inputs over a period of time, select the **Running histogram** parameter.

When the input is complex, the block sorts the elements according to their magnitude.

Ports

Input

`In` — Data input

vector | matrix | *N*-D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input data type must be double precision, single precision, integer, or fixed point, with power-of-two slope and zero bias.

This port is unnamed until you select the **Running histogram** parameter and set the **Reset port** parameter to any option other than `None`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed_point`
Complex Number Support: Yes

`Rst` — Reset port

scalar

Specify the reset event that causes the block to reset the running histogram. The reset signal and the input data signal must have the same rate.

Dependencies

To enable this port, select the **Running histogram** parameter and set the **Reset port** parameter to any option other than `None`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Port_1` — Histogram output

vector | matrix | *N*-D array

When you do not select the **Running histogram** parameter, the blocks computes the frequency distribution in each column of the input, or along the entire input. Consider a two-dimensional input signal of size *M*-by-*N*.

When you set **Find the histogram over** to:

- `Each column` — The block computes a histogram for each column of the input. The output is an n-by-N matrix, where n is the **Number of bins** you specify. The jth column of the output matrix contains the histogram for the data in the jth column of the M-by-N input matrix.
- `Entire input` — The block computes a histogram for the entire input vector, matrix, or N-D array. The output is an n-by-1 vector, where n is the **Number of bins** you specify.

When you select the **Running histogram** parameter, the block computes the frequency distribution of all inputs over a period of time. In this case, when you set **Find the histogram over** to:

- `Each column` — The block computes a running histogram for each column of the input. The output is an n-by-N matrix, where n is the **Number of bins** you specify. The jth column of the output matrix contains the running histogram for the data in the jth column of the M-by-N matrix input.
- `Entire input` — The block computes the running histogram for the entire input vector, matrix, or N-D array. The output is an n-by-1 vector, where n is the **Number of bins** you specify.

The output data type is `uint32` when the input has a data type other than `single` or `double`. The largest number that can be represented by `uint32` is $2^{32}-1$. If the range of any input exceeds this value, the block wraps the value back to `0`.

Data Types: `single` | `double` | `uint32`

Parameters

Main Tab

`Lower limit of histogram` — Lower boundary of the lowest-valued bin

scalar

Specify the lower boundary of the lowest-valued bin as a real scalar. This parameter does not accept `NaN` or `Inf`. If the input has a value less than **Lower limit of histogram**, the block places this element in the lowest-valued bin.

`Upper limit of histogram` — Upper boundary of the highest-valued bin
scalar

Specify the upper boundary of the highest-valued bin as a real scalar. This parameter does not accept `NaN` or `Inf`. If the input has a value greater than **Upper limit of histogram**, the block places this element in the highest-valued bin.

`Number of bins` — Number of histogram bins
positive integer

Specify the number of histogram bins as a positive integer.

`Find the histogram over` — Dimension over which the block computes the histogram
`Each column` (default) | `Entire input`

When you do not select the **Running histogram** parameter and you set **Find the histogram over** parameter to:

- `Each column` — The block computes the histogram over each column.
- `Entire input` — The block outputs the histogram over the entire input.

When you select the **Running histogram** parameter and you set **Find the histogram over** parameter to:

- `Each column` — The block computes the running histogram over each column.
- `Entire input` — The block outputs the running histogram over the entire input.

`Normalized` — Number of histogram bins
off (default) | on

When you select this parameter, the block output, y, is normalized, that is, `sum(y) = 1`. This parameter does not apply for fixed-point input signals.

`Running histogram` — Option to select running histogram
off (default) | on

When you select this parameter, the block tracks the frequency distribution of inputs over a period of time.

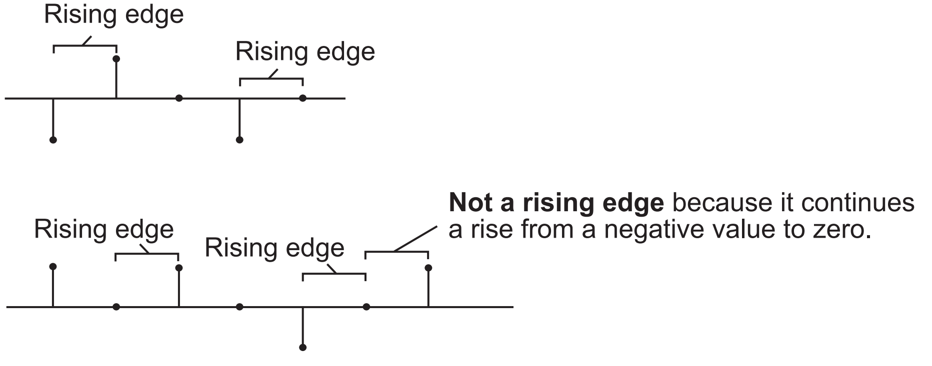
`Reset port` — Reset event
`Non-zero sample` (default) | `None` | `Rising edge` | `Falling edge` | `Either edge` |

The block resets the running histogram and empties all the bins whenever a reset event is detected at the optional **Rst** port. The reset sample time must equal the input sample time.

Use this parameter to specify the reset event.

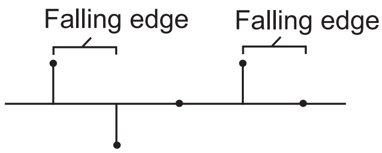
- `Non-zero sample` — Triggers a reset operation at each sample time, when the **Rst** input is not zero.
- `None` — Disables the **Rst** port.
- `Rising edge` — Triggers a reset operation when the **Rst** input does one of the following:

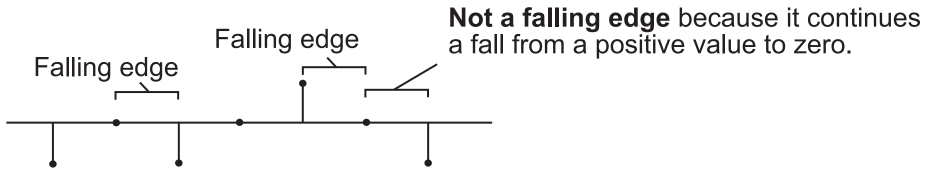
  - Rises from a negative value to either a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.



- `Falling edge` — Triggers a reset operation when the **Rst** input does one of the following:

  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.





- `Either edge` — Triggers a reset operation when the **Rst** input is a `Rising edge` or `Falling edge`.

Note: When running simulations in the Simulink multitasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

Dependencies

To enable this parameter, select the **Running histogram** parameter.

Data Types Tab

Note: To use these parameters, the data input must be complex and fixed point. For all other inputs, the parameters on the **Data Types** tab are ignored.

`Rounding mode` — Method of rounding operation

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Select the rounding mode for fixed-point operations.

`Overflow mode` — Method of overflow action

`Wrap` (default) | `Saturate`

Select the overflow mode for fixed-point operations.

`Product output` — Product output data type

`Inherit: Same as input` (default) | `fixdt([],16,0)`

The squares of the real and imaginary parts of the complex input are stored in the **Product output** data type.

- `Inherit: Same as input` — The block specifies the product output data type to be the same as the input data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink).

`Accumulator` — Accumulator data type

`Inherit: Same as input` (default) | `Inherit: Same as product output` | `fixdt([],16,0)`

The result of the sum of the squares of the real and imaginary parts of the complex input are stored in the **Accumulator** data type.

You can set this parameter to:

- `Inherit: Same as input` — The block specifies the accumulator data type to be the same as the input data type.
- `Inherit: Same as product output` — The block specifies the accumulator data type to be the same as the product output data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink).

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block.

Model Examples

Algorithms

Histogram

The histogram value for a given bin represents the frequency of occurrence of the input values bracketed by that bin. **Upper limit of histogram** specifies the upper boundary of the highest-valued bin, B_M. **Lower limit of histogram** specifies the lower boundary of the lowest-valued bin, B_m. The bins have equal width, given by:

$$\Delta = \frac{B_M - B_m}{n}$$

The bins are centered at the following location:

$$B_m + \left(k + \frac{1}{2}\right)\Delta \qquad\qquad k = 0,1,2,...,n-1$$

n is the number of bins, specified through the **Number of bins** parameter.

Input values that fall on the border between two bins are placed into the lower valued bin. Each bin includes its upper boundary. For example, a bin of width 4 centered on the value 5 contains the input value 7, but not 3. Input values greater than the **Upper limit of histogram** parameter are placed in the highest valued bin. Similarly, values less than the **Lower limit of histogram** parameter are placed in the lowest valued bin.

The block sorts the complex data into bins according to the magnitude of the data. The magnitude is computed by taking the sum of the squares of the real and imaginary components of the complex data.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion

Design and simulate fixed-point algorithms using Fixed-Point Designer™.

The parameters on the **Data Types** tab are used only for complex fixed-point inputs. Complex inputs are sorted by their magnitude, which is the sum of the squares of the real and imaginary components of the input. The results of the squares of the real and imaginary parts are stored in the **Product output** data type. The result of the sum of the squares is stored in the **Accumulator** data type. The parameters on the **Data Types** tab are ignored for all other inputs.

See Also

See Also

Functions
`histogram`

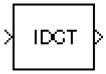
System Objects
dsp.Maximum | dsp.Minimum | dsp.Mean | dsp.Median

Blocks
`Maximum` | `Median` | `Minimum` | `Sort`

Introduced before R2006a

IDCT

Inverse discrete cosine transform (IDCT) of input



Library

Transforms

`dspxfrm3`

Description

The IDCT block computes the inverse discrete cosine transform (IDCT) of each channel in the M-by-N input matrix, `u`.

```
y = idct(u)     % Equivalent MATLAB code
```

For all N-D input arrays, the block computes the IDCT across the first dimension. The size of the first dimension (frame size), must be a power of two. To work with other frame sizes, use the `Pad` block to pad or truncate the frame size to a power-of-two length.

When the input is an M-by-N matrix, the block treats each input column as an independent channel containing M consecutive samples. The block outputs an M-by-N matrix whose lth column contains the length-M IDCT of the corresponding input column.

$$y(m,l) = \sum_{k=1}^{M} w(k)u(k,l)\cos\frac{\pi(2m-1)(k-1)}{2M}, \qquad m = 1,\ldots,M$$

where

$$w(k) = \begin{cases} \dfrac{1}{\sqrt{M}}, & k = 1 \\ \sqrt{\dfrac{2}{M}}, & 2 \le k \le M \end{cases}$$

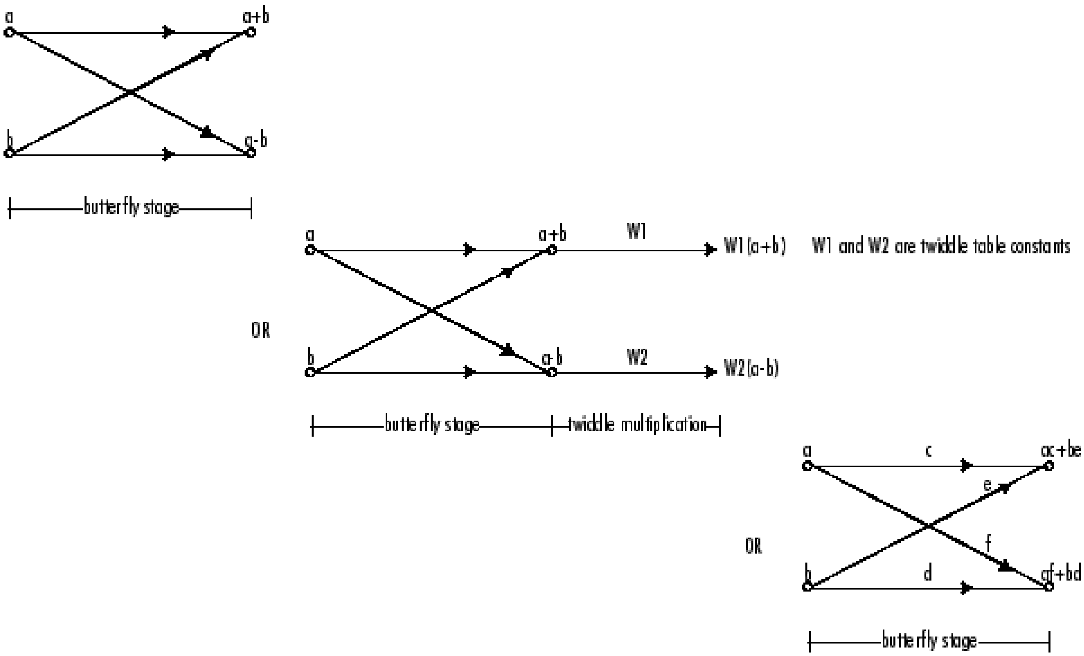
The **Sine and cosine computation** parameter determines how the block computes the necessary sine and cosine values. This parameter has two settings, each with its advantages and disadvantages, as described in the following table.

Sine and Cosine Computation Parameter Setting	Sine and Cosine Computation Method	Effect on Block Performance
`Table lookup`	The block computes and stores the trigonometric values before the simulation starts, and retrieves them during the simulation. When you generate code from the block, the processor running the generated code stores the trigonometric values computed by the block in a speed-optimized table, and retrieves the values during code execution.	The block usually runs much more quickly, but requires extra memory for storing the precomputed trigonometric values.
`Trigonometric fcn`	The block computes sine and cosine values during the simulation. When you generate code from the block, the processor running the generated code computes the sine and cosine values while the code runs.	The block usually runs more slowly, but does not need extra data memory. For code generation, the block requires a support library to emulate the trigonometric functions, increasing the size of the generated code.

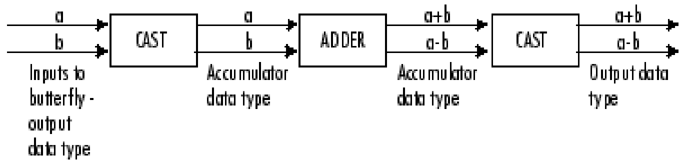
Fixed-Point Data Types

The following diagrams show the data types used within the IDCT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the IDCT block dialog as discussed in “Dialog Box” on page 2-940.

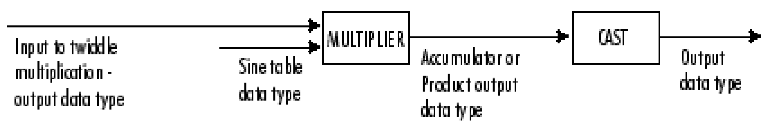
Inputs to the IDCT block are first cast to the output data type and stored in the output buffer. Each butterfly stage processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type.



Butterfly Stage Data Types



Twiddle Multiplication Data Types

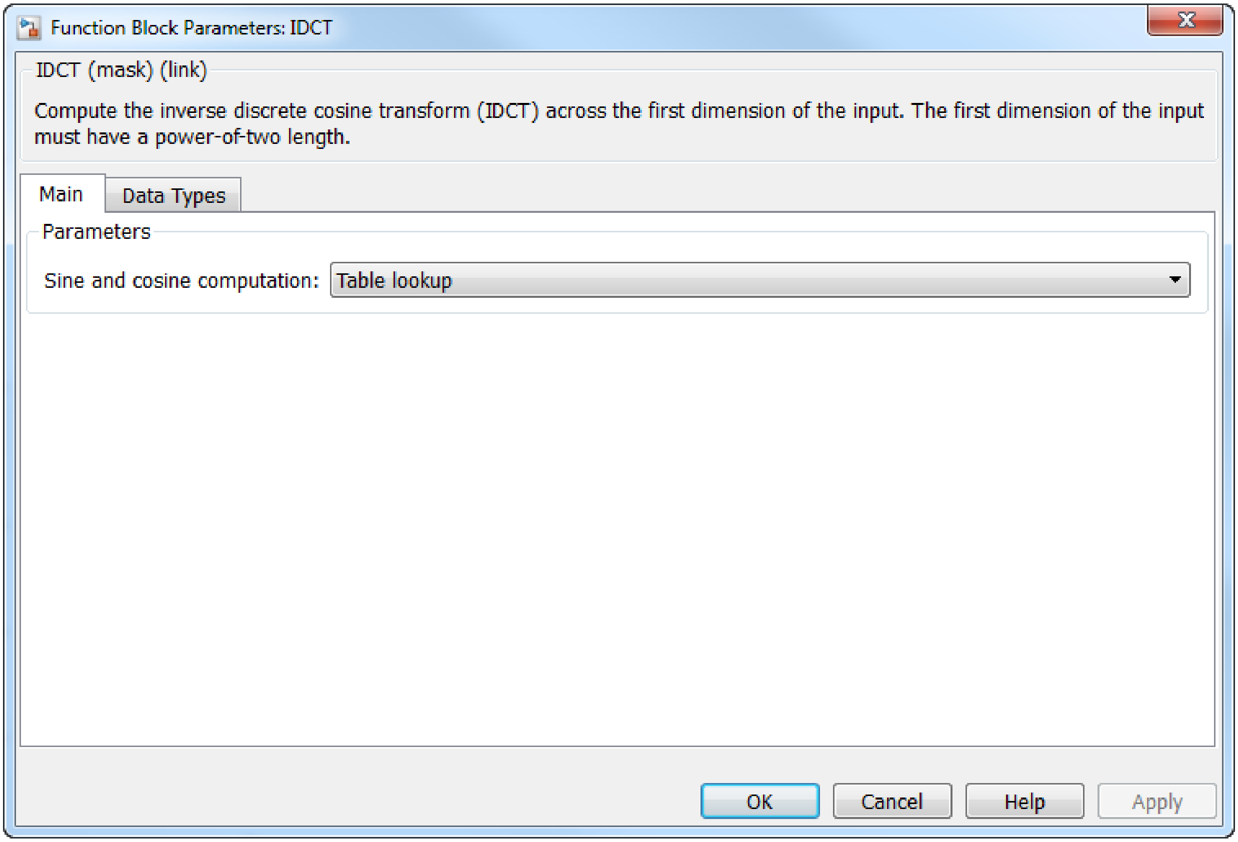


The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

Note: When the block input is fixed point, all internal data types are signed fixed point.

Dialog Box

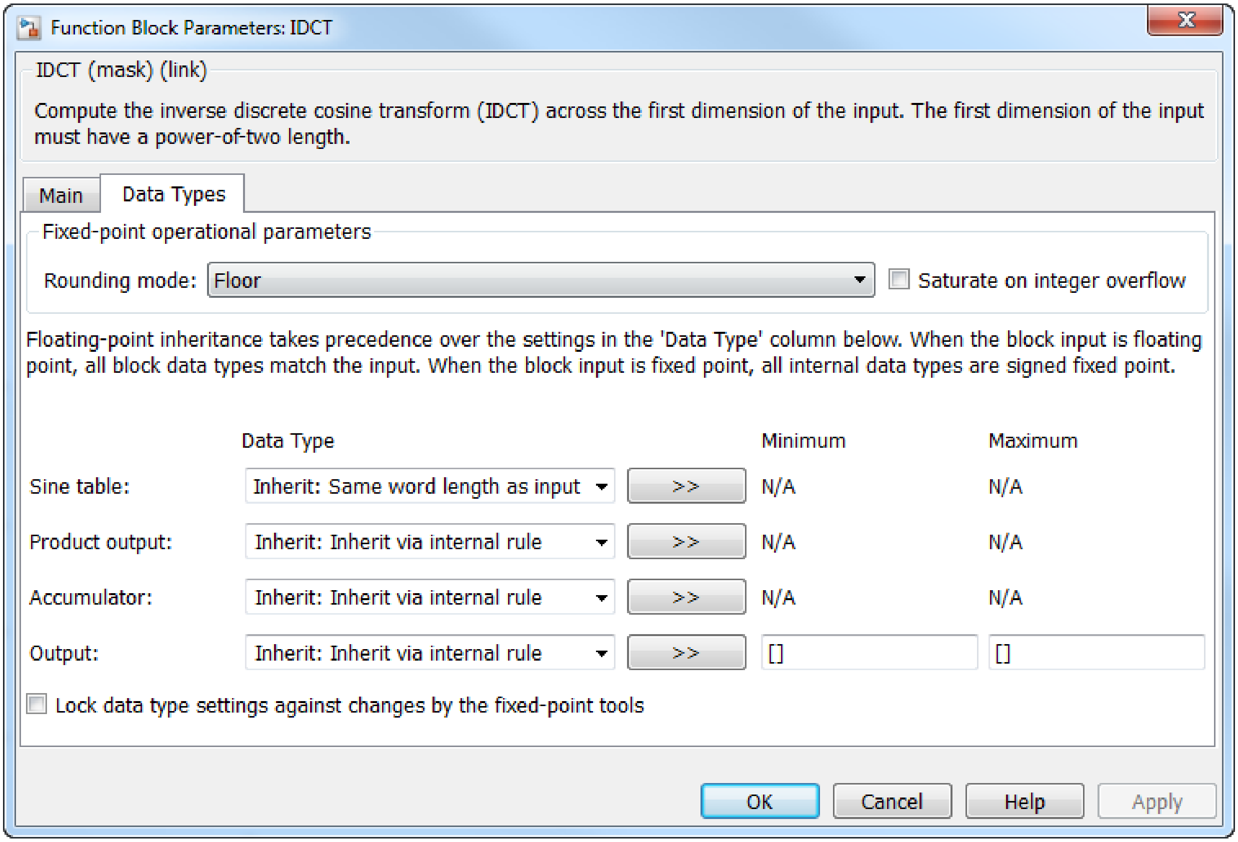
The **Main** pane of the IDCT block dialog appears as follows.



Sine and cosine computation

Sets the block to compute sines and cosines by either looking up sine and cosine values in a speed-optimized table (`Table lookup`), or by making sine and cosine function calls (`Trigonometric fcn`). See the table in the "Description" on page 2-937 section.

The **Data Types** pane of the IDCT block dialog appears as follows.



Rounding mode

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameter; they always round to `Nearest`.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations. The sine table values do not obey this parameter; instead, they are always saturated.

Sine table data type

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

The sine table values do not obey the **Rounding mode** and **Saturate on integer overflow** parameters; instead, they are always saturated and rounded to `Nearest`.

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-938 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-938 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-938 for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`.

  When you select `Inherit: Inherit via internal rule`, the block calculates the output word length and fraction length automatically. The internal rule first calculates an ideal output word length and fraction length using the following equations:

$$WL_{ideal\ output} = WL_{input} + floor(\log_2(DCT\ length - 1)) + 1$$

$$FL_{ideal\ output} = FL_{input}$$

  Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges" (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

See Also

See Also

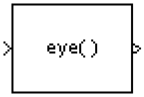
Functions
`idct`

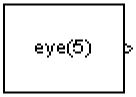
Blocks
`DCT` | `IFFT`

Introduced before R2006a

Identity Matrix

Generate matrix with ones on main diagonal and zeros elsewhere





Library

- Sources

  `dspsrcs4`
- Math Functions / Matrices and Linear Algebra / Matrix Operations

  `dspmtrx3`

Description

The Identity Matrix block generates a rectangular matrix with ones on the main diagonal and zeros elsewhere.

When you select the **Inherit output port attributes from input port** check box, the input port is enabled, and an *M*-by-*N* matrix input generates an *M*-by-*N* matrix output with the same sample period as the input. The values in the input matrix are ignored.

```
y = eye([M N])      % Equivalent MATLAB code
```

When you do not select the **Inherit output port attributes from input port** check box, the input port is disabled, and the dimensions of the output matrix are determined by the **Matrix size** parameter. A scalar value, *M*, specifies an *M*-by-*M* identity matrix, while a two-element vector, [*M N*], specifies an *M*-by-*N* unit-diagonal matrix. You can specify the output sample period using the **Sample time** parameter.

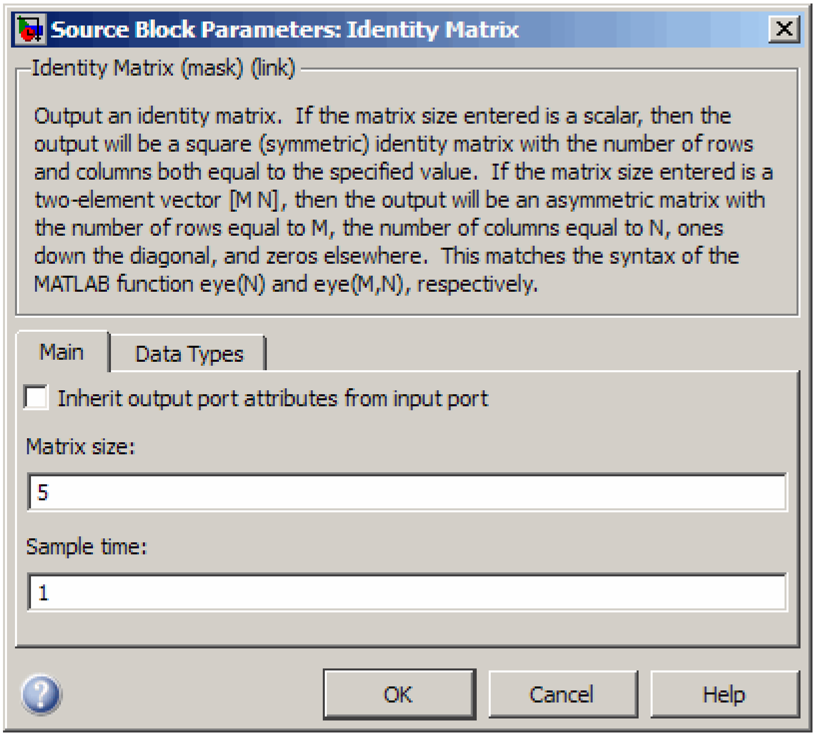
Examples

Set **Matrix size** to `[3 6]` to generate the 3-by-6 unit-diagonal matrix below.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Dialog Box

The **Main** pane of the Identity Matrix block dialog appears as follows.



Inherit output port attributes from input port

Enables the input port when selected. In this mode, the output inherits its dimensions, sample period, and data type from the input. The output is always real.

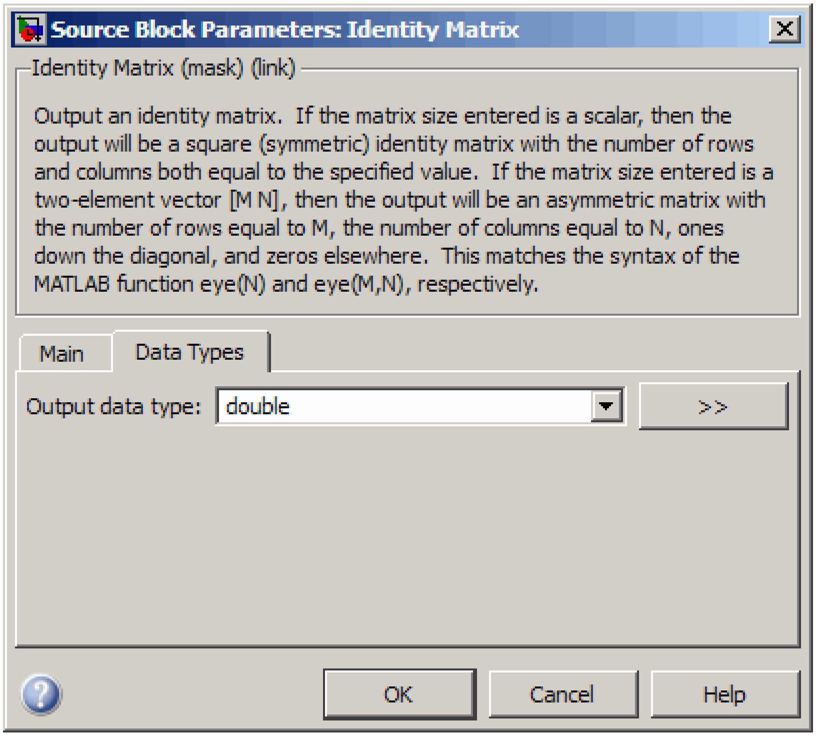
Matrix size

The number of rows and columns in the output matrix: a scalar *M* for a square *M*-by-*M* output, or a vector [*M N*] for an *M*-by-*N* output. This parameter is disabled when you select **Inherit input port attributes from input port**.

Sample time

The discrete sample period of the output. This parameter is disabled when you select **Inherit input port attributes from input port**.

The **Data Types** pane of the Identity Matrix block dialog appears as follows.



Output data type

Specify the output data type for this block. You can select one of the following:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`. When you select this option, the output data type and scaling matches that of the next downstream block.
- A built in data type, such as `double`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

Constant

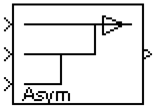
Simulink

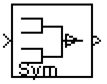
`eye`	MATLAB

Introduced before R2006a

IDWT

Inverse discrete wavelet transform (IDWT) of input or reconstruct signals from subbands with smaller bandwidths and slower sample rates





Library

Transforms

`dspxfrm3`

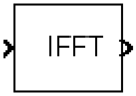
Description

The IDWT block is the same as the Dyadic Synthesis Filter Bank block in the Multirate Filters library, but with different default settings. See the `Dyadic Synthesis Filter Bank` block reference page for more information on how to use the block.

Introduced before R2006a

IFFT

Inverse fast Fourier transform (IFFT) of input



Library

Transforms

`dspxfrm3`

Description

The IFFT block computes the inverse fast Fourier transform (IFFT) across the first dimension of an *N-D* input array.

When you specify an FFT length not equal to the length of the input vector, (or first dimension of the input array), the block implements zero-padding, truncating, or modulo-*M*, (FFT length) data wrapping. This occurs before the IFFT operation.

```
y = ifft(u,M)  % P ≤ M
```

Wrapping:

```
y(:,l) = ifft(datawrap(u(:,l),M))  % P > M; l = 1,...,N
```

Truncating:

```
y (:,l) = ifft(u,M)     % P > M; l = 1,...,N
```

When the input length, *P*, is greater than the FFT length, *M*, you may see magnitude increases in your IFFT output. These magnitude increases occur because the IFFT block uses modulo-*M* data wrapping to preserve all available input samples.

To avoid such magnitude increases, you can truncate the length of your input sample, *P*, to the FFT length, *M*. To do so, place a `Pad` block before the IFFT block in your model.

The kth entry of the lth output channel, $y(k, l)$, is equal to the kth point of the M-point inverse discrete Fourier transform (IDFT) of the lth input channel:

$$y(k,l) = \frac{1}{M}\sum_{p=1}^{P} u(p,l)e^{j2\pi(p-1)(k-1)/M} \qquad k = 1,\ldots,M$$

The output of this block has the same dimensions as the input. If the input signal has a floating-point data type, the data type of the output signal uses the same floating-point data type. Otherwise, the output can be any fixed-point data type. The block computes scaled and unscaled versions of the IFFT.

The input to this block can be floating-point or fixed-point, real or complex, and conjugate symmetric. The block uses one of two possible FFT implementations. You can select an implementation based on the FFTW library or an implementation based on a collection of Radix-2 algorithms. You can select `Auto` to allow the block to choose the implementation.

FFTW Implementation

The FFTW implementation provides an optimized FFT calculation including support for power-of-two and non-power-of-two transform lengths in both simulation and code generation. Generated code using the FFTW implementation will be restricted to MATLAB host computers. The data type must be floating-point. Refer to Simulink Coder for more details on generating code.

Radix-2 Implementation

The Radix-2 implementation supports bit-reversed processing, fixed or floating-point data, and allows the block to provide portable C-code generation using the Simulink Coder. The dimension M of the M-by-N input matrix, must be a power of two. To work with other input sizes, use the `Pad` block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation.

With Radix-2 selected, the block implements one or more of the following algorithms:

- Butterfly operation
- Double-signal algorithm
- Half-length algorithm
- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm

Radix-2 Algorithms for Real or Complex Input Complexity

Parameter Settings	Algorithms Used for IFFT Computation
☐ Input is in bit-reversed order ☐ Input is conjugate symmetric	Bit-reversal operation and radix-2 DIT
☑ Input is in bit-reversed order ☐ Input is conjugate symmetric	Radix-2 DIT
☐ Input is in bit-reversed order ☑ Input is conjugate symmetric	Bit-reversal operation and radix-2 DIT in conjunction with the half-length and double-signal algorithms
☑ Input is in bit-reversed order ☑ Input is conjugate symmetric	Radix-2 DIT in conjunction with the half-length and double-signal algorithms

Radix-2 Optimization for the Table of Trigonometric Values

In certain situations, the block's Radix–2 algorithm computes all the possible trigonometric values of the twiddle factor

$$e^{j\frac{2\pi k}{K}}$$

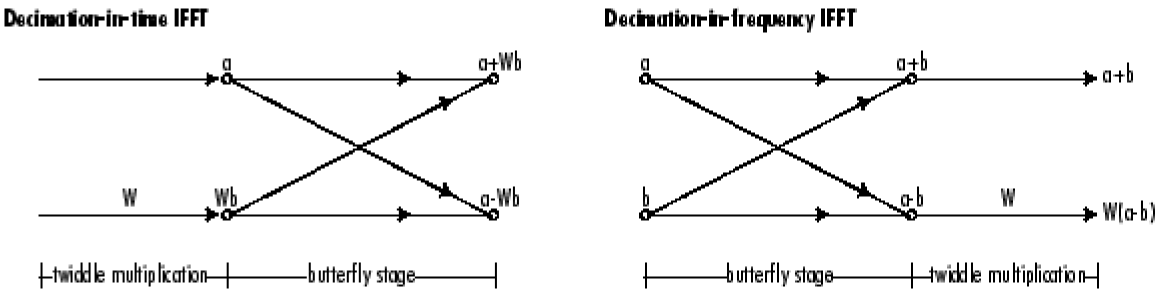
where K is the greater value of either M or N and $k = 0,\cdots,K-1$. The block stores these values in a table and retrieves them during simulation. The number of table entries for fixed-point and floating-point is summarized in the following table:

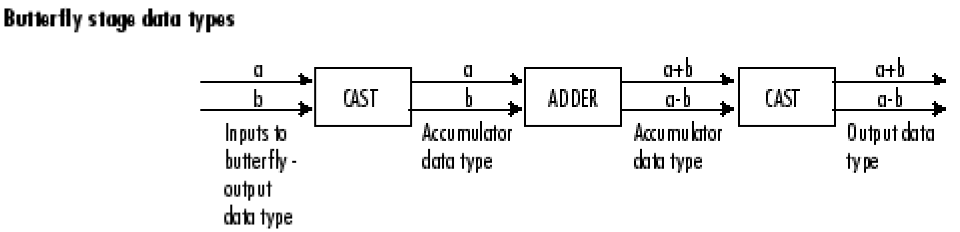
Number of Table Entries for N-Point FFT	
floating-point	3 N/4
fixed-point	N

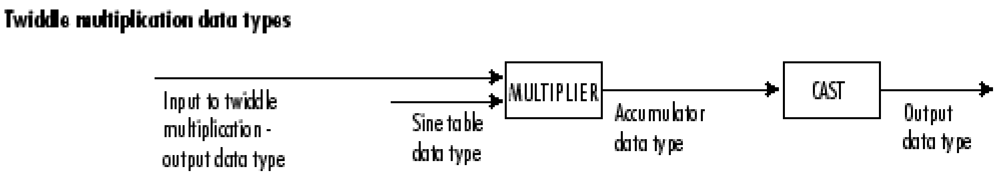
Fixed-Point Data Types

The following diagrams show the data types used in the IFFT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the IFFT dialog box as discussed in "Dialog Box" on page 2-957.

Inputs to the IFFT block are first cast to the output data type and stored in the output buffer. Each butterfly stage then processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type. The block multiplies in a twiddle factor before each butterfly stage in a decimation-in-time IFFT and after each butterfly stage in a decimation-in-frequency IFFT.





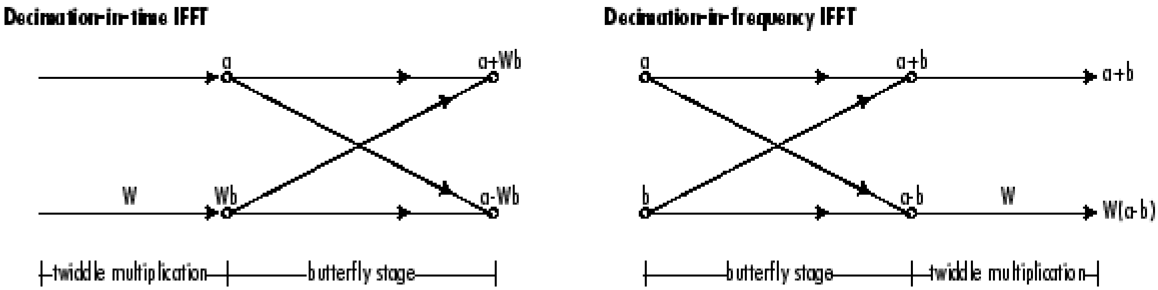


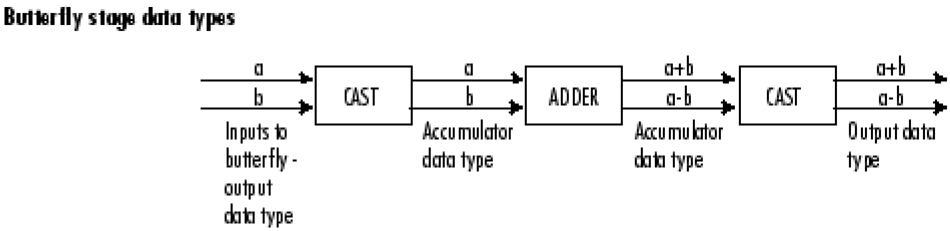
The multiplier output appears in the accumulator data type because both of the inputs to the multiplier are complex. For details on the complex multiplication performed, refer to “Multiplication Data Types”.

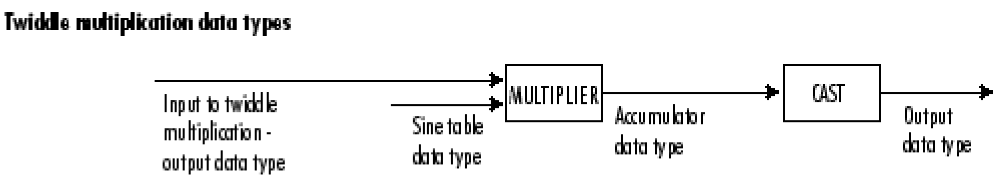
Fixed-Point Data Types

The following diagrams show the data types used within the IFFT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the IFFT block dialog, as discussed in "Dialog Box" on page 2-957.

The IFFT block first casts input to the output data type and then stores it in the output buffer. Each butterfly stage then processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type. The block multiplies in a twiddle factor before each butterfly stage in a decimation-in-time IFFT, and after each butterfly stage in a decimation-in-frequency IFFT.





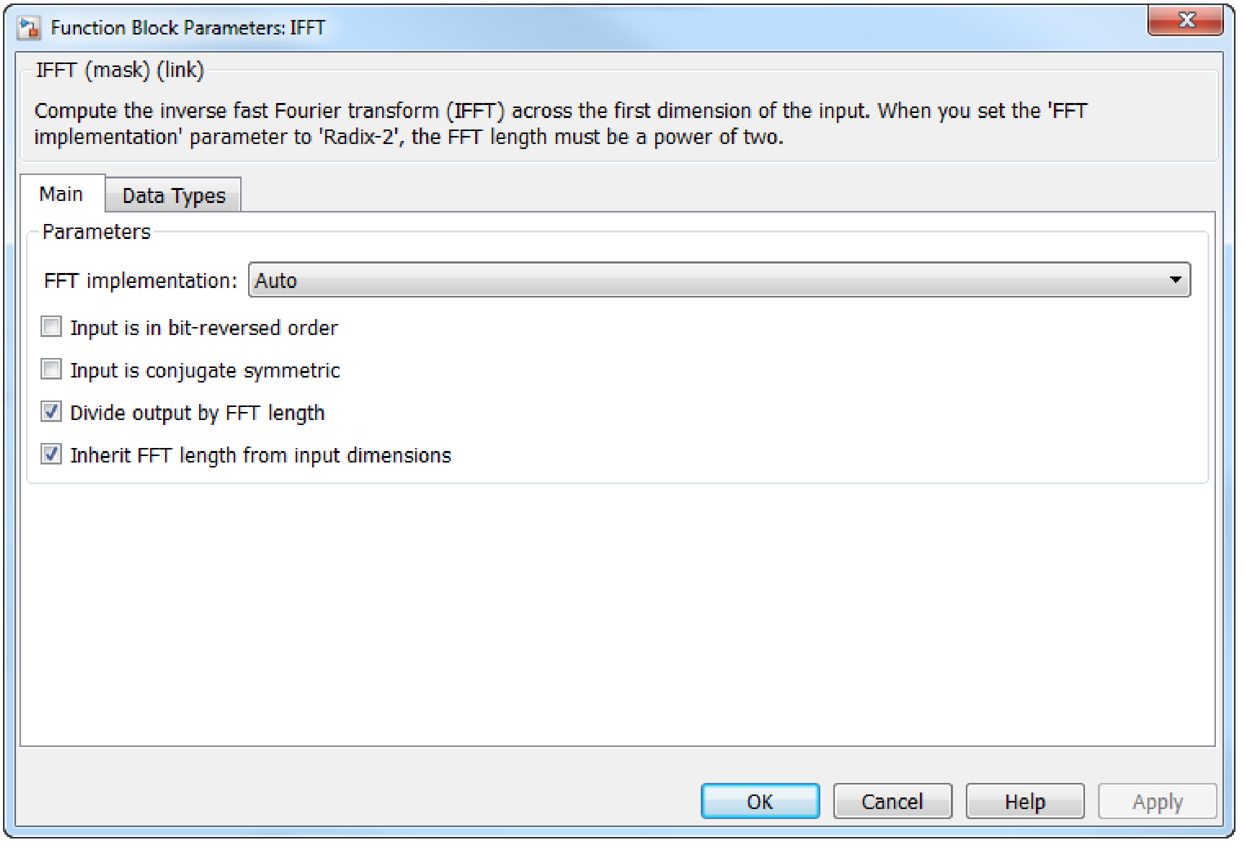


The output of the multiplier is in the accumulator data type because both of the inputs to the multiplier are complex. For details on the complex multiplication performed, see “Multiplication Data Types”.

Note: When the block input is fixed point, all internal data types are signed fixed point.

Dialog Box

The **Main** pane of the IFFT block dialog appears as follows.



FFT implementation

Set this parameter to `FFTW` to support an arbitrary length input signal. The block restricts generated code with FFTW implementation to MATLAB host computers.

Set this parameter to `Radix-2` for bit-reversed processing, fixed or floating-point data, or for portable C-code generation using the Simulink Coder. The dimension M of the M-by-N input matrix, must be a power of two. To work with other input sizes, use the `Pad` block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation. See "Radix-2 Implementation" on page 2-952.

Set this parameter to `Auto` to let the block choose the FFT implementation. For non-power-of-two transform lengths, the block restricts generated code to MATLAB host computers.

Input is in bit-reversed order

Select or clear this check box to designate the order of the input channel elements. Select this check box when the input should appear in reversed order, and clear it when the input should appear in linear order. The block yields invalid outputs when you do not set this parameter correctly. This check box only appears when you set the **FFT implementation** parameter to `Radix-2` or `Auto`.

You cannot select this check box if you have cleared the **Inherit FFT length from input dimensions** check box, and you are specifying the FFT length using the **FFT length** parameter. Also, it cannot be selected when you set the **FFT implementation** parameter to `FFTW`.

For more information on ordering of the output, see "Linear and Bit-Reversed Output Order".

Input is conjugate symmetric

Select this option when the block inputs conjugate symmetric data and you want real-valued outputs. Selecting this check box optimizes the block's computation method.

The FFT block yields conjugate symmetric output when you input real-valued data. Taking the IFFT of a conjugate symmetric input matrix produces real-valued output. Therefore, if the input to the block is both floating point and conjugate symmetric, and you select this check box, the block produces real-valued outputs.

You cannot select this check box if you have cleared the **Inherit FFT length from input dimensions** check box, and you are specifying the FFT length using the **FFT length** parameter.

If you input conjugate symmetric data to the IFFT block and do not select this check box, the IFFT block outputs a complex-valued signal with small imaginary parts. The block outputs invalid data if you select this option with non conjugate symmetric input data.

Divide output by FFT length

When you select this check box, the block computes its output according to the IDFT equation, discussed in the Description section.

When you clear this check box, the block computes the output using a modified version of the IDFT: $M \cdot y(k,l)$, which is defined by the following equation:

$$M \cdot y(k,l) = \sum_{p=1}^{P} u(p,l) e^{j2\pi(p-1)(k-1)/M} \quad k = 1,\ldots,M$$

Notice, the modified IDFT equation does not include the multiplication factor of 1/*M*.

Inherit FFT length from input dimensions

Select to inherit the FFT length from the input dimensions. If you do not select this parameter, the **FFT length** parameter becomes available to specify the length. You cannot clear this parameter when you select either the **Input is in bit-reversed order** or the **Input is conjugate symmetric** parameter.

FFT length

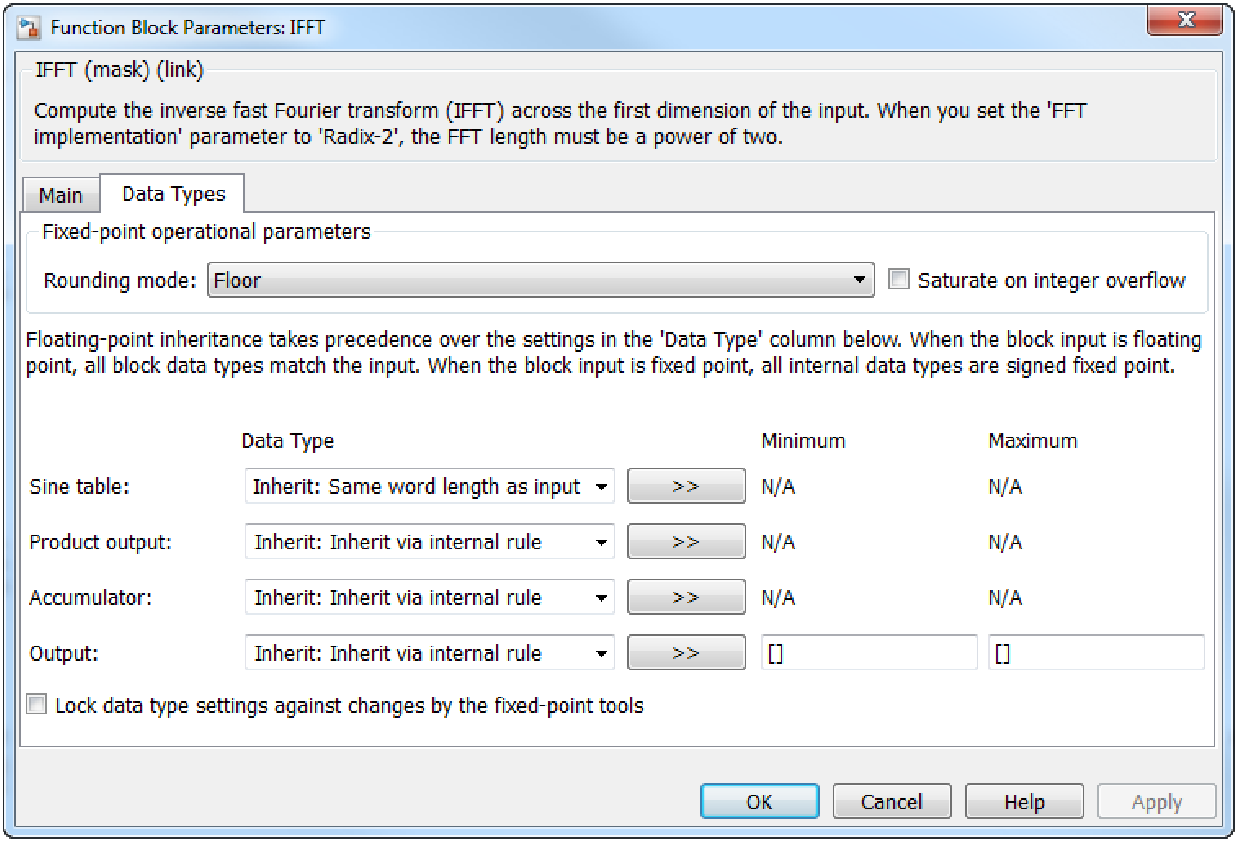
Specify FFT length. This parameter only becomes available if you do not select the **Inherit FFT length from input dimensions** parameter.

When you set the **FFT implementation** parameter to `Radix-2`, or when you check the **Output in bit-reversed order** check box, this value must be a power of two.

Wrap input data when FFT length is shorter than input length

Choose to wrap or truncate the input, depending on the FFT length. If this parameter is checked, modulo-length data wrapping occurs before the FFT operation, given FFT length is shorter than the input length. If this property is unchecked, truncation of the input data to the FFT length occurs before the FFT operation. The default is checked.

The **Data Types** pane of the IFFT block dialog appears as follows.



Rounding mode

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameter; instead, they always round to `Nearest`.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations. The sine table values do not obey this parameter; instead, they are always saturated.

Sine table data type

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

The sine table values do not obey the **Rounding mode** and **Saturate on integer overflow** parameters; instead, they are always saturated and rounded to `Nearest`.

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-955 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-955 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-955 for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`.

  When you select `Inherit: Inherit via internal rule`, the block calculates the output word length and fraction length automatically. The equations that the block uses to calculate the ideal output word length and fraction length depend on the setting of the **Divide output by FFT length** check box.

  - When you select the **Divide output by FFT length** check box, the ideal output word and fraction lengths are the same as the input word and fraction lengths.
  - When you clear the **Divide output by FFT length** check box, the block computes the ideal output word and fraction lengths according to the following equations:

$$WL_{ideal\ output} = WL_{input} + floor(\log_2(FFT\ length - 1)) + 1$$

$$FL_{ideal\ output} = FL_{input}$$

  Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Examples

See “Transform Frequency-Domain Data into Time Domain” in the *DSP System Toolbox User's Guide*.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only)

Port	Supported Data Types
	• 8-, 16-, and 32-bit signed integers

References

[1] Orfanidis, S. J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1996, p. 497.

[2] Proakis, John G. and Dimitris G. Manolakis. *Digital Signal Processing*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 1996.

[3] FFTW (http://www.fftw.org)

[4] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT,"*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- When the following conditions apply, the executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB:
  - **FFT implementation** is set to `FFTW`.
  - **Inherit FFT length from input dimensions** is cleared, and **FFT length** is set to a value that is not a power of two.

  Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "How To Run a Generated Executable Outside MATLAB".

- When the FFT length is a power of two, you can generate standalone C and C++ code from this block.

See Also

See Also

Functions
`bitrevorder` | `fft` | `ifft`

Blocks
`FFT` | `IDCT` | `Pad`

Introduced before R2006a

IIR Halfband Interpolator

Interpolate signal using polyphase IIR halfband filter

Library

Filtering/Filter Designs

`dspfdesign`

Description

The IIR Halfband Interpolator block performs efficient polyphase interpolation of the input signal by a factor of two. To design the halfband filter, you can specify the block to use an elliptic design or a quasi-linear phase design. The block uses these design methods to compute the filter coefficients. To filter the inputs, the block uses a polyphase structure. The allpass filters in the polyphase structure are in a minimum multiplier form.

Elliptic design introduces nonlinear phase and creates the filter using fewer coefficients than quasi linear design. Quasi-linear phase design overcomes phase nonlinearity at the cost of additional coefficients.

Alternatively, instead of designing the halfband filter using a design method, you can specify the filter coefficients directly. When you choose this option, the allpass filters in the two branches of the polyphase implementation can be in a minimum multiplier form or in a wave digital form.

You can also use the block to implement the synthesis portion of a two-band filter bank to synthesize a signal from lowpass and highpass subbands.

The input signal can be a real- or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel.

Parameters

Filter specification

Parameters used to design the IIR halfband filter. Because the filter design has only two degrees of freedom, you can specify only two of the three parameters:

- `Transition width and stopband attenuation` (default) — Design the filter using **Transition width (Hz)** and **Stopband attenuation (dB)**. This design is the minimum order design.
- `Filter order and transition width` — Design the filter using **Filter order** and **Transition width (Hz)**.
- `Filter order and stopband attenuation` — Design the filter using **Filter order** and **Stopband attenuation (dB)**.
- `Coefficients`— Specify the filter coefficients directly using the enabled parameters.

Transition width (Hz)

Transition width of the IIR halfband filter, specified as a real positive scalar in Hz. The transition width must be less than half the input sample rate.This parameter applies when **Filter specification** is set to `Filter order and transition width` or `Transition width and stopband attenuation`. The default is `4.1e3`.

Filter order

Filter order, specified as a finite positive integer. If you set **Design method** to `Elliptic`, then **Filter order** must be an odd integer greater than one. If you set **Design method** to `Quasi-linear phase`, then **Filter order** must be a multiple of four. This parameter applies when **Filter specification** is set to `Filter order and transition width` or `Filter order and stopband attenuation`. The default is `9`.

Stopband attenuation (dB)

Minimum attenuation needed in the stopband of the IIR halfband filter, specified as a real positive scalar in dB. This parameter applies when **Filter specification** is set to `Filter order and stopband attenuation` or `Transition width and stopband attenuation`. The default is `80`.

Design method

Design method for the IIR halfband filter.

- `Elliptic` (default) — The filter has nonlinear phase and uses few coefficients.
- `Quasi-linear phase` — The first branch of the polyphase filter structure is a pure delay, which results in an approximately linear phase response.

This parameter applies when you set **Filter specification** to any option except `Coefficients`.

Internal allpass structure

Internal allpass filter implementation structure, specified as `Minimum multiplier` or `Wave Digital Filter`. This parameter applies when you set **Filter specification** to `Coefficients`. Each structure uses a different coefficients set, independently stored in the corresponding coefficients property. The default is `Minimum multiplier`.

Make the first branch a pure delay

When you select this check box, the first branch of the polyphase filter structure becomes a pure delay, and the **Branch 1 allpass polynomial coefficients** and **Branch 1 Wave Digital coefficients** parameters do not apply. This parameter applies when you set **Filter specification** to `Coefficients`.

By default, this check box is selected.

Delay length in samples for branch 1

Length of the first branch delay, specified as a finite positive scalar. This parameter applies when you set **Filter specification** to `Coefficients` and select **Make the first branch a pure delay**. The default is `1`.

Branch 1 allpass polynomial coefficients

Allpass polynomial filter coefficients of the first branch, specified as an *N*-by-`1` or *N*-by-`2` matrix of *N* first-order or second-order allpass sections. This parameter applies only when you set **Filter specification** to `Coefficients` and **Internal allpass structure** to `Minimum multiplier`. The default is [`0.1284563; 0.7906755`].

Branch 2 allpass polynomial coefficients

Allpass polynomial filter coefficients of the second branch, specified as an *N*-by-`1` or *N*-by-`2` matrix of *N* first-order or second-order allpass sections. This parameter applies only when you set **Filter specification** to `Coefficients` and **Internal allpass structure** to `Minimum multiplier`. The default is `0.4295667`.

Branch 1 Wave Digital coefficients

Allpass filter coefficients of the first branch in Wave Digital Filter form, specified as an *N*-by-`1` or *N*-by-`2` matrix of *N* first-order or second-order allpass sections. This parameter applies only when you set **Filter specification** to `Coefficients` and **Internal allpass structure** to `Wave Digital Filter`. The default is `[0.1284563; 0.7906755]`.

Branch 2 Wave Digital coefficients

Allpass filter coefficients of the second branch in Wave Digital Filter form, specified as an *N*-by-1 or *N*-by-2 matrix of *N* first-order or second-order allpass sections. This parameter applies only when you set **Filter specification** to `Coefficients` and **Internal allpass structure** to `Wave Digital Filter`. The default is `0.4295667`.

Last section of branch 2 is first order

When you select this check box, the last section of the second branch is treated as a first order section. This parameter applies only when you set **Filter specification** to `Coefficients`. When the coefficients of the second branch are in an *N*-by-2 matrix, the block ignores the second element of the last row of the matrix. The last section of the second branch then becomes a first-order section.

When this check box is cleared, the last section of the second branch is treated as a second-order section. When the coefficients of the second branch are in an *N*-by-1 matrix, the block ignores this parameter.

By default, this check box is cleared.

Input highpass subband

When you select this check box, the block acts as a synthesis filter bank. The block accepts two inputs to synthesize: lowpass and highpass subbands. When you clear this check box, the block acts as an IIR half band interpolator and accepts a single vector or matrix as input. By default, this check box is cleared.

Inherit sample rate from input

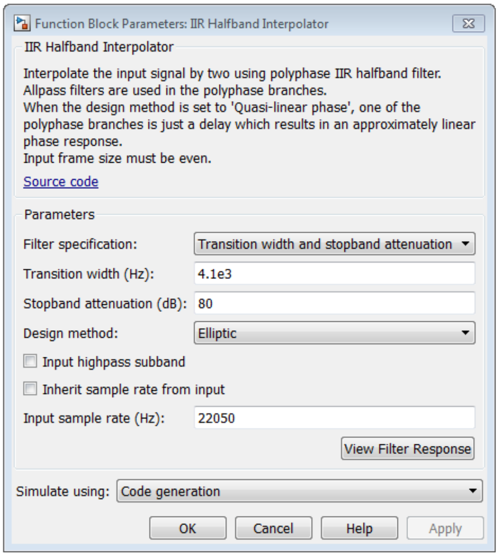
When you select this check box, the block inherits its sample rate from the input signal. The block calculates the sample rate based on the sample time of the input port. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**.

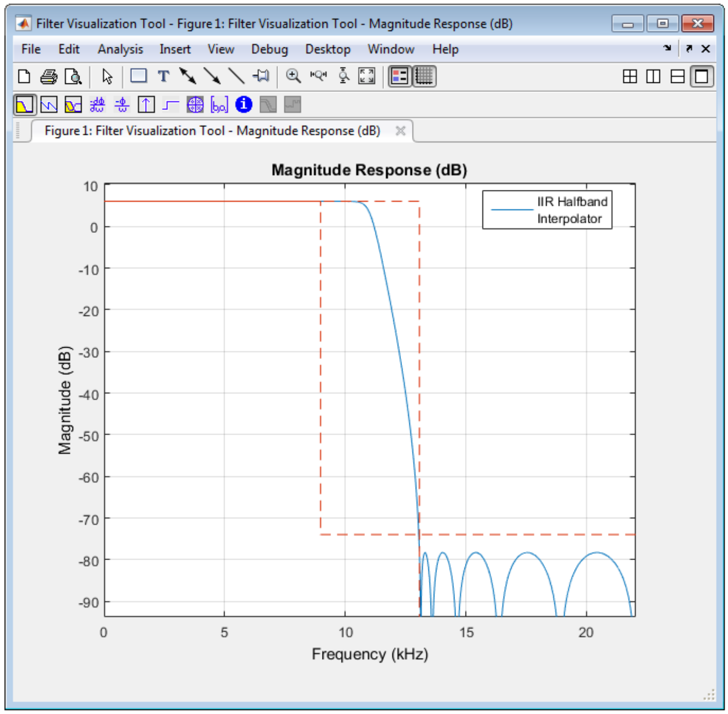
Input sample rate (Hz)

Input sample rate, specified as a scalar in Hz. The default is `44100`. You can specify an input sample rate when the **Inherit sample rate from input** check box is cleared.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the IIR Halfband Interpolator. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.





To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

See Also

dsp.IIRHalfbandInterpolator	DSP System Toolbox
dsp.IIRHalfbandDecimator	DSP System Toolbox
`IIR Halfband Decimator`	DSP System Toolbox
`FIR Halfband Interpolator`	DSP System Toolbox
`FIR Halfband Decimator`	DSP System Toolbox

Algorithm

This block brings the capabilities of the dsp.IIRHalfbandInterpolator System object to the Simulink environment.

For information on the algorithms used by this block, see the Algorithms on page 4-1043 section of `dsp.IIRHalfbandInterpolator`.

Introduced in R2015b

IIR Halfband Decimator

Decimate signal using polyphase IIR halfband filter

Library

Filtering/Filter Designs

`dspfdesign`

Description

The IIR Halfband Decimator block performs polyphase decimation of the input signal by a factor of two. To design the halfband filter, you can specify the block to use an elliptic design or a quasi-linear phase design. The block uses these design methods to compute the filter coefficients. To filter the inputs, the block uses a polyphase structure. The allpass filters in the polyphase structure are in a minimum multiplier form.

Elliptic design introduces nonlinear phase and creates the filter using fewer coefficients than quasi linear design. Quasi-linear phase design overcomes phase nonlinearity at the cost of additional coefficients.

Alternatively, instead of designing the halfband filter using a design method, you can specify the filter coefficients directly. When you choose this option, the allpass filters in the two branches of the polyphase implementation can be in a minimum multiplier form or in a wave digital form.

You can also use the block to implement the analysis portion of a two-band filter bank to filter a signal into lowpass and highpass subbands.

The input signal can be a real- or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The number of rows in the input signal must be a multiple of 2.

Parameters

Filter specification

Parameters used to design the IIR halfband filter. Because the filter design has only two degrees of freedom, you can specify only two of the three parameters:

- `Transition width and stopband attenuation` (default) — Design the filter using **Transition width (Hz)** and **Stopband attenuation (dB)**. This design is the minimum order design.
- `Filter order and transition width` — Design the filter using **Filter order** and **Transition width (Hz)**.
- `Filter order and stopband attenuation` — Design the filter using **Filter order** and **Stopband attenuation (dB)**.
- `Coefficients`— Specify the filter coefficients directly using the enabled parameters.

Transition width (Hz)

Transition width of the IIR halfband filter, specified as a real positive scalar in Hz. The transition width must be less than half the input sample rate. This parameter applies when **Filter specification** is set to `Filter order and transition width` or `Transition width and stopband attenuation`. The default is `4.1e3`.

Filter order

Filter order, specified as a finite positive integer. If you set **Design method** to `Elliptic`, then **Filter order** must be an odd integer greater than one. If you set **Design method** to `Quasi-linear phase`, then **Filter order** must be a multiple of four. This parameter applies when **Filter specification** is set to `Filter order and transition width` or `Filter order and stopband attenuation`. The default is `9`.

Stopband attenuation (dB)

Minimum attenuation needed in the stopband of the IIR halfband filter, specified as a real positive scalar in dB. This parameter applies when **Filter specification** is set to `Filter order and stopband attenuation` or `Transition width and stopband attenuation`. The default is `80`.

Design method

Design method for the IIR halfband filter.

- `Elliptic` (default) — The filter has nonlinear phase and uses few coefficients.
- `Quasi-linear phase` — The first branch of the polyphase filter structure is a pure delay, which results in an approximately linear phase response.

This parameter applies when you set **Filter specification** to any option except `Coefficients`.

Internal allpass structure

Internal allpass filter implementation structure, specified as `Minimum multiplier` or `Wave Digital Filter`. This parameter applies when you set **Filter specification** to `Coefficients`. Each structure uses a different coefficients set, independently stored in the corresponding coefficients property. The default is `Minimum multiplier`.

Make the first branch a pure delay

When you select this check box, the first branch of the polyphase filter structure becomes a pure delay, and the **Branch 1 allpass polynomial coefficients** and **Branch 1 Wave Digital coefficients** parameters do not apply. This parameter applies when you set **Filter specification** to `Coefficients`.

By default, this check box is selected.

Delay length in samples for branch 1

Length of the first branch delay, specified as a finite positive scalar. This parameter applies when you set **Filter specification** to `Coefficients` and select **Make the first branch a pure delay**. The default is `1`.

Branch 1 allpass polynomial coefficients

Allpass polynomial filter coefficients of the first branch, specified as an *N*-by-`1` or *N*-by-`2` matrix of *N* first-order or second-order allpass sections. This parameter applies only when you set **Filter specification** to `Coefficients` and **Internal allpass structure** to `Minimum multiplier`. The default is [`0.1284563; 0.7906755`].

Branch 2 allpass polynomial coefficients

Allpass polynomial filter coefficients of the second branch, specified as an *N*-by-`1` or *N*-by-`2` matrix of *N* first-order or second-order allpass sections. This parameter applies only when you set **Filter specification** to `Coefficients` and **Internal allpass structure** to `Minimum multiplier`. The default is `0.4295667`.

Branch 1 Wave Digital coefficients

Allpass filter coefficients of the first branch in Wave Digital Filter form, specified as an *N*-by-`1` or *N*-by-`2` matrix of *N* first-order or second-order allpass sections. This parameter applies only when you set **Filter specification** to `Coefficients` and **Internal allpass structure** to `Wave Digital Filter`. The default is [`0.1284563; 0.7906755`].

Branch 2 Wave Digital coefficients

Allpass filter coefficients of the second branch in Wave Digital Filter form, specified as an *N*-by-1 or *N*-by-2 matrix of *N* first-order or second-order allpass sections. This parameter applies only when you set **Filter specification** to `Coefficients` and **Internal allpass structure** to `Wave Digital Filter`. The default is `0.4295667`.

Last section of branch 2 is first order

When you select this check box, the last section of the second branch is treated as a first order section. This parameter applies only when you set **Filter specification** to `Coefficients`. When the coefficients of the second branch are in an *N*-by-2 matrix, the block ignores the second element of the last row of the matrix. The last section of the second branch then becomes a first-order section.

When this check box is cleared, the last section of the second branch is treated as a second-order section. When the coefficients of the second branch are in an *N*-by-1 matrix, the block ignores this parameter.

By default, this check box is cleared.

Output highpass subband

When you select this check box, the block acts as an analysis filter bank, producing two power-complementary outputs. When you clear this check box, the block acts as an IIR halfband decimator and accepts a single vector or matrix as input. By default, this check box is cleared.

Inherit sample rate from input

When you select this check box, the block inherits its sample rate from the input signal. The block calculates the sample rate based on the sample time of the input port. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**.

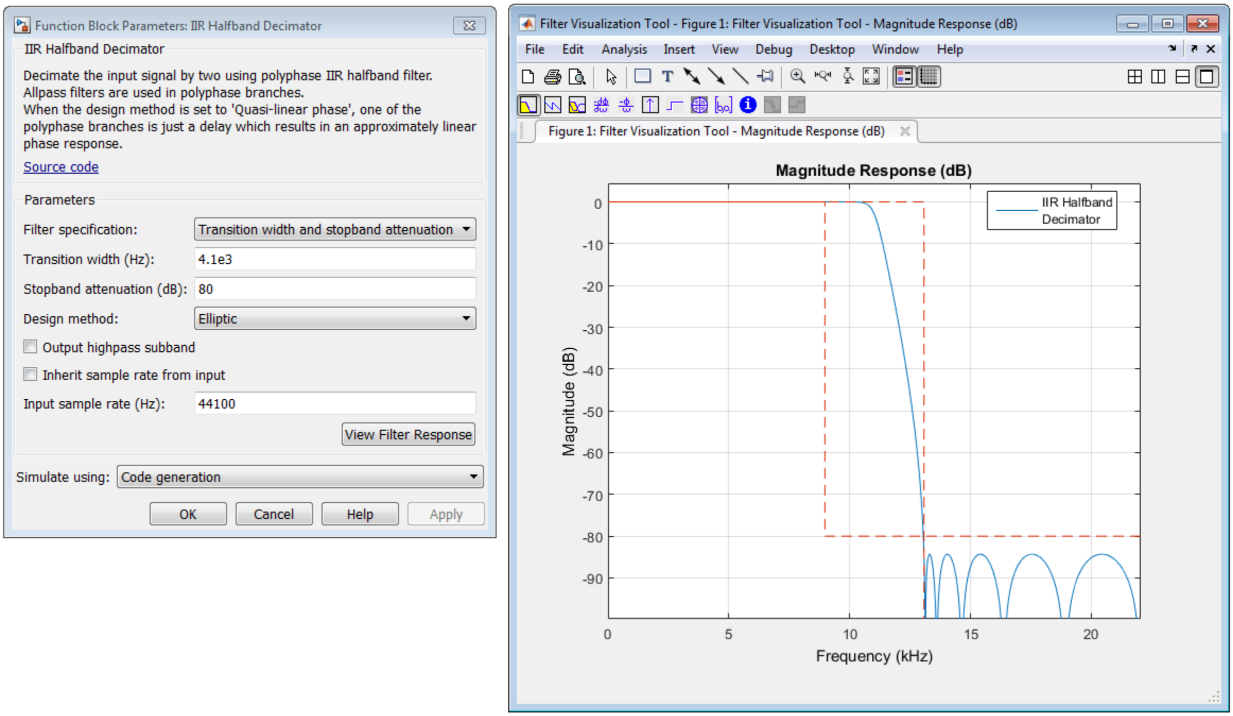
This parameter applies when you set **Filter specification** to any coefficients except `Coefficients`.

Input sample rate (Hz)

Input sample rate, specified as a scalar in Hz. The default is `44100`. You can specify an input sample rate when the **Inherit sample rate from input** check box is cleared.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the IIR Halfband Decimator. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

See Also

dsp.IIRHalfbandInterpolator	DSP System Toolbox
dsp.IIRHalfbandDecimator	DSP System Toolbox
`IIR Halfband Interpolator`	DSP System Toolbox
`FIR Halfband Interpolator`	DSP System Toolbox
`FIR Halfband Decimator`	DSP System Toolbox

Algorithm

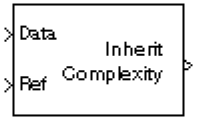
This block brings the capabilities of the dsp.IIRHalfbandDecimator System object to the Simulink environment.

For information on the algorithms used by this block, see the Algorithms on page 4-1022 section of `dsp.IIRHalfbandDecimator`.

Introduced in R2015b

Inherit Complexity

Change complexity of input to match reference signal



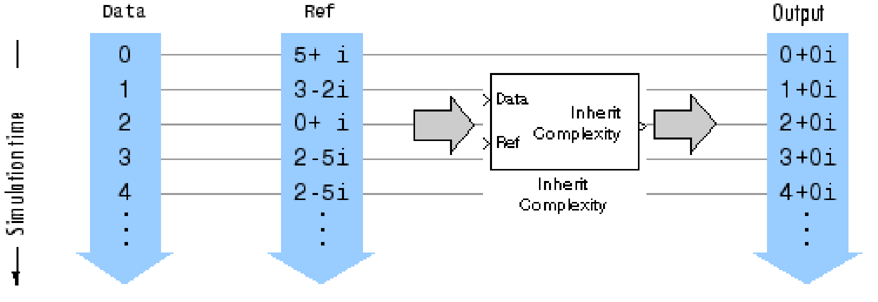
Library

Signal Management / Signal Attributes

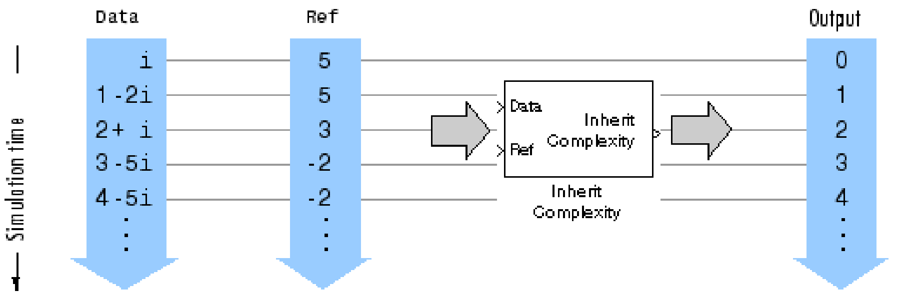
`dspsigattribs`

Description

The Inherit Complexity block alters the input data at the Data port to match the complexity of the reference input at the Ref port. When the Data input is real, and the Ref input is complex, the block appends a zero-valued imaginary component, `0i`, to each element of the Data input.



When the `Data` input is complex, and the `Ref` input is real, the block outputs the real component of the `Data` input.



When both the `Data` input and `Ref` input are real, or when both the `Data` input and `Ref` input are complex, the block propagates the `Data` input with no change.

Supported Data Types

Port	Supported Data Types
Data	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Ref	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
	• 8-, 16-, and 32-bit unsigned integers

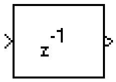
See Also

`Check Signal Attributes`	DSP System Toolbox
Complex to Magnitude-Angle	Simulink
Complex to Real-Imag	Simulink
Magnitude-Angle to Complex	Simulink
Real-Imag to Complex	Simulink

Introduced before R2006a

Integer Delay (Obsolete)

Delay input by integer number of sample periods



Library

`dspobslib`

Description

Note: The Integer Delay block will be removed from the product in a future release. We strongly recommend replacing this block with the `Delay` block.

The Integer Delay block delays a discrete-time input by the number of sample intervals specified in the **Delay** parameter. Noninteger delay values are rounded to the nearest integer, and negative delays are clipped at 0.

Sample-Based Operation

When the input is a sample-based M-by-N matrix, the block treats each of the $M*N$ matrix elements as an independent channel. The **Delay** parameter, v, can be an M-by-N matrix of positive integers that specifies the number of sample intervals to delay each channel of the input, or a scalar integer by which to equally delay all channels.

For example, when the input is M-by-1 and v is the matrix `[v(1) v(2) ... v(M)]'`, the first channel is delayed by `v(1)` sample intervals, the second channel is delayed by `v(2)` sample intervals, and so on. Note that when a channel is delayed for Δ sample-time units, the output sample at time t is the input sample at time $t - \Delta$. When $t - \Delta$ is negative, then the output is the corresponding value specified by the **Initial conditions** parameter.

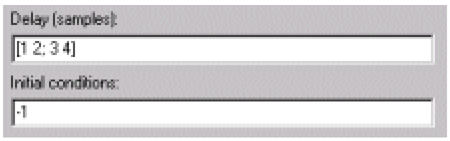
A 1-D vector of length M is treated as an M-by-1 matrix, and the output is 1-D.

The **Initial conditions** parameter specifies the output of the block during the initial delay in each channel. The initial delay for a particular channel is the time elapsed from the start of the simulation until the first input in that channel is propagated to the output. Both fixed and time-varying initial conditions can be specified in a variety of ways to suit the dimensions of the input.

Fixed Initial Conditions

A fixed initial condition in sample-based mode can be specified as one of the following:

- Scalar value to be repeated at each sample time of the initial delay (for every channel). For a 2-by-2 input with the parameter settings below,

  the block generates the following sequence of matrices at the start of the simulation,

  $$\begin{bmatrix} -1 & -1 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^1 & -1 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^2 & u_{12}^1 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^3 & u_{12}^2 \\ u_{21}^1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^4 & u_{12}^3 \\ u_{21}^2 & u_{22}^1 \end{bmatrix}, \ldots$$

  where u_{ij}^k is the *i,j*th element of the *k*th matrix in the input sequence.

- Array of size *M*-by-*N*-by-*d*. In this case, you can set different fixed initial conditions for each element of a sample-based input. This setting is explained further in the Array bullet in "Time-Varying Initial Conditions" on page 2-984.

Initial conditions cannot be specified by full matrices.

Time-Varying Initial Conditions

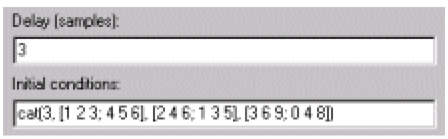
A time-varying initial condition in sample-based mode can be specified in one of the following ways:

- Vector of length *d*, where *d* is the maximum value specified for any channel in the **Delay** parameter. The vector can be a *L*-by-*d*, 1-by-*d*, or 1-by-1-by-*d*. The d elements of the vector are output in sequence, one at each sample time of the initial delay.

For a scalar input and the parameters shown below,

the block outputs the sequence `-1, -1, -1, 0, 1,...` at the start of the simulation.

- Array of dimension M-by-N-by-d, where d is the value specified for the **Delay** parameter (the maximum value when the **Delay** is a vector) and M and N are the number of rows and columns, respectively, in the input matrix. The d pages of the array are output in sequence, one at each sample time of the initial delay. For a 2-by-3 input, and the parameters below,



the block outputs the matrix sequence

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \begin{bmatrix} 2 & 4 & 6 \\ 1 & 3 & 5 \end{bmatrix}, \begin{bmatrix} 3 & 6 & 9 \\ 0 & 4 & 8 \end{bmatrix}$$

at the start of the simulation. Note that setting **Initial conditions** to an array with the same matrix for each entry implements constant initial conditions; a different constant initial condition for each input matrix element (channel).

Initial conditions cannot be specified by full matrices.

Frame-Based Operation

When the input is a frame-based M-by-N matrix, the block treats each of the N columns as an independent channel, and delays each channel as specified by the **Delay** parameter.

For frame-based inputs, the **Delay** parameter can be a scalar integer by which to equally delay all channels. It can also be a 1-by-N row vector, each element of which serves as the delay for the corresponding channel of the N-channel input. Likewise, it can also be an M-by-1 column vector, each element of which serves as the delay for one of the corresponding M samples for each channel. The **Delay** parameter can be an M-by-N matrix of positive integers as well; in this case, each element of each channel is delayed by the corresponding element in the delay matrix. For instance, if the fifth element of the third column of the delay matrix was 3, then the fifth element of the third channel of the input matrix is always delayed by three sample-time units.

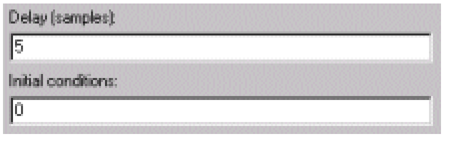
When a channel is delayed for Δ sample-time units, the output sample at time t is the input sample at time $t - \Delta$. When $t - \Delta$ is negative, then the output is the corresponding value specified in the **Initial conditions** parameter.

The **Initial conditions** parameter specifies the output during the initial delay. Both fixed and time-varying initial conditions can be specified. The initial delay for a particular channel is the time elapsed from the start of the simulation until the first input in that channel is propagated to the output.

Fixed Initial Conditions

The settings shown below specify fixed initial conditions. The value entered in the **Initial conditions** parameter is repeated at the output for each sample time of the initial delay. A fixed initial condition in frame-based mode can be one of the following:

- Scalar value to be repeated for all channels of the output at each sample time of the initial delay. For a general M-by-N input with the parameter settings below,

  the first five samples in each of the N channels are zero. Notice that when the frame size is larger than the delay, all of these zeros are all included in the first output from the block.
- Array of size 1-by-N-by-D. In this case, you can also specify different fixed initial conditions for each channel. See “Time-Varying Initial Conditions” on page 2-987 for details.

Initial conditions cannot be specified by full matrices.

Time-Varying Initial Conditions

The following settings specify time-varying initial conditions. For time-varying initial conditions, the values specified in the **Initial conditions** parameter are output in sequence during the initial delay. A time-varying initial condition in frame-based mode can be specified in the following ways:

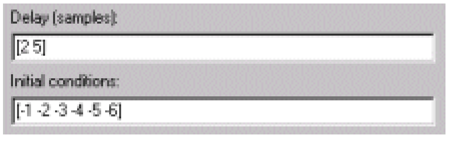
- Vector of length D, where each of the N channels have the same initial conditions sequence specified in the vector. D is defined as follows:
  - When an element of the delay entry is less than the frame size,

    $D = d + 1$

    where d is the maximum delay.
  - When the all elements of the delay entry are greater than the input frame size,

    $D = d +$ `input frame size` $- 1$

  Only the first d entries of the initial condition vector are used; the rest of the values are ignored, but you must include them nonetheless. For a two-channel ramp input `[1:100; 1:100]'` with a frame size of 4 and the parameter settings below,

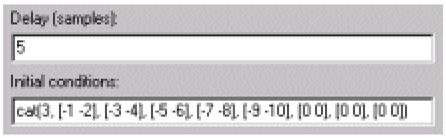
  

  the block outputs the following sequence of frames at the start of the simulation.

  $$\begin{bmatrix} -4 & -1 \\ -5 & -2 \\ 1 & -3 \\ 2 & -4 \end{bmatrix}, \begin{bmatrix} 3 & -5 \\ 4 & 1 \\ 5 & 2 \\ 6 & 3 \end{bmatrix}, \begin{bmatrix} 7 & 4 \\ 8 & 5 \\ 9 & 6 \\ 10 & 7 \end{bmatrix}, \ldots$$

  Note that since one of the delays, 2, is less than the frame size of the input, 4, the length of the **Initial conditions** vector is the sum of the maximum delay and 1 (5+1), which is 6. The first five entries of the initial conditions vector are used by the channel with the maximum delay, and the rest of the entries are ignored. Since the first channel is delayed for less than the maximum delay (2 sample time units), it only makes use of two of the initial condition entries.

- Array of size 1-by-N-by-D, where D is defined in "Time-Varying Initial Conditions" on page 2-987. In this case, the kth entry of each 1-by-N entry in the array corresponds to an initial condition for the kth channel of the input matrix. Thus, a 1-by-N-by-D initial conditions input allows you to specify different initial conditions for each channel. For instance, for a two-channel ramp input `[1:100; 1:100]'` with a frame size of 4 and the parameter settings below,



the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -1 & -2 \\ -3 & -4 \\ -5 & -6 \\ -7 & -8 \end{bmatrix}, \begin{bmatrix} -9 & -10 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \ldots$$

Note that the channels have distinct time varying initial conditions; the initial conditions for channel 1 correspond to the first entry of each length-2 row vector in the initial conditions array, and the initial conditions for channel 2 correspond to the second entry of each row vector in the initial conditions array. Only the first five entries in the initial conditions array are used; the rest are ignored.

The 1-by-N-by-D array entry can also specify different fixed initial conditions for every channel; in this case, every 1-by-N entry in the array would be identical, so that the initial conditions for each column are fixed over time.

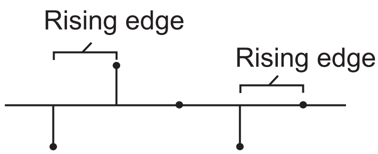
Initial conditions cannot be specified by full matrices.

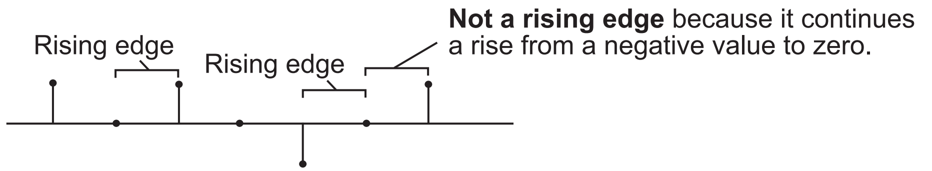
Resetting the Delay

The block resets the delay whenever it detects a reset event at the optional Rst port. The reset sample time must be a positive integer multiple of the input sample time.

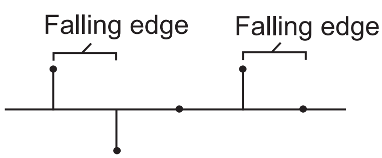
You specify the reset event in the **Reset port** parameter:

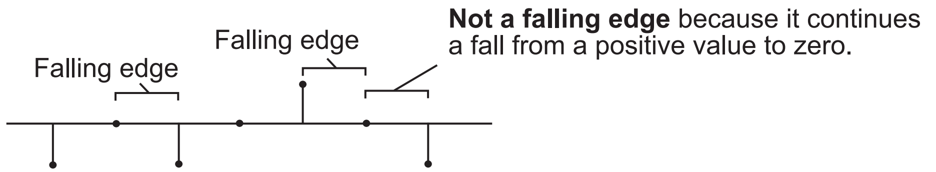
- `None` disables the Rst port.
- `Rising edge` — Triggers a reset operation when the Rst input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)





- `Falling edge` — Triggers a reset operation when the `Rst` input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)





- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

Examples

The dspafxr example illustrates an audio reverberation system built around the Integer Delay block.

Parameters

Delay

The number of sample periods to delay the input signal.

Initial conditions

The value of the block's output during the initial delay.

Reset port

Determines the reset event that causes the block to reset the delay. For more information, see “Resetting the Delay” on page 2-988.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)

- Boolean — The block accepts Boolean inputs to the Rst port, which is enabled by the **Reset port** parameter.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

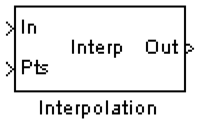
See Also

Unit Delay	Simulink
`Variable Fractional Delay`	DSP System Toolbox
Variable Integer Delay	Simulink

Introduced in R2008b

Interpolation

Interpolate values of real input samples



Library

Signal Operations

`dspsigops`

Description

The Interpolation block interpolates discrete, real, inputs using linear or FIR interpolation. The block accepts both sample- and frame-based input data in the form of a vector, matrix, or sample-based N-D array. The block outputs a scalar, vector, matrix, or N-D array of the interpolated values, which has the same frame status as the input data.

You must specify the interpolation points (times at which to interpolate values) in a one-based interpolation array, I_{Pts}. An entry of 1 in I_{Pts} refers to the first sample of the input data, an entry of 2.5 refers to the sample half-way between the second and third input sample, and so on. Depending on the dimensions of the input data, I_{Pts} can be a scalar, a length-P row or column vector, a P-by-N frame-based matrix, or a sample-based N-D array where P is the size of the first dimension of the N-D array. In most cases, P can be any positive integer. For more information about valid interpolation arrays, refer to the tables in "How the Block Applies Interpolation Arrays to Inputs" on page 2-993.

In most cases, the block applies I_{Pts} across the first dimension of an N-D input array, or to each input vector. You can set the block to apply the same interpolation array for all input data (static interpolation points entered on the block mask) or to use a different interpolation array for each N-D array, matrix, or vector input (time-varying interpolation points received via the Pts input port).

Sections of This Reference Page

- "Specifying Static Interpolation Points" on page 2-993
- "Specifying Time-Varying Interpolation Points" on page 2-993
- "How the Block Applies Interpolation Arrays to Inputs" on page 2-993
- "Handling Out-of-Range Interpolation Points" on page 2-997
- "Linear Interpolation Mode" on page 2-998
- "FIR Interpolation Mode" on page 2-999
- "Parameters" on page 2-1000
- "Supported Data Types" on page 2-1001

Specifying Static Interpolation Points

To supply the block with a static interpolation array (an interpolation array applied to every vector or N-D array of input data), perform the following steps:

- Set the **Source of interpolation points** parameter to `Specify via dialog`.
- Enter the interpolation array in the **Interpolation points** parameter. To learn about interpolation arrays, see "How the Block Applies Interpolation Arrays to Inputs" on page 2-993.

Specifying Time-Varying Interpolation Points

To supply the block with time-varying interpolation arrays (where the block uses a different interpolation array for each vector or N-D array input), perform the following steps:

1. Set the **Source of interpolation points** parameter to `Input port`, the **Pts** port appears on the block.
2. Generate a signal of interpolation arrays, and supply it to the Pts port. The block uses the input to this port as the interpolation points. To learn about interpolation arrays, see "How the Block Applies Interpolation Arrays to Inputs" on page 2-993.

How the Block Applies Interpolation Arrays to Inputs

The interpolation array I_{Pts} represents the points in time at which to interpolate values of the input signal. An entry of 1 in I_{Pts} refers to the first sample of the input, an entry of

2.5 refers to the sample half-way between the second and third input sample, and so on. In most cases, when I_{Pts} is a vector, it can be of any length.

Valid values in the interpolation array, I_{Pts}, range from 1 to the number of samples in each channel of the input. To learn how the block handles out of range interpolation values, see “Handling Out-of-Range Interpolation Points” on page 2-997.

Depending on the dimension and frame status of the input and the dimension of I_{Pts}, the block usually applies I_{Pts} to the input in one of the following ways:

- Applies the I_{Pts} array across the first dimension of a sample-based N-D array or frame-based matrix input, resulting in a sample-based N-D array or frame-based matrix output.
- Applies the vector I_{Pts} to each input vector (as if the input vector were a single channel), resulting in a vector output with the same orientation as the input (row or column).

The following tables summarize how the block applies the interpolation array I_{Pts} to all the possible types of sample- and frame-based inputs, and show the resulting output dimensions.

The first table describes the block's behavior when the **Source of interpolation points** is `Specify via dialog` and the input is sample based.

Input Dimensions (Sample Based)	Valid Dimensions of Interpolation Array I_{Pts}	How Block Applies I_{Pts} to Input	Output Dimensions (Sample Based)
N-D Array (ex. M-by-N-by-Q)	1-by-P row	Applies I_{Pts} to the first dimension of the input array	P-by-N-by-Q array
	P-by-1 column		
	P-by-N-by-Q array	Applies the columns of I_{Pts} to the corresponding columns of the input array	
M-by-1 column	1-by-P row (block treats I_n as a column)	Applies I_{Pts} to the input column	P-by-1 column
	P-by-1 column		

Input Dimensions (Sample Based)	Valid Dimensions of Interpolation Array I_{Pts}	How Block Applies I_{Pts} to Input	Output Dimensions (Sample Based)
1-by-N row	1-by-P row	Applies I_n to the input row	1-by-P row
	P-by-1 column (block treats I_{Pts} as a row)		

The next table describes the block's behavior when the **Source of interpolation points** is `Specify via dialog` and the input is frame based.

Input Dimensions (Frame Based)	Valid Dimensions of Interpolation Array I_{Pts}	How Block Applies I_{Pts} to Input	Output Dimensions (Frame Based)
M-by-N matrix	1-by-N row	Applies each column of I_{Pts} (each element of I_{Pts}) to the corresponding column of the input matrix	1-by-N row
	P-by-1 column	Applies I_{Pts} to each input column	P-by-N matrix
	P-by-N matrix	Applies the columns of I_{Pts} to the corresponding columns of the input matrix	
M-by-1 column	1-by-P row (block treats I_{Pts} as a column)	Applies I_{Pts} to the input column	P-by-1 column
	P-by-1 column		
1-by-N row (not recommended)	1-by-N row	Not Applicable. Block copies input vector	1-by-N row, a copy of the input vector
	P-by-1 column		P-by-N matrix where each row is a copy of the input vector
	P-by-N matrix		

The next table describes the block's behavior when the **Source of interpolation points** is `Input port` and the input is sample based.

Input Dimensions (Sample Based)	Valid Dimensions of Interpolation Array I_{Pts}	How Block Applies I_{Pts} to Input	Output Dimensions (Sample Based)
N-D Array (ex. M-by-N-by-Q)	Sample-based 1-by-P row	Applies I_{Pts} to the first dimension of the input array	P-by-N-by-Q array
	Sample- or frame-based P-by-1 column		
	Sample-based P-by-N-by-Q array	Applies the columns of I_{Pts} to the corresponding columns of the input array	
M-by-1 column	Sample-based 1-by-P row	Applies I_{Pts} to the input column	P-by-1 column
	Sample- or frame-based P-by-1 column		
1-by-N row	Sample-based 1-by-P row	Applies I_{Pts} to the input row	1-by-P row
	Sample or frame-based P-by-1 column		

The next table describes the block's behavior when the **Source of interpolation points** is `Input port` and the input is frame based.

Input Dimensions (Frame Based)	Valid Dimensions of Interpolation Array I_{Pts}	How Block Applies I_{Pts} to Input	Output Dimensions (Frame Based)
M-by-N matrix	Frame-based 1-by-N row	Applies each column of I_{Pts} (each element of I_{Pts}) to the corresponding column of the input matrix	1-by-N row
	Sample-based 1-by-P row	Applies I_{Pts} to each input column	P-by-N matrix

Input Dimensions (Frame Based)	Valid Dimensions of Interpolation Array I_{Pts}	How Block Applies I_{Pts} to Input	Output Dimensions (Frame Based)
	Frame- or sample-based P-by-1 column		
	Frame- or sample-based P-by-N matrix	Applies the columns of I_{Pts} to the corresponding columns of the input matrix	
M-by-1 column	Sample-based 1-by-P row	Applies I_{Pts} to the input column	P-by-1 column
	Frame- or sample-based P-by-1 column		
1-by-N row (not recommended)	Frame-based 1-by-N row	Not Applicable. Block copies input vector	1-by-N row, a copy of the input vector
	Sample-based 1-by-P row		P-by-N matrix where each row is a copy of the input vector
	Frame- or sample-based P-by-1 column		
	Frame- or sample-based P-by-N matrix		

Handling Out-of-Range Interpolation Points

Valid values in the interpolation array I_{Pts} range from 1 to the number of samples in each channel of the input. For instance, given a length-5 input vector `D`, all entries of I_{Pts} must range from 1 to 5. I_{Pts} cannot contain entries such as 7 or -9, since there is no 7th or -9th entry in `D`.

The **Out of range interpolation points** parameter sets how the block handles interpolation points that are fall outside the valid range, and has the following settings:

- `Clip` — The block replaces any out-of-range values in I_{Pts} with the closest value in the valid range (from 1 to the number of input samples), and then proceeds with computations using the clipped version of I_{Pts}.

- `Clip and warn` — In addition to `Clip`, the block issues a warning at the MATLAB command line every time clipping occurs.
- `Error` — When the block encounters an out-of-range value in I_{Pts}, the simulation stops, and the block issues an error at the MATLAB command line.

Example of Clipping

Suppose the block is set to clip out-of-range interpolation points, and gets the following input vector and interpolation points:

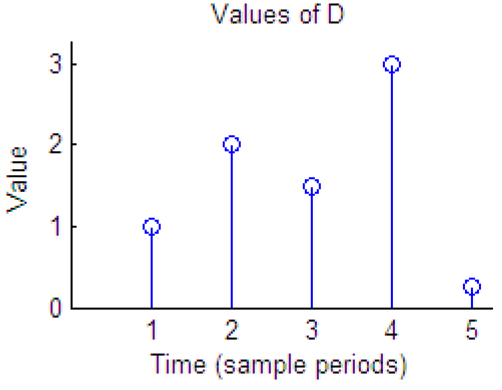
- `D = [11 22 33 44]'`
- $\mathtt{I_{Pts}}$ `= [10 2.6 -3]'`

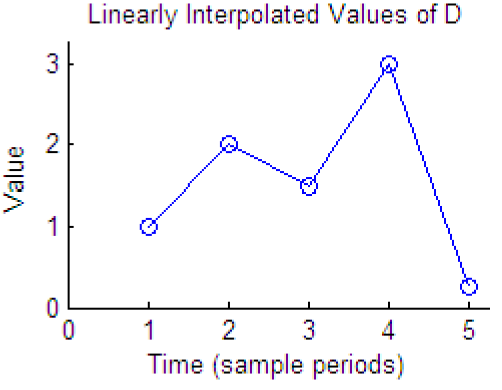
Because *D* has four samples, valid interpolation points range from 1 to 4. The block clips the interpolation point 10 to 4 and the point -3 to 1, resulting in the clipped interpolation vector $\mathtt{I_{PtsClipped}}$ `= [4 2.6 1]'`.

Linear Interpolation Mode

When **Interpolation Mode** is set to `Linear`, the block interpolates data values by assuming that the data varies linearly between samples taken at adjacent sample times.

For instance, if the input signal *D* = [1 2 1.5 3 0.25]', the following plot on the left shows the samples in *D*, and the plot on the right shows the linearly interpolated values between the samples in *D*.



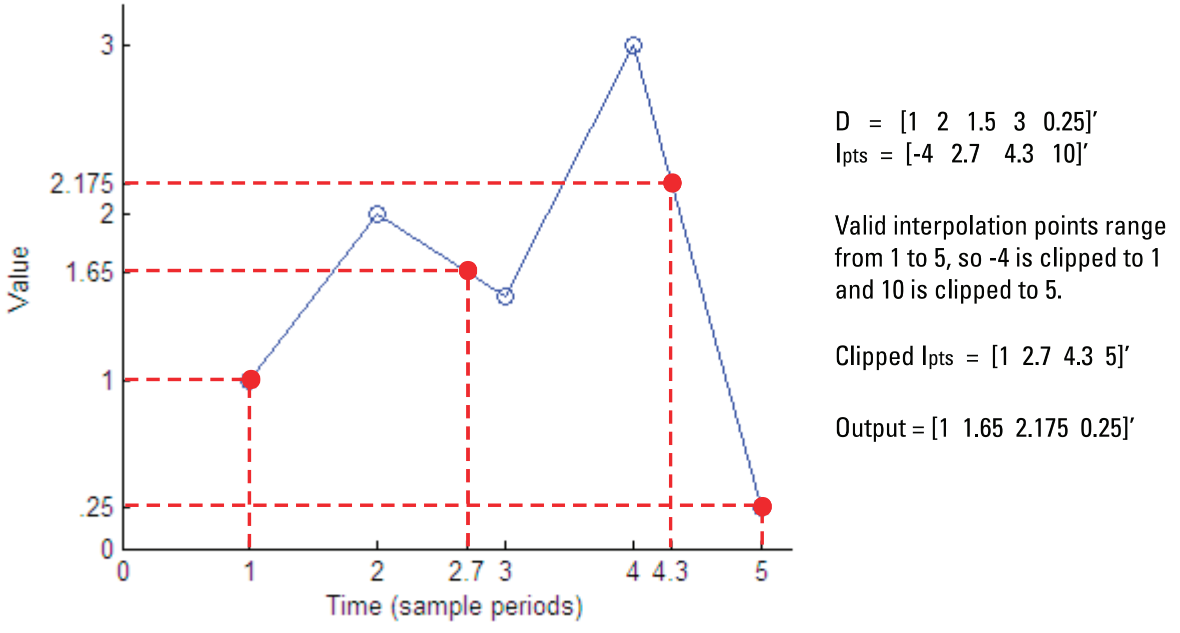


The following figure illustrates the case of a block in linear interpolation mode that is set to clip out-of-range interpolation points. The vector D supplies the input data and the vector I_{Pts} supplies the interpolation points:

- `D = [1 2 1.5 3 0.25]'`
- `I`$_{Pts}$` = [-4 2.7 4.3 10]'`

The block clips the invalid interpolation points, and outputs the linearly interpolated values in a vector, `[1 1.65 2.175 0.25]'`.

Interpolated Values of D at Clipped Interpolation Points



FIR Interpolation Mode

When **Interpolation Mode** is set to `FIR`, the block interpolates data values using an FIR interpolation filter, specified by various block parameters. See in the Variable Fractional Delay block reference for more information.

Parameters

Source of interpolation points

Choose how you want to specify the interpolation points. If you select `Specify via dialog`, the **Interpolation points** parameter become available. Use this option for static interpolation points. If you select `Input port`, the Pts port appears on the block. The block uses the input to this port as the interpolation points. Use this option for time-varying interpolation points. For more information, see "Specifying Static Interpolation Points" on page 2-993 and "Specifying Time-Varying Interpolation Points" on page 2-993.

Interpolation points

The array of points in time at which to interpolate the input signal (I_{Pts}). An entry of 1 in I_{Pts} refers to the first sample of the input, an entry of 2.5 refers to the sample half-way between the second and third input sample, and so on. See "How the Block Applies Interpolation Arrays to Inputs" on page 2-993. Tunable (Simulink).

Interpolation mode

Sets the block to interpolate by either linear or FIR interpolation. For more information, see "Linear Interpolation Mode" on page 2-998 and "FIR Interpolation Mode" on page 2-999.

Interpolation filter half-length

Specify the half-length of the FIR interpolation filter (`P`). To perform the interpolation in `FIR` mode, the block uses the nearest 2*`P` low-rate samples. In most cases, `P` low-rate samples must appear below and above each interpolation point. However, if you interpolate at a low-rate sample point, the block includes that low-rate sample in the required 2*`P` samples and requires only 2*`P`–1 neighboring low-rate samples. If an interpolation point does not have the required number of neighboring low-rate samples, the block interpolates that point using linear interpolation.

This parameter becomes available only when the **Interpolation mode** is set to `FIR`. For more information, see "FIR Interpolation Mode" on page 2-999.

Interpolation points per input sample

Also known as the *upsampling factor*, this parameter defines the number of points per input sample (`L`) at which the block computes a unique FIR interpolation filter. To perform the FIR Interpolation, the block uses a polyphase structure with `L` filter arms of length 2*`P`.

For example, if `L=4`, the block constructs a polyphase filter with four arms. The block then interpolates at points corresponding to 1 +*i*/**L**, 2 +*i*/**L**, 3 +*i*/**L**..., where the integers

1, 2, and 3 represent the low-rate samples, and *i*=`0,1,2,3`. To interpolate at a point that does not directly correspond to an arm of the polyphase filter requires an extra computation. The block first rounds that point down to the nearest value that does correspond to an arm of the polyphase filter. Thus, to interpolate at the point 2.2, the block rounds 2.2 down to 2, and computes the FIR interpolation using the first arm of the polyphase filter structure. Similarly, to interpolate the point 2.65, the block rounds the value down to 2.5 and uses the third arm of the polyphase filter structure.

This parameter becomes available only when the **Interpolation mode** is set to `FIR`. For more information, see "FIR Interpolation Mode" on page 2-999.

Normalized input bandwidth

The bandwidth of the input divided by Fs/2 (half the input sample frequency).

This parameter is only available when the **Interpolation mode** is set to `FIR`. For more information, see "FIR Interpolation Mode" on page 2-999.

Out of range interpolation points

When an interpolation point is out of range, this parameter sets the block to either clip the interpolation point, clip the value and issue a warning at the MATLAB command line, or stop the simulation and issue an error at the MATLAB command line. For more information, see "Handling Out-of-Range Interpolation Points" on page 2-997.

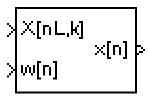
Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point
Pts	• Double-precision floating point • Single-precision floating point
Out	• Double-precision floating point • Single-precision floating point

Introduced before R2006a

Inverse Short-Time FFT

Recover time-domain signals by performing inverse short-time, fast Fourier transform (FFT)



Inverse
Short-Time FFT

Library

Transforms

`dspxfrm3`

Description

The Inverse Short-Time FFT block reconstructs the time-domain signal from the frequency-domain output of the Short-Time FFT block using a two-step process. First, the block performs the overlap add algorithm shown below.

$$x[n] = \frac{L}{W(0)} \sum_{p=-\infty}^{\infty} \left[\frac{1}{N} \sum_{k=0}^{N-1} X[pL,k] e^{j2\pi kn/N} \right]$$

Then, the block rebuffers the signal in order to reconstruct the time-domain signal. Depending on the analysis window used by the Short-Time FFT block, the Inverse Short-Time FFT block might or might not achieve perfect reconstruction of the time domain signal.

Connect your complex-valued, single-channel or multichannel input signal to the X(n,k) port. The block accepts unoriented vector, column vector and matrix input. The block outputs the real or complex-valued, single-channel or multichannel inverse short-time FFT at port x(n).

Connect your complex-valued, single-channel analysis window to the w(n) port. When you select the **Assert if analysis window does not support perfect signal**

reconstruction check box, the block displays an error when the input signal cannot be perfectly reconstructed. The block uses the values you enter for the **Analysis window length (W)** and **Reconstruction error tolerance**, or maximum amount of allowable error in the reconstruction process, to determine if the signal can be perfectly reconstructed.

Examples

The dspstsa example illustrates how to use the Short-Time FFT and Inverse Short-Time FFT blocks to remove the background noise from a speech signal. To open the `dspstsa` model, type `dspstsa` in the MATLAB command prompt.

Parameters

Analysis window length

Enter the length of the analysis window. This parameter is visible when you select the **Assert if analysis window does not support perfect signal reconstruction** check box.

Overlap between consecutive STFFT frames (in samples)

Enter the number of samples of overlap for each frame of the Short-Time FFT block's input signal. This value should be the same as the **Overlap between consecutive windows (in samples)** parameter in the Short-Time FFT block parameters dialog.

Samples per output frame

Enter the desired frame size of the output signal.

Input is conjugate symmetric

Select this check box when the input to the block is both floating point and conjugate symmetric, and you want real-valued outputs. When you select this check box when the input is not conjugate symmetric, the output of the block is invalid. This parameter cannot be used for fixed-point signals.

Assert if analysis window does not support perfect signal reconstruction

Select this check box to display an error when the analysis window used by the Short-Time FFT block does not support perfect signal reconstruction.

Reconstruction error tolerance

Enter the amount of acceptable error in the reconstruction of the original signal. This parameter is visible when you select the **Assert if analysis window does not support perfect signal reconstruction** check box.

Supported Data Types

Port	Supported Data Types
X(n,k)	• Double-precision floating point • Single-precision floating point
w(n)	• Double-precision floating point • Single-precision floating point
x(n)	• Double-precision floating point • Single-precision floating point

References

[1] Quatieri, Thomas E. *Discrete-Time Speech Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 2001.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

When the input length is not a power of two, the executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

When the input length is a power of two, you can generate standalone C and C++ code from this block.

See Also

See Also

Functions
`pwelch`

Blocks
`Burg Method` | `Magnitude FFT` | `Periodogram` | `Short-Time FFT` | `Spectrum Analyzer` | `Window Function` | `Yule-Walker Method`

Introduced before R2006a

Inverse Sinc Filter

Design inverse sinc filter

Library

Filtering / Filter Designs

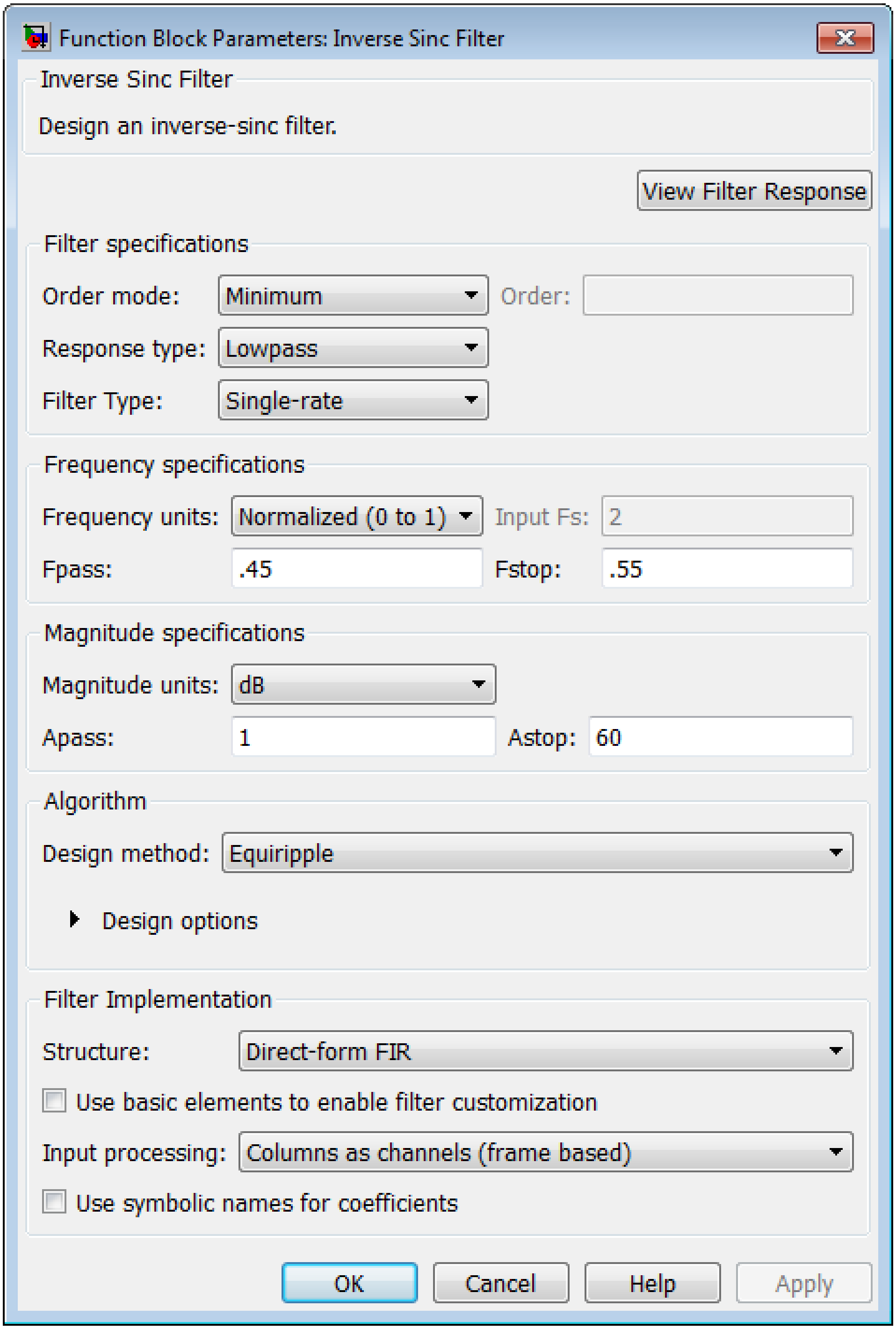
`dspfdesign`

Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See "Inverse Sinc Filter Design — Main Pane" on page 5-766 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.


Function Block Parameters: Inverse Sinc Filter
Inverse Sinc Filter
Design an inverse-sinc filter.
View Filter Response
Filter specifications
Order mode:
Minimum
Order:
Response type:
Lowpass
Filter Type:
Single-rate
Frequency specifications
Frequency units:
Normalized (0 to 1)
Input Fs:
2
Fpass:
.45
Fstop:
.55
Magnitude specifications
Magnitude units:
dB
Apass:
1
Astop:
60
Algorithm
Design method:
Equiripple
Design options
Filter Implementation
Structure:
Direct-form FIR
Use basic elements to enable filter customization
Input processing:
Columns as channels (frame based)
Use symbolic names for coefficients
OK
Cancel
Help
Apply

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Response type

Select `Lowpass` or `Highpass` to design an inverse sinc lowpass or highpass filter.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

Order

Enter the filter order. This option is enabled only if you set the **Order mode** to `Specify`.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

Regions between specification values such as **Fpass** and **Fstop** represent transition regions where the filter response is not constrained.

Frequency constraints

When **Order mode** is `Specify`, select the filter features that the block uses to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- `Passband edges` — For IIR filters, define the filter by specifying frequencies for the edges of the passband.
- `Stopband edges` — For IIR filters, define the filter by specifying frequencies for the edges of the stopbands.
- `6 dB point` — For FIR filters, define the filter response by specifying the locations of the 6 dB point. The 6 dB point is the frequency for the point six decibels below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is

available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

F6dB

When **Frequency constraints** is `6 dB point`, specify the frequency of the 6 dB point. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default)
- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default FIR method is `Equiripple`.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Specify the phase constraint of the filter as `Linear`, `Maximum`, or `Minimum`.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options;

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set **Stopband shape**, **Stopband decay** specifies the amount of decay applied to the stopband. the following conditions apply to **Stopband decay** based on the value of **Stopband Shape**:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent *n* in the relation $(1/f)^n$ to define the stopband decay. The block applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Sinc frequency factor

A frequency dilation factor. The **Sinc frequency factor**, *C* , parameterizes the passband magnitude response for a lowpass design through H(ω) = sinc(Cω)^(-P) and through H(ω) = sinc(C(1-ω))^(-P) for a highpass design.

Sinc power

Negative power of passband magnitude response. The **Sinc power**, *P*, parameterizes the passband magnitude response for a lowpass design through H(ω) = sinc(Cω)^(-P) and through H(ω) = sinc(C(1-ω))^(-P) for a highpass design.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure.

Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

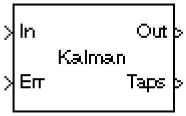
Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2006b

Kalman Adaptive Filter (Obsolete)

Compute filter estimates for inputs using Kalman adaptive filter algorithm



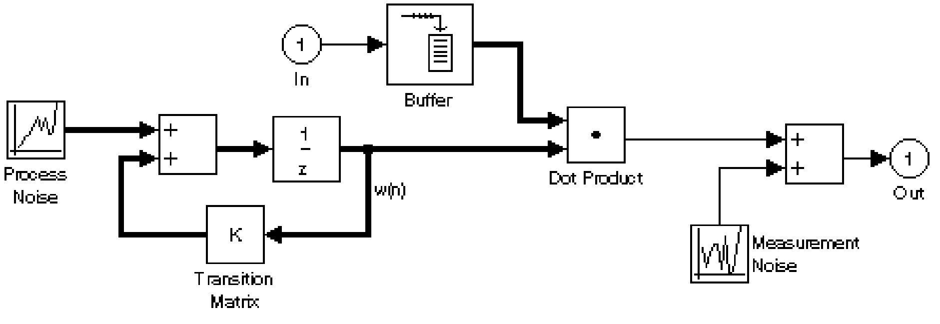
Library

`dspobslib`

Description

Note The Kalman Adaptive Filter block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the `Kalman Filter` block.

The Kalman Adaptive Filter block computes the optimal linear minimum mean-square estimate (MMSE) of the FIR filter coefficients using a one-step predictor algorithm. This Kalman filter algorithm is based on the following physical realization of a dynamic system.



The Kalman filter assumes that there are no deterministic changes to the filter taps over time (that is, the transition matrix is identity), and that the only observable output from the system is the filter output with additive noise. The corresponding Kalman filter is expressed in matrix form as

$$g(n) = \frac{K(n-1)u(n)}{u^H(n)K(n-1)u(n) + Q_M}$$
$$y(n) = u^H(n)\hat{w}(n)$$
$$e(n) = d(n) - y(n)$$
$$\hat{w}(n+1) = \hat{w}(n) + e(n)g(n)$$
$$K(n) = K(n-1) - g(n)u^H(n)K(n-1) + Q_p$$

The variables are as follows

Variable	Description
n	The current algorithm iteration
$u(n)$	The buffered input samples at step n
$K(n)$	The correlation matrix of the state estimation error
$g(n)$	The vector of Kalman gains at step n
$\hat{w}(n)$	The vector of filter-tap estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
Q_M	The correlation matrix of the measurement noise
Q_P	The correlation matrix of the process noise

The correlation matrices, Q_M and Q_P, are specified in the parameter dialog by scalar variance terms to be placed along the matrix diagonals, thus ensuring that these matrices are symmetric. The filter algorithm based on this constraint is also known as the random-walk Kalman filter.

The implementation of the algorithm in the block is optimized by exploiting the symmetry of the input covariance matrix $K(n)$. This decreases the total number of computations by a factor of two.

The block icon has port labels corresponding to the inputs and outputs of the Kalman algorithm. Note that inputs to the In and Err ports must be sample-based scalars with the same complexity. The signal at the Out port is a scalar, while the signal at the Taps port is a sample-based vector.

Block Ports	Corresponding Variables
In	u, the scalar input, which is internally buffered into the vector $u(n)$
Out	$y(n)$, the filtered scalar output
Err	$e(n)$, the scalar estimation error
Taps	$\hat{w}(n)$, the vector of filter-tap estimates

An optional Adapt input port is added when you select the **Adapt port** check box in the dialog. When this port is enabled, the block continuously adapts the filter coefficients while the Adapt input is nonzero. A zero-valued input to the `Adapt` port causes the block to stop adapting, and to hold the filter coefficients at their current values until the next nonzero Adapt input.

The **FIR filter length** parameter specifies the length of the filter that the Kalman algorithm estimates. The **Measurement noise variance** and the **Process noise variance** parameters specify the correlation matrices of the measurement and process noise, respectively. The **Measurement noise variance** must be a scalar, while the **Process noise variance** can be a vector of values to be placed along the diagonal, or a scalar to be repeated for the diagonal elements.

The **Initial value of filter taps** specifies the initial value $\hat{w}(0)$ as a vector, or as a scalar to be repeated for all vector elements. The **Initial error correlation matrix** specifies the initial value $K(0)$, and can be a diagonal matrix, a vector of values to be placed along the diagonal, or a scalar to be repeated for the diagonal elements.

Parameters

FIR filter length

The length of the FIR filter.

Measurement noise variance

The value to appear along the diagonal of the measurement noise correlation matrix. Tunable (Simulink).

Process noise variance

The value to appear along the diagonal of the process noise correlation matrix. Tunable (Simulink).

Initial value of filter taps

The initial FIR filter coefficients.

Initial error correlation matrix

The initial value of the error correlation matrix.

Adapt port

Enables the `Adapt` port.

References

Haykin, S. *Adaptive Filter Theory*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1996.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

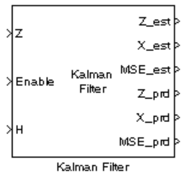
LMS Adaptive Filter (Obsolete)	DSP System Toolbox
RLS Adaptive Filter (Obsolete)	DSP System Toolbox

See “Adaptive Filters in Simulink” for related information.

Introduced in R2008b

Kalman Filter

Predict or estimate states of dynamic systems



Library

Filtering/Adaptive Filters

`dspadpt3`

Description

Use the Kalman Filter block to predict or estimate the state of a dynamic system from a series of incomplete and/or noisy measurements. Suppose you have a noisy linear system that is defined by the following equations:

$$x_k = Ax_{k-1} + w_{k-1}$$
$$z_k = Hx_k + v_k$$

This block can use the previously estimated state, $\hat{x}_{k-1}$, to predict the current state at time k, x_k^-, as shown by the following equation:

$$x_k^- = A\hat{x}_{k-1}$$
$$P_k^- = A\hat{P}_{k-1}A^T + Q$$

The block can also use the current measurement, z_k, and the predicted state, x_k^-, to estimate the current state value at time k, $\hat{x}_k$, so that it is a more accurate approximation:

$$K_k = P_k^- H^T \left(H P_k^- H^T + R \right)^{-1}$$

$$\hat{x}_k = x_k^- + K_k \left(z_k - H x_k^- \right)$$

$$\hat{P}_k = \left(I - K_k H \right) P_k^-$$

The variables in the previous equations are defined in the following table.

Variable	Definition	Default Value or Initial Condition
x	State	N/A
$\hat{x}$	Estimated state	`zeros([6, 1])`
x^-	Predicted state	N/A
A	State transition matrix	$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$
w	Process noise	N/A
z	Measurement	N/A
H	Measurement matrix	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$
v	Measurement noise	N/A
$\hat{P}$	Estimated error covariance	`10*eye(6)`
P^-	Predicted error covariance	N/A
Q	Process noise covariance	`0.05*eye(6)`
K	Kalman gain	N/A

Variable	Definition	Default Value or Initial Condition
R	Measurement noise covariance	`eye(4)`
I	Identity matrix	N/A

In the previous equations, z is a vector of measurement values. Most of the time, the block processes Z, an M-by-N matrix, where M is the number of measurement values and N is the number of filters.

Use the **Number of filters** parameter to specify the number of filters to use to predict or estimate the current value.

Use the **Enable filters** parameter to specify which filters are enabled or disabled at each time step. If you select `Always`, the filters are always enabled. If you choose `Specify via input port <Enable>`, the Enable port appears on the block. The input to this port must be a row vector of 1s and 0s whose length is equal to the number of filters. For example, if there are 3 filters and the input to the Enable port is [1 0 1], only the first and third filter are enabled at this time step. If you select the **Reset the estimated state and estimated error covariance when filters are disabled** check box, the estimated and predicted states as well as the estimated error covariance that correspond to the disabled filters are reset to their initial values.

Note: All filters have the same state transition matrix, measurement matrix, initial conditions, and noise covariance, but their state, measurement, enable, and MSE signals are unique. Within the state, measurement, enable, and MSE signals, each column corresponds to a filter.

Use the **Measurement matrix source** parameter to specify how to enter the measurement matrix values. If you select `Specify via dialog`, the **Measurement matrix** parameter appears in the dialog box. If you select `Input port <H>`, the H port appears on the block. Use this port to specify your measurement matrix.

See the Radar Tracking example for a demonstration of how to use this block. You can open this example by typing

```
aero_radmod_dsp
```

at the MATLAB command prompt.

Parameters

Number of filters

Specify the number of filters to use to predict or estimate the current value.

Enable filters

Specify which filters are enabled or disabled at each time step. If you select `Always`, the filters are always enabled. If you choose `Specify via input port <Enable>`, the Enable port appears on the block.

Reset the estimated state and estimated error covariance when filters are disabled

If you select this check box, the estimated and predicted states as well as the estimated error covariance that correspond to the disabled filters are reset to their initial values. This parameter is visible if, for the **Enable filters** parameter, you select `Specify via input port <Enable>`.

Initial condition for estimated state

Enter the initial condition for the estimated state.

Initial condition for estimated error covariance

Enter the initial condition for the estimated error covariance.

State transition matrix

Enter the state transition matrix.

Process noise covariance

Enter the process noise covariance.

Measurement matrix source

Specify how to enter the measurement matrix values. If you select `Specify via dialog`, the **Measurement matrix** parameter appears in the dialog box. If you select `Input port <H>`, the H port appears on the block.

Measurement matrix

Enter the measurement matrix values. This parameter is visible if you select `Specify via dialog` for the **Measurement matrix source** parameter.

Measurement noise covariance

Enter the measurement noise covariance.

Output estimated measurement <Z_est>

Select this check box if you want the block to output the estimated measurement.

Output estimated state <X_est>

Select this check box if you want the block to output the estimated state.

Output MSE of estimated state <MSE_est>

Select this check box if you want the block to output the mean-squared error of the estimated state.

Output predicted measurement <Z_prd>

Select this check box if you want the block to output the predicted measurement.

Output predicted state <X_prd>

Select this check box if you want the block to output the predicted state.

Output MSE of predicted state <MSE_prb>

Select this check box if you want the block to output the mean-squared error of the predicted state.

References

[1] Haykin, Simon. *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice Hall, 1996.

[2] Welch, Greg and Gary Bishop, “An Introduction to the Kalman Filter,” TR 95–041, Department of Computer Science, University of North Carolina.

Supported Data Types

Port	Input/Output	Supported Data Types
Z	M-by-N measurement where M is the length of the measurement vector and N is the number of filters.	• Double-precision floating point • Single-precision floating point
Enable	1-by-N vector of 1s and 0s where N is the number of filters.	• Double-precision floating point • Single-precision floating point • Boolean
H	M-by-P measurement matrix where M is the length of the measurement vector and P is the length of the filter state vectors.	Same as Z port

Port	Input/Output	Supported Data Types
Z_est	M-by-N estimated measurement matrix where M is the length of the measurement vector and N is the number of filters.	Same as Z port
X_est	P-by-N estimated state matrix where P is the length of the filter state vectors and N is the number of filters.	Same as Z port
MSE_est	1-by-N vector that represents the mean-squared-error of the estimated state. N is the number of filters.	Same as Z port
Z_prd	M-by-N predicted measurement matrix where M is the length of the measurement vector and N is the number of filters.	Same as Z port
X_prd	P-by-N predicted state matrix where P is the length of the filter state vectors and N is the number of filters.	Same as Z port
MSE_prd	1-by-N vector that represents the mean-squared-error of the predicted state. N is the number of filters.	Same as Z port

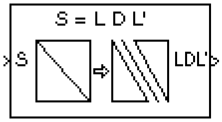
See Also

`LDL Solver`	DSP System Toolbox

Introduced in R2007a

LDL Factorization

Factor square Hermitian positive definite matrices into lower, upper, and diagonal components



Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

`dspfactors`

Description

The LDL Factorization block uniquely factors the square Hermitian positive definite input matrix S as

$$S = LDL^*$$

where L is a lower triangular square matrix with unity diagonal elements, D is a diagonal matrix, and L^* is the Hermitian (complex conjugate) transpose of L. Only the diagonal and lower triangle of the input matrix are used, and any imaginary component of the diagonal entries is disregarded.

The block's output is a composite matrix with lower triangle elements l_{ij} from L, diagonal elements d_{ij} from D, and upper triangle elements u_{ij} from L^*. The output format is shown below for a 5-by-5 matrix.

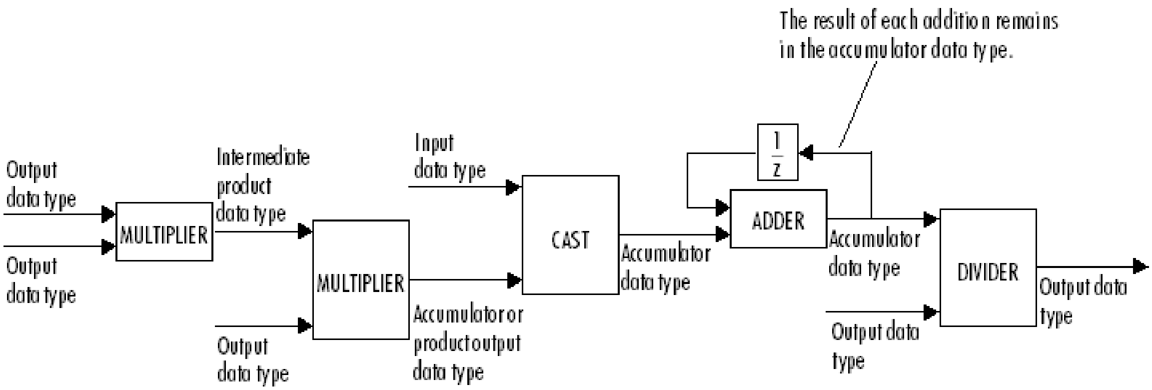
$$\begin{bmatrix} d_{11} & u_{12} & u_{13} & u_{14} & u_{15} \\ l_{21} & d_{22} & u_{23} & u_{24} & u_{25} \\ l_{31} & l_{32} & d_{33} & u_{34} & u_{35} \\ l_{41} & l_{42} & l_{43} & d_{44} & u_{45} \\ l_{51} & l_{52} & l_{53} & l_{54} & d_{55} \end{bmatrix} \qquad u_{ij} = l_{ji}^*$$

LDL factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable. It is more efficient than Cholesky factorization because it avoids computing the square roots of the diagonal elements.

The algorithm requires that the input be square and Hermitian positive definite. When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter.

Fixed-Point Data Types

The following diagram shows the data types used within the LDL Factorization block for fixed-point signals.

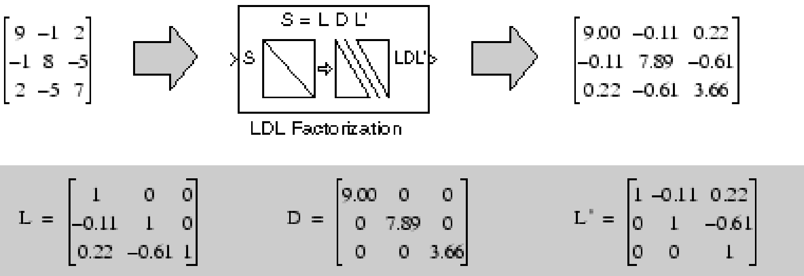


You can set the intermediate product, product output, accumulator, and output data types in the block dialog as discussed below.

The output of the second multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

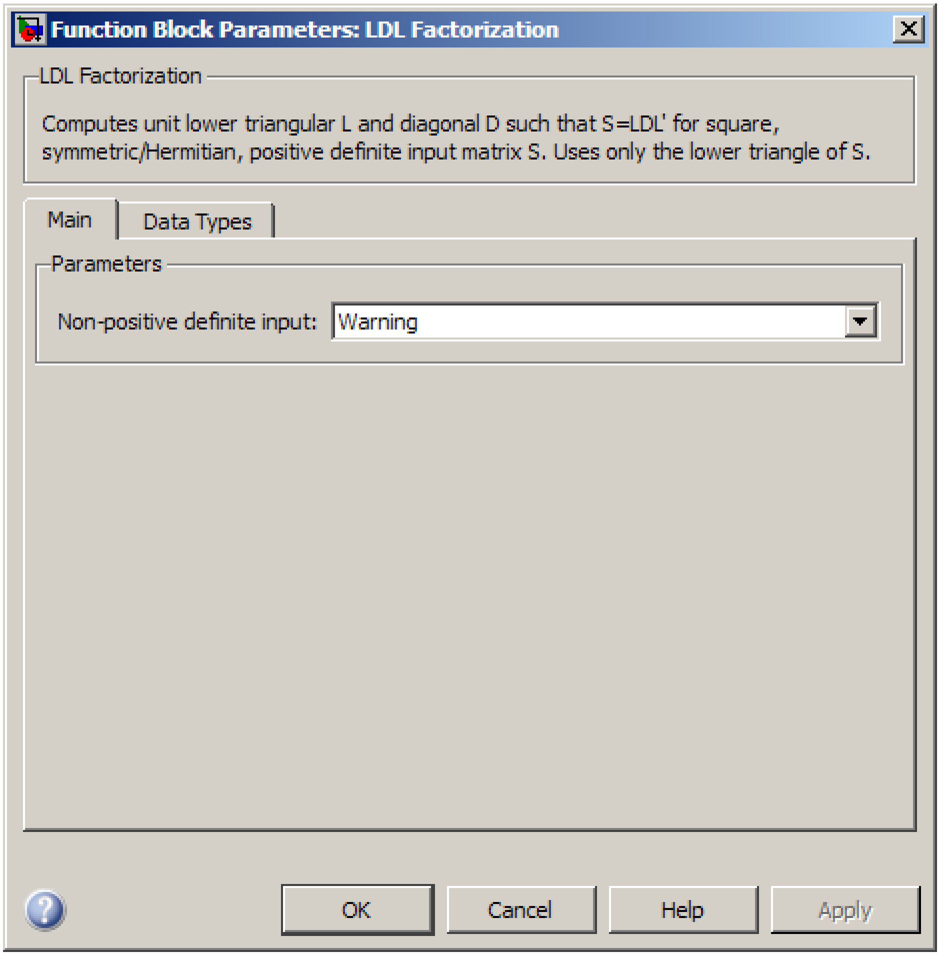
Examples

LDL decomposition of a 3-by-3 Hermitian positive definite matrix:



Dialog Box

The **Main** pane of the LDL Factorization block dialog appears as follows.



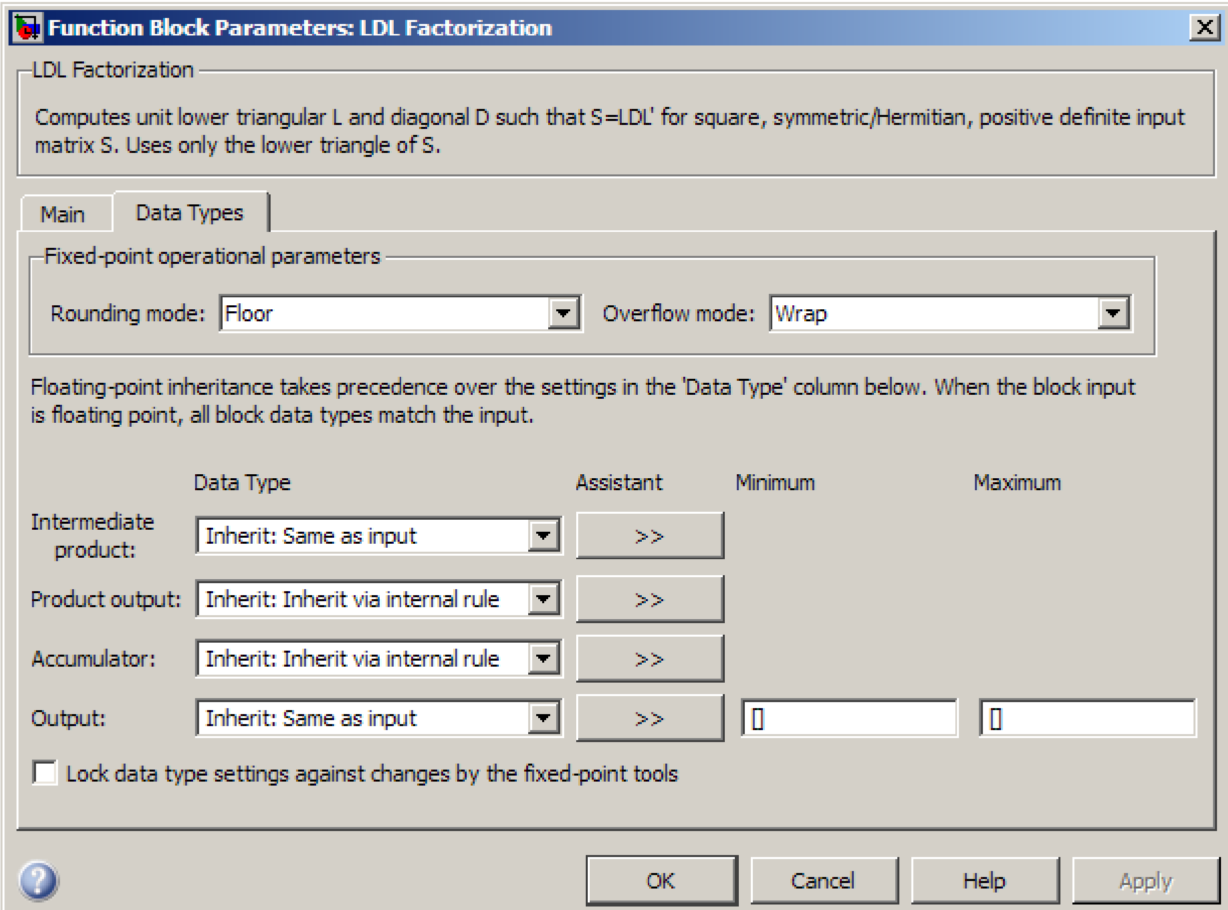
Non-positive definite input

Specify the action when nonpositive definite matrix inputs occur:

- `Ignore` — Proceed with the computation and do not issue an alert. The output is not a valid factorization. A partial factorization is present in the upper left corner of the output.
- `Warning` — Display a warning message in the MATLAB Command Window, and continue the simulation. The output is not a valid factorization. A partial factorization is present in the upper left corner of the output.

- `Error` — Display an error dialog and terminate the simulation.

The **Data Types** pane of the LDL Factorization block dialog appears as follows.



Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Intermediate product

Specify the intermediate product data type. As shown in “Fixed-Point Data Types” on page 2-1026, the output of the multiplier is cast to the intermediate product data type before the next element of the input is multiplied into it. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-1026 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-1026 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-1026 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

Port	Supported Data Types
S	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
LDL'	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

See Also

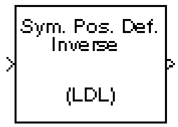
`Cholesky Factorization`	DSP System Toolbox
`LDL Inverse`	DSP System Toolbox
`LDL Solver`	DSP System Toolbox
`LU Factorization`	DSP System Toolbox
`QR Factorization`	DSP System Toolbox

See “Matrix Factorizations” for related information.

Introduced before R2006a

LDL Inverse

Compute inverse of Hermitian positive definite matrix using LDL factorization



Library

Math Functions / Matrices and Linear Algebra / Matrix Inverses

`dspinverses`

Description

The LDL Inverse block computes the inverse of the Hermitian positive definite input matrix S by performing an LDL factorization.

$$S^{-1} = \left(LDL^{*}\right)^{-1}$$

L is a lower triangular square matrix with unity diagonal elements, D is a diagonal matrix, and L^{*} is the Hermitian (complex conjugate) transpose of L. Only the diagonal and lower triangle of the input matrix are used, and any imaginary component of the diagonal entries is disregarded.

LDL factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable. It is more efficient than Cholesky factorization because it avoids computing the square roots of the diagonal elements.

The algorithm requires that the input be Hermitian positive definite. When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- `Ignore` — Proceed with the computation and do not issue an alert. The output is not a valid inverse.

- `Warning` — Display a warning message in the MATLAB command window, and continue the simulation. The output is not a valid inverse.
- `Error` — Display an error dialog and terminate the simulation.

Note The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog, it is set to `Ignore` in the code generated for this block by Simulink Coder code generation software.

Parameters

Non-positive definite input

Response to nonpositive definite matrix inputs.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

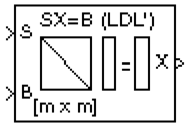
`Cholesky Inverse`	DSP System Toolbox
`LDL Factorization`	DSP System Toolbox
`LDL Solver`	DSP System Toolbox
`LU Inverse`	DSP System Toolbox
`Pseudoinverse`	DSP System Toolbox
`inv`	MATLAB

See “Matrix Inverses” for related information.

Introduced before R2006a

LDL Solver

Solve *SX*=*B* for *X* when *S* is square Hermitian positive definite matrix



Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

`dspsolvers`

Description

The LDL Solver block solves the linear system *S*X=*B* by applying LDL factorization to the matrix at the S port, which must be square (*M*-by-*M*) and Hermitian positive definite. Only the diagonal and lower triangle of the matrix are used, and any imaginary component of the diagonal entries is disregarded. The input to the B port is the right side *M*-by-*N* matrix, *B*. The *M*-by-*N* output matrix *X* is the unique solution of the equations.

A length-*M* unoriented vector input for right side *B* is treated as an *M*-by-1 matrix.

When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- `Ignore` — Proceed with the computation and do not issue an alert. The output is not a valid solution.
- `Warning` — Proceed with the computation and display a warning message in the MATLAB Command Window. The output is not a valid solution.
- `Error` — Display an error dialog and terminate the simulation.

Note The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog, it is set to `Ignore` in the code generated for this block by Simulink Coder code generation software.

Algorithm

The LDL algorithm uniquely factors the Hermitian positive definite input matrix S as
$S = LDL^*$

where L is a lower triangular square matrix with unity diagonal elements, D is a diagonal matrix, and L^* is the Hermitian (complex conjugate) transpose of L.

The equation
$LDL^*X = B$

is solved for X by the following steps:

1 Substitute
  $Y = DL^*X$
2 Substitute
  $Z = L^*X$
3 Solve one diagonal and two triangular systems.
  $LY = B$
  $DZ = Y$
  $L^*X = Z$

Parameters

Non-positive definite input

Response to nonpositive definite matrix inputs.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

`Autocorrelation LPC`	DSP System Toolbox

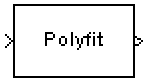
`Cholesky Solver`	DSP System Toolbox
`LDL Factorization`	DSP System Toolbox
`LDL Inverse`	DSP System Toolbox
`Levinson-Durbin`	DSP System Toolbox
`LU Solver`	DSP System Toolbox
`QR Solver`	DSP System Toolbox

See “Linear System Solvers” for related information.

Introduced before R2006a

Least Squares Polynomial Fit

Compute polynomial coefficients that best fit input data in least-squares sense



Library

Math Functions / Polynomial Functions

`dsppolyfun`

Description

The Least Squares Polynomial Fit block computes the coefficients of the nth order polynomial that best fits the input data in the least-squares sense, where you specify n in the **Polynomial order** parameter. A distinct set of n+1 coefficients is computed for each column of the M-by-N input, u.

For a given input column, the block computes the set of coefficients, c_1, c_2, ..., c_{n+1}, that minimizes the quantity

$$\sum_{i=1}^{M} (u_i - \hat{u}_i)^2$$

where u_i is the ith element in the input column, and

$$\hat{u}_i = f(x_i) = c_1 x_i^n + c_2 x_i^{n-1} + \ \ldots \ + c_{n+1}$$

The values of the independent variable, x_1, x_2, ..., x_M, are specified as a length-M vector by the **Control points** parameter. The same M control points are used for all N polynomial fits, and can be equally or unequally spaced. The equivalent MATLAB code is shown below.

```
c = polyfit(x,u,n)      % Equivalent MATLAB code
```

For convenience, the block treats length-M unoriented vector input as an M-by-1 matrix.

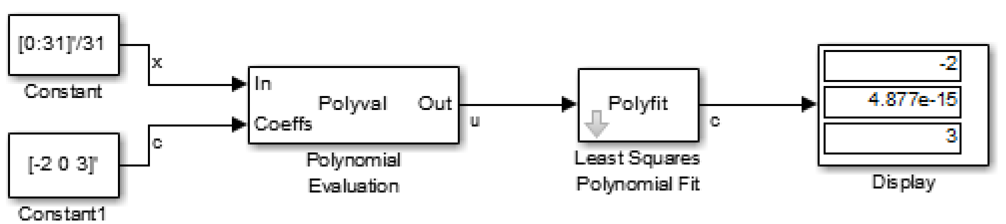
Each column of the (n+1)-by-N output matrix, c, represents a set of n+1 coefficients describing the best-fit polynomial for the corresponding column of the input. The coefficients in each column are arranged in order of descending exponents, c_1, c_2, ..., c_{n+1}.

Examples

In the ex_leastsquarespolyfit_ref model below, the Polynomial Evaluation block uses the second-order polynomial

$$y = -2u^2 + 3$$

to generate four values of dependent variable y from four values of independent variable u, received at the top port. The polynomial coefficients are supplied in the vector `[-2 0 3]` at the bottom port. Note that the coefficient of the first-order term is zero.



The **Control points** parameter of the Least Squares Polynomial Fit block is configured with the same four values of independent variable u that are used as input to the Polynomial Evaluation block, `[1 2 3 4]`. The Least Squares Polynomial Fit block uses these values together with the input values of dependent variable y to reconstruct the original polynomial coefficients.

Parameters

Control points

The values of the independent variable to which the data in each input column correspond. For an M-by-N input, this parameter must be a length-M vector. Tunable (Simulink).

Polynomial order

The order, n, of the polynomial to be used in constructing the best fit. The number of coefficients is n+1.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

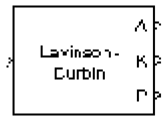
See Also

Detrend	DSP System Toolbox
Polynomial Evaluation	DSP System Toolbox
Polynomial Stability Test	DSP System Toolbox
`polyfit`	MATLAB

Introduced before R2006a

Levinson-Durbin

Solve linear system of equations using Levinson-Durbin recursion



Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

`dspsolvers`

Description

The Levinson-Durbin block solves the nth-order system of linear equations
$Ra = b$

in the cases where:

- R is a Hermitian, positive-definite, Toeplitz matrix.
- b is identical to the first column of R shifted by one element and with the opposite sign.

$$\begin{bmatrix} r(1) & r^*(2) & \cdots & r^*(n) \\ r(2) & r(1) & \cdots & r^*(n-1) \\ \vdots & \vdots & \ddots & \vdots \\ r(n) & r(n-1) & \cdots & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(n+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(n+1) \end{bmatrix}$$

The input to the block, `r = [r(1) r(2) ... r(n+1)]`, can be a vector or a matrix. If the input is a matrix, the block treats each column as an independent channel and solves it separately. Each channel of the input contains lags *0* through *n* of an autocorrelation sequence, which appear in the matrix *R*.

The block can output the polynomial coefficients, *A*, the reflection coefficients, *K*, and the prediction error power, *P*, in various combinations. The **Output(s)** parameter allows you to enable the *A* and *K* outputs by selecting one of the following settings:

- `A` — For each channel, port A outputs `A = [1 a(2) a(3) ... a(n+1)]`, the solution to the Levinson-Durbin equation. *A* has the same dimension as the input. You can also view the elements of each output channel as the coefficients of an *n*th-order autoregressive (AR) process.
- `K` — For each channel, port K outputs `K = [k(1) k(2) ... k(n)]`, which contains *n* reflection coefficients and has the same dimension as the input, less one element. A scalar input channel causes an error when you select `K`. You can use reflection coefficients to realize a lattice representation of the AR process described later in this page.
- `A and K` — The block outputs both representations at their respective ports. A scalar input channel causes an error when you select `A and K`.

Select the **Output prediction error power (P)** check box to output the prediction error power for each channel, *P*. For each channel, *P* represents the power of the output of an FIR filter with taps *A* and input autocorrelation described by *r*, where *A* represents a prediction error filter and *r* is the input to the block. In this case, *A* is a whitening filter. *P* has one element per input channel.

When you select the **If the value of lag 0 is zero, A=[1 zeros], K=[zeros], P=0** check box (default), an input channel whose `r(1)` element is zero generates a zero-valued output. When you clear this check box, an input with `r(1) = 0` generates `NaN`s in the output. In general, an input with `r(1) = 0` is invalid because it does not construct a positive-definite matrix *R*. Often, however, blocks receive zero-valued inputs at the start of a simulation. The check box allows you to avoid propagating `NaN`s during this period.

Applications

One application of the Levinson-Durbin formulation implemented by this block is in the Yule-Walker AR problem, which concerns modeling an unknown system as an autoregressive process. You would model such a process as the output of an all-pole IIR filter with white Gaussian noise input. In the Yule-Walker problem, the use of the signal's autocorrelation sequence to obtain an optimal estimate leads to an $Ra = b$ equation of the type shown above, which is most efficiently solved by Levinson-Durbin recursion. In this case, the input to the block represents the autocorrelation sequence, with `r(1)` being the zero-lag value. The output at the block's A port then contains the coefficients of the autoregressive process that optimally models the system. The coefficients are ordered in descending powers of z, and the AR process is minimum phase. The prediction error, G, defines the gain for the unknown system, where $G = \sqrt{P}$:

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ \ldots \ + a(n+1)z^{-n}}$$

The output at the block's K port contains the corresponding reflection coefficients, [k(1) k(2) ... k(n)], for the lattice realization of this IIR filter. The Yule-Walker AR Estimator block implements this autocorrelation-based method for AR model estimation, while the Yule-Walker Method block extends the method to spectral estimation.

Another common application of the Levinson-Durbin algorithm is in linear predictive coding, which is concerned with finding the coefficients of a moving average (MA) process (or FIR filter) that predicts the next value of a signal from the current signal sample and a finite number of past samples. In this case, the input to the block represents the signal's autocorrelation sequence, with r(1) being the zero-lag value, and the output at the block's A port contains the coefficients of the predictive MA process (in descending powers of z).

$$H(z) = A(z) = 1 + a(2)z^{-1} + \ \ldots \ a(n+1)z^{-n}$$

These coefficients solve the following optimization problem:

$$\min_{\{a_i\}}$$

$$E\left[\left|x_n - \sum_{i=1}^{N} a_i x_{n-i}\right|^2\right]$$

Again, the output at the block's K port contains the corresponding reflection coefficients, `[k(1) k(2) ... k(n)]`, for the lattice realization of this FIR filter.

The Autocorrelation LPC block in the Linear Prediction library implements this autocorrelation-based prediction method.

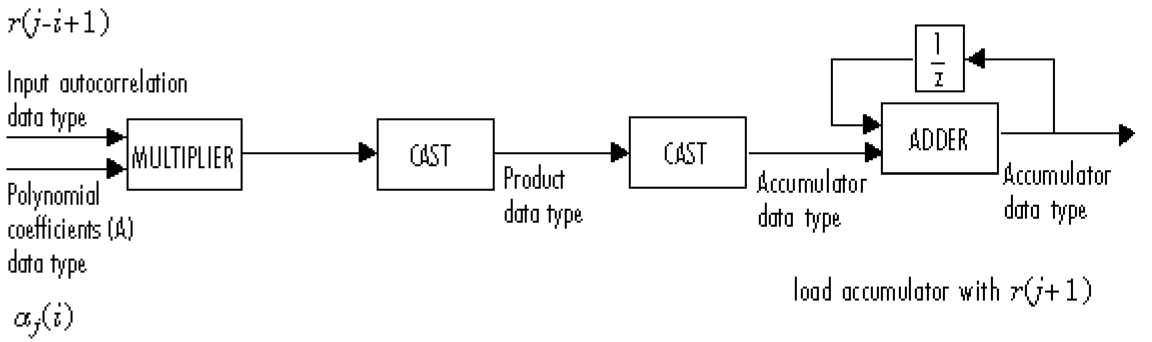
Fixed-Point Data Types

The diagrams in this section show the data types used within the Levinson-Durbin block for fixed-point signals.

After initialization the block performs n updates. At the (j+1) update,

$$\text{value in accumulator } = r(j+1) + \sum a_j(i) \times r(j-i+1)$$

The following diagram displays the fixed-point data types used in this calculation:



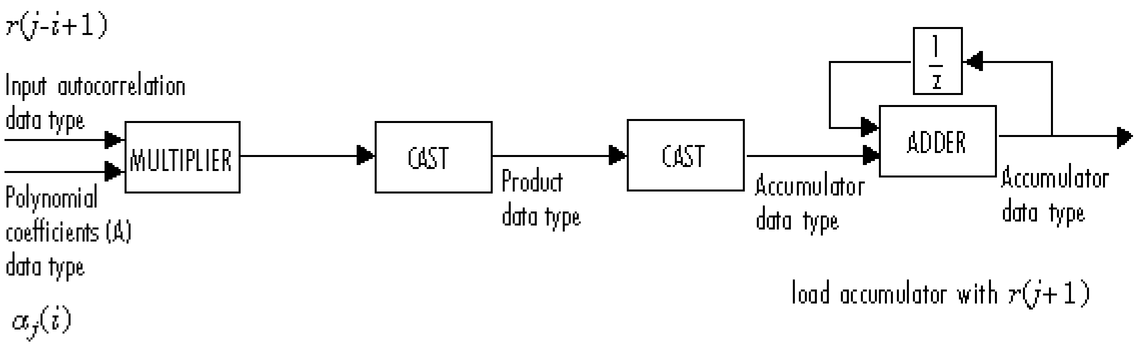
The block then updates the reflection coefficients K according to

$$K_{j+1} = \text{ value in accumulator } / P_j$$

The block then updates the prediction error power P according to

$$P_{j+1} = P_j - P_j \times K_{j+1} \times \text{conj}(K_{j+1})$$

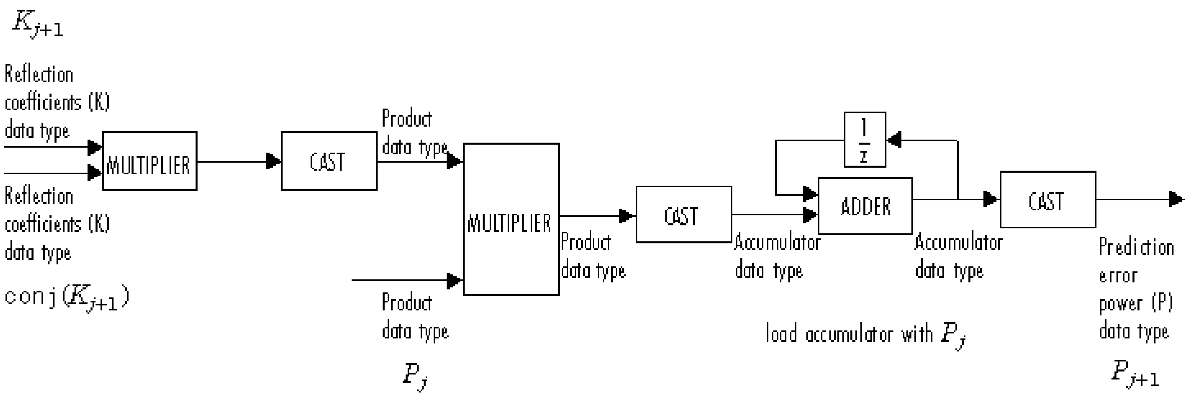
The next diagram displays the fixed-point data types used in this calculation:



The polynomial coefficients A are then updated according to

$$a_{j+1}(i) = a_j(i) + K_{j+1} \times \mathrm{conj}\left(\mathrm{a}_\mathrm{j}(j-1+i)\right)$$

This diagram displays the fixed-point data types used in this calculation:

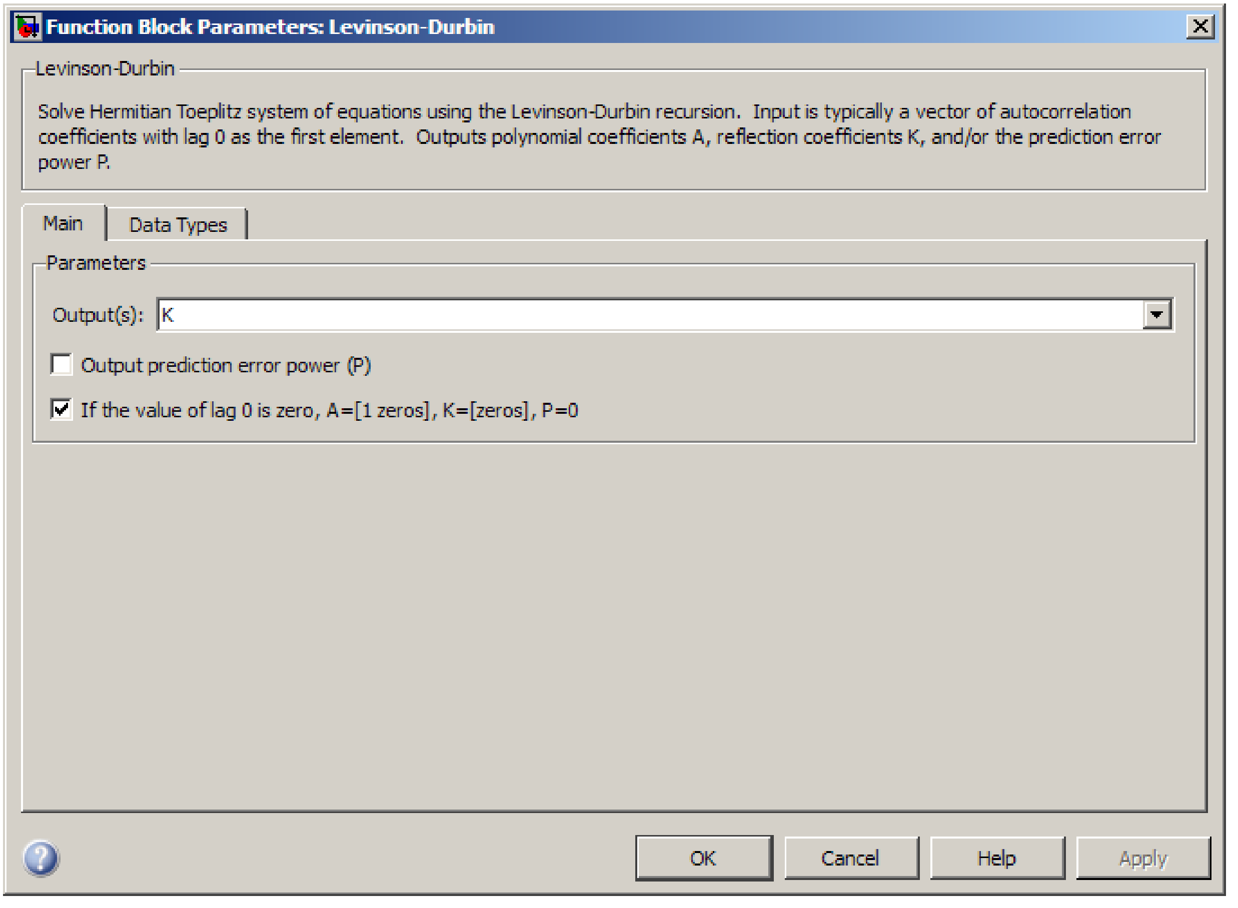


Algorithm

The algorithm requires $O(n^2)$ operations for each input channel. This implementation is therefore much more efficient for large n than standard Gaussian elimination, which requires $O(n^3)$ operations per channel.

Dialog Box

The **Main** pane of the Levinson-Durbin block dialog box appears as follows.



Output(s)

Specify the solution representation of $Ra = b$ to output: model coefficients (A), reflection coefficients (K), or both (A and K). When the input is a scalar or row vector, you must set this parameter to A.

Output prediction error power (P)

Select to output the prediction error at port P.

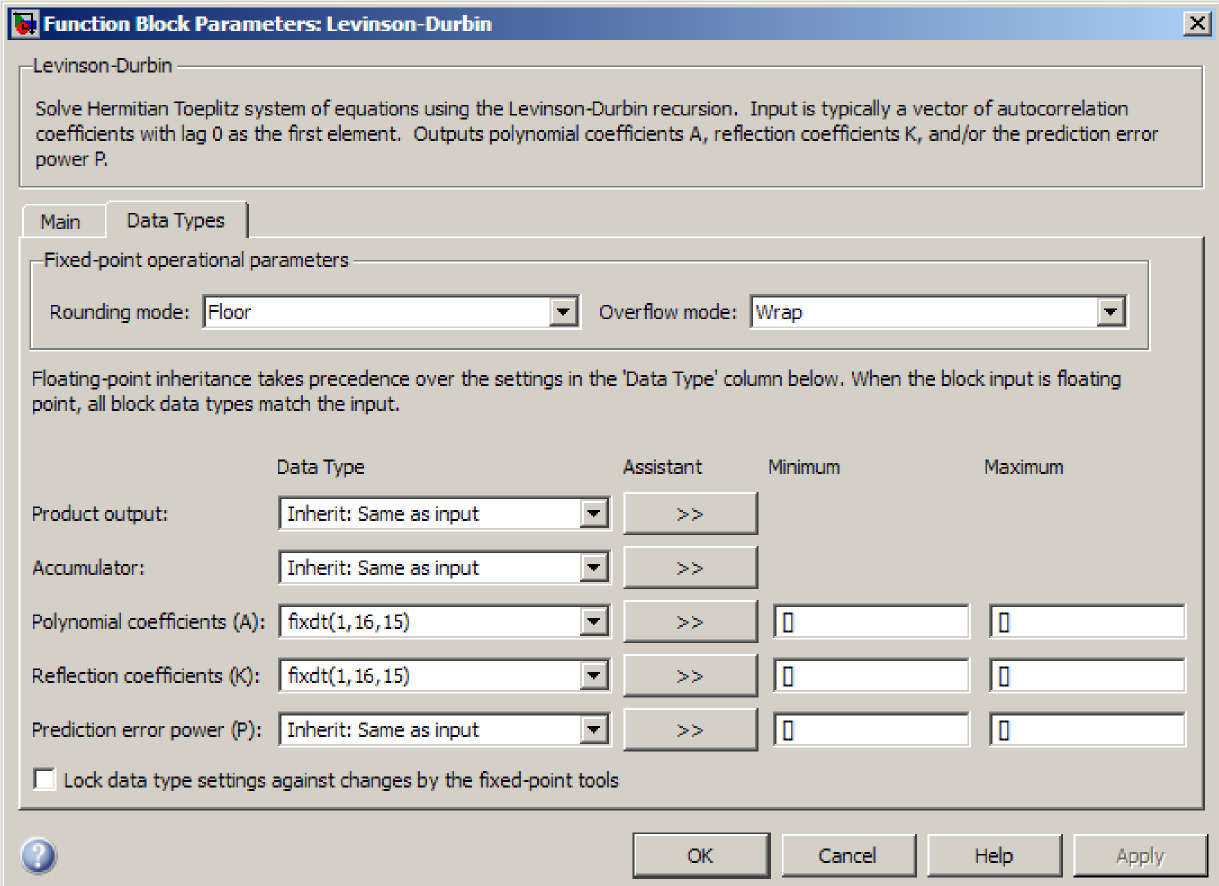
If the value of lag 0 is zero, A=[1 zeros], K=[zeros], P=0

When you select this check box and the first element of the input, `r(1)`, is zero, the block outputs the following vectors, as appropriate:

- `A = [1 zeros(1,n)]`
- `K = [zeros(1,n)]`
- `P = 0`

When you clear this check box, the block outputs a vector of `NaN`s for each channel whose `r(1)` element is zero.

The **Data Types** pane of the Levinson-Durbin block dialog box appears as follows.



Note: Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-1045 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-1045 for illustrations depicting the use of the accumulator data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Polynomial coefficients (A)

Specify the polynomial coefficients (*A*) data type. See “Fixed-Point Data Types” on page 2-1045 for illustrations depicting the use of the A data type in this block. You can set it to an expression that evaluates to a valid data type, for example, `fixdt(1,16,15)`.

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **A** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Reflection coefficients (K)

Specify the polynomial coefficients (*A*) data type. See “Fixed-Point Data Types” on page 2-1045 for illustrations depicting the use of the K data type in this block. You can set it to an expression that evaluates to a valid data type, for example, `fixdt(1,16,15)`.

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **K** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Prediction error power (P)

Specify the prediction error power (*P*) data type. See “Fixed-Point Data Types” on page 2-1045 for illustrations depicting the use of the P data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **P** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Minimum

Specify the minimum values that the polynomial coefficients, reflection coefficients, or prediction error power should have. The default value is `[]` (unspecified).Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters” (Simulink))
- Automatic scaling of fixed-point data types

Maximum

Specify the maximum values that the polynomial coefficients, reflection coefficients, or prediction error power should have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see "Specify Minimum and Maximum Values for Block Parameters" (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

References

Golub, G. H. and C. F. Van Loan. Sect. 4.7 in *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Ljung, L. *System Identification: Theory for the User*. Englewood Cliffs, NJ: Prentice Hall, 1987. Pgs. 278–280.

Kay, Steven M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1988.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

See Also

`Cholesky Solver`	DSP System Toolbox
`LDL Solver`	DSP System Toolbox
`Autocorrelation LPC`	DSP System Toolbox
`LU Solver`	DSP System Toolbox
`QR Solver`	DSP System Toolbox

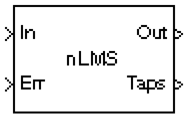
`Yule-Walker AR Estimator`	DSP System Toolbox
`Yule-Walker Method`	DSP System Toolbox
`levinson`	Signal Processing Toolbox

See "Linear System Solvers" for related information.

Introduced before R2006a

LMS Adaptive Filter (Obsolete)

Compute filter estimates for input using LMS adaptive filter algorithm



Library

`dspobslib`

Description

Note The LMS Adaptive Filter block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the `LMS Filter` block.

The LMS Adaptive Filter block implements an adaptive FIR filter using the stochastic gradient algorithm known as the normalized least mean-square (LMS) algorithm.

$$y(n) = \hat{w}^H(n-1)u(n)$$

$$e(n) = d(n) - y(n)$$

$$\hat{w}(n) = \hat{w}(n-1) + \frac{u(n)}{a + u^H(n)u(n)}\mu e^*(n)$$

The variables are as follows.

Variable	Description
n	The current algorithm iteration
$u(n)$	The buffered input samples at step n
$\hat{w}(n)$	The vector of filter-tap estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
μ	The adaptation step size

To overcome potential numerical instability in the tap-weight update, a small positive constant (a = 1e-10) has been added in the denominator.

To turn off normalization, clear the **Use normalization** check box in the parameter dialog. The block then computes the filter-tap estimate as

$$\hat{w}(n) = \hat{w}(n-1) + u(n)\mu e^*(n)$$

The block icon has port labels corresponding to the inputs and outputs of the LMS algorithm. Note that inputs to the In and Err ports must be sample-based scalars. The signal at the Out port is a scalar, while the signal at the Taps port is a sample-based vector.

Block Ports	Corresponding Variables
`In`	u, the scalar input, which is internally buffered into the vector $u(n)$
`Out`	$y(n)$, the filtered scalar output
`Err`	$e(n)$, the scalar estimation error
`Taps`	$\hat{w}(n)$, the vector of filter-tap estimates

An optional `Adapt` input port is added when you select the **Adapt input** check box in the dialog. When this port is enabled, the block continuously adapts the filter coefficients while the `Adapt` input is nonzero. A zero-valued input to the Adapt port causes the block to stop adapting, and to hold the filter coefficients at their current values until the next nonzero `Adapt` input.

The **FIR filter length** parameter specifies the length of the filter that the LMS algorithm estimates. The **Step size** parameter corresponds to μ in the equations. Typically, for convergence in the mean square, μ must be greater than 0 and less than 2. The **Initial value of filter taps** specifies the initial value $\hat{w}(0)$ as a vector, or as a scalar to be repeated for all vector elements. The **Leakage factor** specifies the value of the leakage factor, 1 –μ α, in the leaky LMS algorithm below. This parameter must be between 0 and 1.

$$\hat{w}(n+1) = (1-\mu\alpha)\hat{w}(n) + \frac{u(n)}{u^H(n)u(n)}\mu e^*(n)$$

Examples

See the lmsadeq and lmsadtde demos.

Parameters

FIR filter length

The length of the FIR filter.

Step-size

The step-size, usually in the range (0, 2). Tunable (Simulink).

Initial value of filter taps

The initial FIR filter coefficients.

Leakage factor

The leakage factor, in the range [0, 1]. Tunable (Simulink).

Use normalization

Select this check box to compute the filter-tap estimate using the normalized equations.

Adapt input

Enables the Adapt port when selected.

References

Haykin, S. *Adaptive Filter Theory*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1996.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

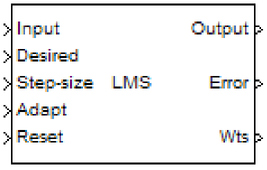
Kalman Adaptive Filter (Obsolete)	DSP System Toolbox
RLS Adaptive Filter (Obsolete)	DSP System Toolbox

See “Adaptive Filters in Simulink” for related information.

Introduced in R2008b

LMS Filter

Compute output, error, and weights using LMS adaptive algorithm



Library

Filtering / Adaptive Filters

`dspadpt3`

Description

The LMS Filter block can implement an adaptive FIR filter using five different algorithms. The block estimates the filter weights, or coefficients, needed to minimize the error, *e(n)*, between the output signal *y(n)* and the desired signal, *d(n)*. Connect the signal you want to filter to the Input port. The input signal can be a scalar or a column vector. Connect the desired signal to the Desired port. The desired signal must have the same data type, complexity, and dimensions as the input signal. The Output port outputs the filtered input signal, which is the estimate of the desired signal. The Error port outputs the result of subtracting the output signal from the desired signal.

When you select `LMS` for the **Algorithm** parameter, the block calculates the filter weights using the least mean-square (LMS) algorithm. This algorithm is defined by the following equations.

$$
\begin{aligned}
y(n) &= \mathbf{w}^T(n-1)\mathbf{u}(n) \\
e(n) &= d(n) - y(n) \\
\mathbf{w}(n) &= \alpha\mathbf{w}(n-1) + f(\mathbf{u}(n), e(n), \mu)
\end{aligned}
$$

The various LMS adaptive filter algorithms available in this block are defined as:

- LMS:

$$f(\mathbf{u}(n),e(n),\mu) = \mu e(n)\mathbf{u}^*(n)$$

- Normalized LMS:

$$f(\mathbf{u}(n),e(n),\mu) = \mu e(n)\frac{\mathbf{u}^*(n)}{\varepsilon + \mathbf{u}^H(n)\mathbf{u}(n)}$$

- Sign-Error LMS:

$$f(\mathbf{u}(n),e(n),\mu) = \mu\, sign(e(n))\mathbf{u}^*(n)$$

- Sign-Data LMS:

$$f(\mathbf{u}(n),e(n),\mu) = \mu e(n)\, sign(\mathbf{u}(n))$$

  where **u**(n) is real.
- Sign-Sign LMS:

$$f(\mathbf{u}(n),e(n),\mu) = \mu\, sign(e(n))\, sign(\mathbf{u}(n))$$

  where **u**(n) is real.

The variables are as follows:

Variable	Description
n	The current time index
$\mathbf{u}(n)$	The vector of buffered input samples at step n
$\mathbf{u}^*(n)$	The complex conjugate of the vector of buffered input samples at step n
$\mathbf{w}(n)$	The vector of filter weight estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n

Variable	Description
μ	The adaptation step size
α	The leakage factor ($0 < \alpha \leq 1$)

In NMLS, to overcome potential numerical instability in the update of the weights, a small positive constant, epsilon, has been added in the denominator. For double-precision floating-point input, epsilon is 2.2204460492503131e-016. For single-precision floating-point input, epsilon is 1.192092896e-07. For fixed-point input, epsilon is 0.

Use the **Filter length** parameter to specify the length of the filter weights vector.

The **Step size (mu)** parameter corresponds to μ in the equations. For convergence of the normalized LMS equations, $0<\mu<2$. You can either specify a step size using the input port, Step-size, or by entering a value in the Block Parameters: LMS Filter dialog.

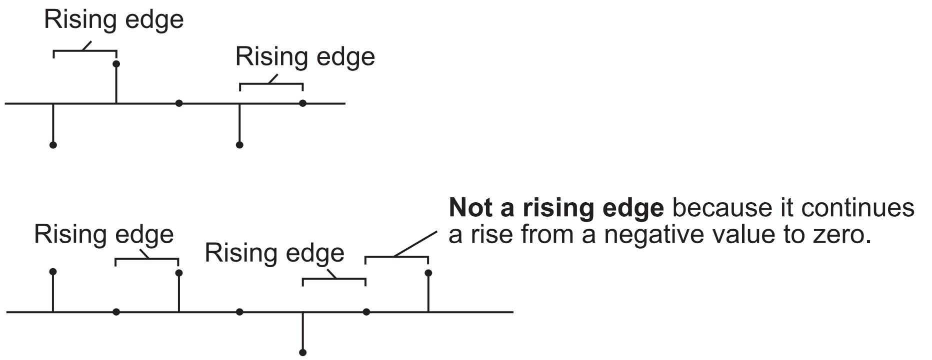
Enter the initial filter weights **w**(0) as a vector or a scalar in the **Initial value of filter weights** text box. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value.

When you select the **Adapt port** check box, an Adapt port appears on the block. When the input to this port is greater than zero, the block continuously updates the filter weights. When the input to this port is less than or equal to zero, the filter weights remain at their current values.

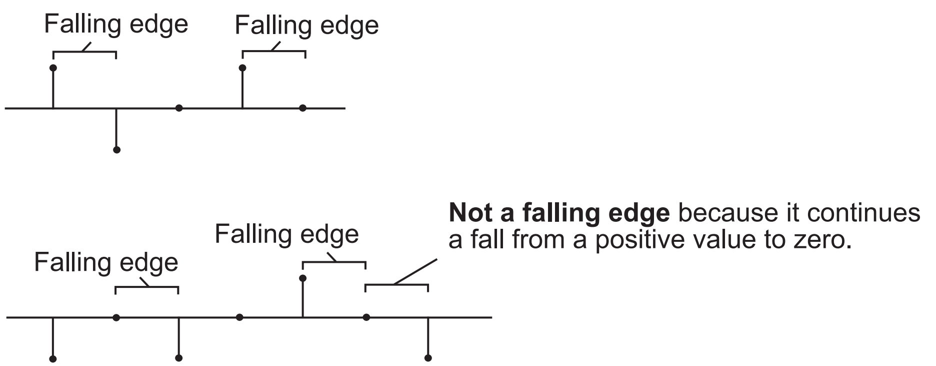
When you want to reset the value of the filter weights to their initial values, use the **Reset port** parameter. The block resets the filter weights whenever a reset event is detected at the Reset port. The reset signal rate must be the same rate as the data signal input.

From the **Reset port** list, select `None` to disable the Reset port. To enable the Reset port, select one of the following from the **Reset port** list:

- `Rising edge` — Triggers a reset operation when the Reset input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- `Falling edge` — Triggers a reset operation when the Reset input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- `Either edge` — Triggers a reset operation when the Reset input is a `Rising edge` or `Falling edge` (as described above)
- `Non-zero sample` — Triggers a reset operation at each sample time that the Reset input is not zero

Select the **Output filter weights** check box to create a Wts port on the block. For each iteration, the block outputs the current updated filter weights from this port.

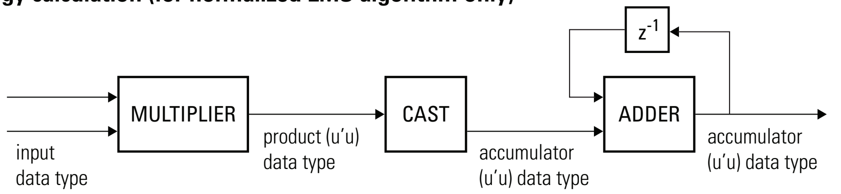
Fixed-Point Data Types

The following diagrams show the data types used within the LMS Filter block for fixed-point signals; the table summarizes the definitions of variables used in the diagrams:

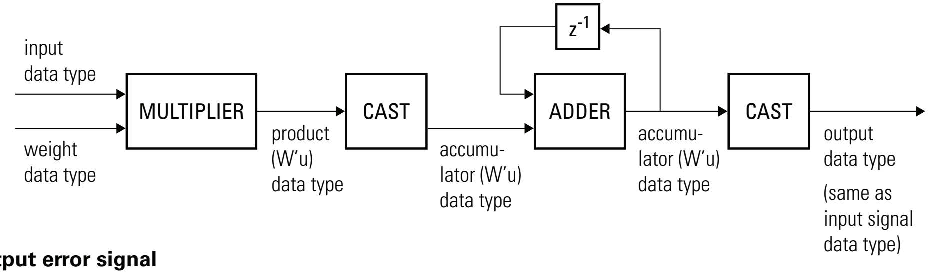
Variable	Definition
u	Input vector
W	Vector of filter weights
μ	Step size
e	Error
Q	Quotient, $Q = \frac{\mu \cdot e}{\mathbf{u}'\mathbf{u}}$
Product **u'u**	Product data type in Energy calculation diagram
Accumulator **u'u**	Accumulator data type in Energy calculation diagram
Product **W'u**	Product data type in Convolution diagram
Accumulator **W'u**	Accumulator data type in Convolution diagram
Product $\mu \cdot e$	Product data type in Product of step size and error diagram
Product $Q \cdot \mathbf{u}$	Product and accumulator data type in Weight update diagram. [1]

[1]The accumulator data type for this quantity is automatically set to be the same as the product data type. The minimum, maximum, and overflow information for this accumulator is logged as part of the product information. Autoscaling treats this product and accumulator as one data type.

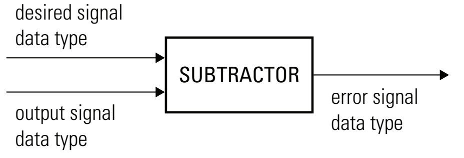
Energy calculation (for normalized LMS algorithm only)



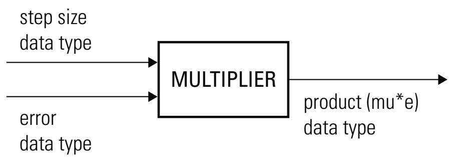
Convolution



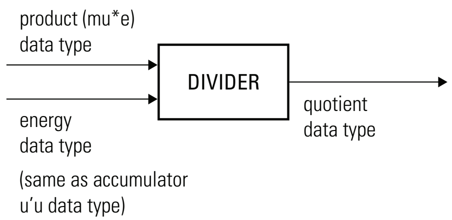
Output error signal



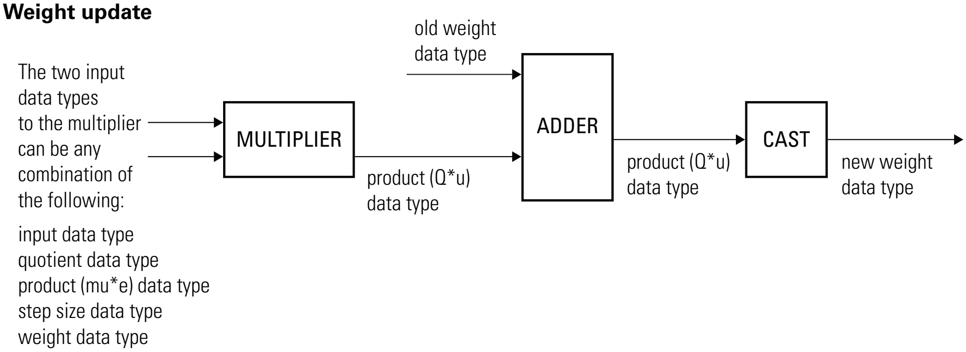
Product of step size and error (for LMS and Sign-Data LMS algorithms only)



Quotient (for normalized LMS only)



Weight update



You can set the data type of the parameters, weights, products, quotient, and accumulators in the block mask. Fixed-point inputs, outputs, and mask parameters of this block must have the following characteristics:

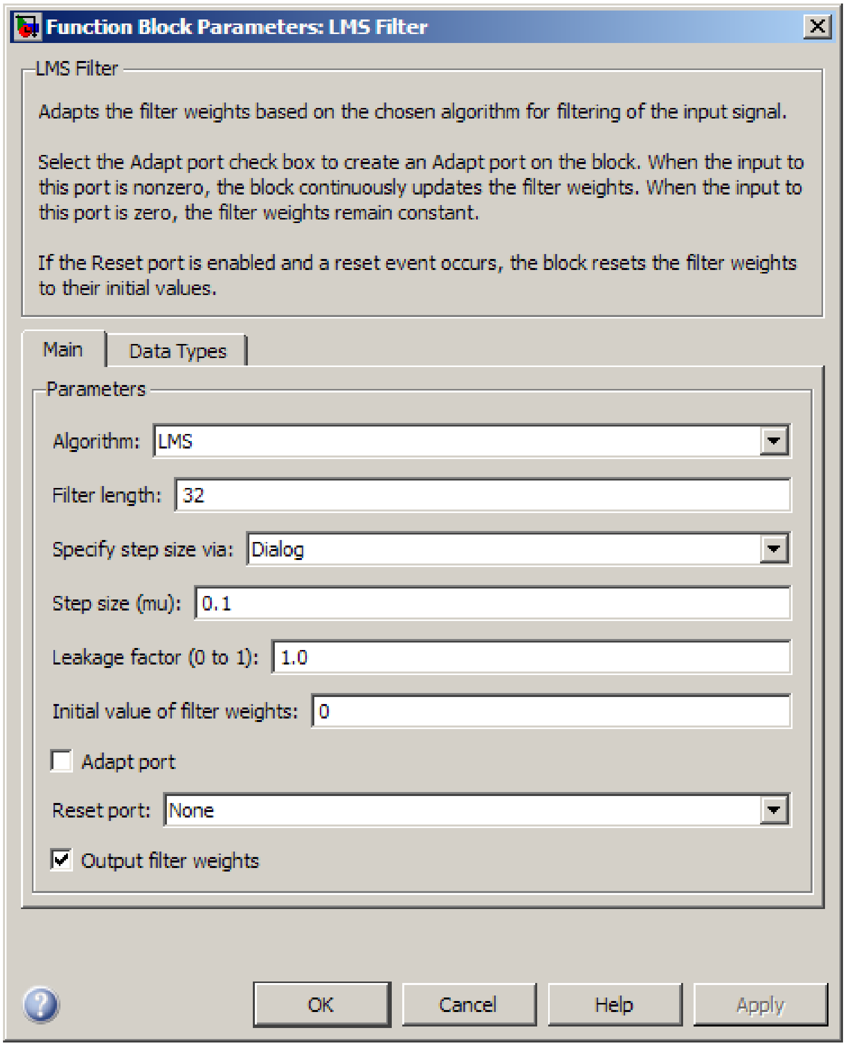
- The input signal and the desired signal must have the same word length, but their fraction lengths can differ.
- The step size and leakage factor must have the same word length, but their fraction lengths can differ.

- The output signal and the error signal have the same word length and the same fraction length as the desired signal.
- The quotient and the product output of the **u'u**, **W'u**, $\mu \cdot e$, and $Q \cdot \mathbf{u}$ operations must have the same word length, but their fraction lengths can differ.
- The accumulator data type of the **u'u** and **W'u** operations must have the same word length, but their fraction lengths can differ.

The output of the multiplier is in the product output data type if at least one of the inputs to the multiplier is real. If both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

Dialog Box

The **Main** pane of the LMS Filter block dialog appears as follows.



Algorithm

Choose the algorithm used to calculate the filter weights.

Filter length

Enter the length of the FIR filter weights vector.

Specify step size via

Select `Dialog` to enter a value for step size in the Block parameters: LMS Filter dialog. Select `Input port` to specify step size using the Step-size input port.

Step size (mu)

Enter the step size μ. Tunable (Simulink).

Leakage factor (0 to 1)

Enter the leakage factor, $0 < 1 - \mu\alpha \leq 1$. Tunable (Simulink).

Initial value of filter weights

Specify the initial values of the FIR filter weights.

Adapt port

Select this check box to enable the Adapt input port.

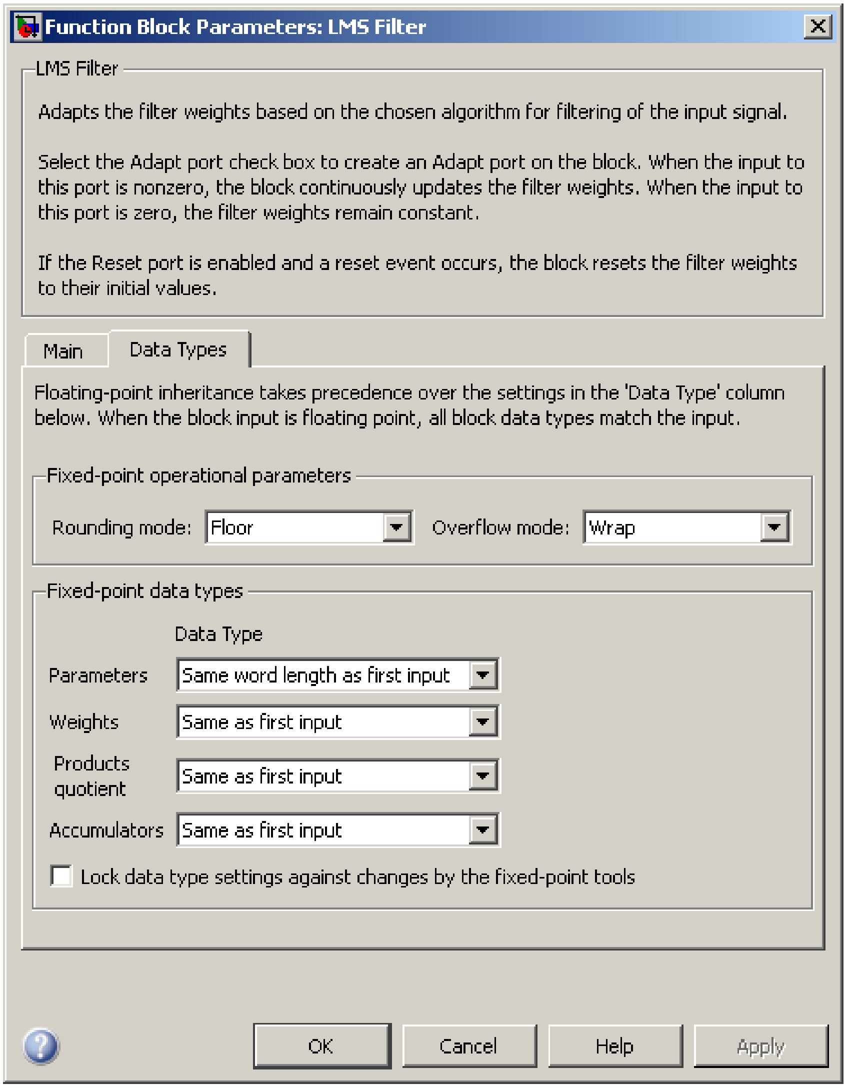
Reset port

Select this check box to enable the Reset input port.

Output filter weights

Select this check box to export the filter weights from the Wts port.

The **Data Types** pane of the LMS Filter block dialog appears as follows.



Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Parameters

This parameter is visible if, for the **Specify step size via** parameter, you choose `Dialog`. Choose how you specify the word length and the fraction length of the leakage factor and step size:

- When you select `Same word length as first input`, the word length of the leakage factor and step size match that of the first input to the block. In this mode, the fraction length of the leakage factor and step size is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Specify word length`, you can enter the word length of the leakage factor and step size, in bits. In this mode, the fraction length of the leakage factor and step size is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the leakage factor and step size, in bits. The leakage factor and the step size must have the same word length, but the fraction lengths can differ.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the leakage factor and step size. The leakage factor and the step size must have the same word length, but the slopes can differ. This block requires a power-of-two slope and a bias of zero.

If, for the **Specify step size via** parameter, you choose `Input port`, the word length of the leakage factor is the same as the word length of the step size input at the Step size port. The fraction length of the leakage factor is automatically set to the best precision possible based on the word length of the leakage factor.

Weights

Choose how you specify the word length and fraction length of the filter weights of the block:

- When you select `Same as first input`, the word length and fraction length of the filter weights match those of the first input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the filter weights, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the filter weights. This block requires a power-of-two slope and a bias of zero.

Products & quotient

Choose how you specify the word length and fraction length of **u'u**, **W'u**, $\mu \cdot e$, $Q \cdot \mathbf{u}$, and the quotient, Q. Here, **u** is the input vector, **W** is the vector of filter weights, μ is the step size, e is the error, and Q is the quotient, which is defined as $Q = \frac{\mu \cdot e}{\mathbf{u}'\mathbf{u}}$

- When you select `Same as first input`, the word length and fraction length of these quantities match those of the first input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of these quantities, in bits. The word length of the quantities must be the same, but the fraction lengths can differ.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of these quantities. The word length of the quantities must be the same, but the slopes can differ. This block requires a power-of-two slope and a bias of zero.

Accumulators

Use this parameter to specify how you would like to designate the word and fraction lengths of the accumulators for the **u'u** and **W'u** operations.

Note This parameter is *not* used to designate the word and fraction lengths of the accumulator for the $Q \cdot \mathbf{u}$ operation. The accumulator data type for this quantity is automatically set to be the same as the product data type. The minimum, maximum, and overflow information for this accumulator is logged as part of the product information. Autoscaling treats this product and accumulator as one data type.

See “Fixed-Point Data Types” on page 2-1062 and “Multiplication Data Types” for illustrations depicting the use of the accumulator data type in this block:

- When you select `Same as first input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the accumulators, in bits. The word length of both the accumulators must be the same, but the fraction lengths can differ.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the accumulators. The word length of both the accumulators

must be the same, but the slopes can differ. This block requires a power-of-two slope and a bias of zero.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see LMS Filter.

References

Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Signed fixed point
Desired	• Must be the same as Input for floating-point signals • Must be any signed fixed-point data type when Input is fixed point
Step-size	• Must be the same as Input for floating-point signals • Must be any signed fixed-point data type when Input is fixed point
Adapt	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
Reset	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers
Output	• Must be the same as Input for floating-point signals • Must be the same as Desired for fixed-point signals
Error	• Must be the same as Input for floating-point signals • Must be the same as Desired for fixed-point signals
Wts	• Must be the same as Input for floating-point signals • Obeys the **Weights** parameter for fixed-point signals

See Also

`LMS Update`	DSP System Toolbox
`RLS Filter`	DSP System Toolbox
`Block LMS Filter`	DSP System Toolbox
`Fast Block LMS Filter`	DSP System Toolbox

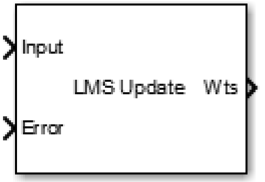
See “Adaptive Filters in Simulink” for related information.

Introduced before R2006a

LMS Update

Estimate weights of LMS adaptive filter

Library: DSP System Toolbox / Filtering / Adaptive Filters



Description

The LMS Update block estimates the weights of an LMS adaptive filter. The block accepts the data and error as inputs and computes the filter weights based on the algorithm the block chooses. For more details on the algorithms, see “Algorithms” on page 2-1076.

You can use this block to compute the adaptive filter weights in applications such as system identification, inverse modeling, and filtered-x LMS algorithms, which are used in acoustic noise cancellation. For more details, see Bibliography.

Ports

Input

`Input` — Data input

scalar

Data input to the adaptive filter. The block accepts single-precision or double-precision floating point inputs. All inputs must be scalars and must have the same data type and precision.

Data Types: `single | double`

Complex Number Support: Yes

`Error` — Error

scalar

Error between the output signal and the desired signal.

Data Types: `single` | `double`
Complex Number Support: Yes

`Mu` — Filter adaptation step size
0.1 (default) | scalar in the range [0,1]

To enable this port, set **Step size source** to `Input port`.

Data Types: `single` | `double`

`Adapt` — Update filter weights
real scalar

When the input to this port is not zero, the block updates the filter weights. When the input to this port is 0, the filter weights do not change.

Data Types: `single` | `double` | `Boolean` | `int16` | `int32` | `int64` | `int8` | `uint16` | `uint32` | `uint64` | `uint8`

`Reset` — Reset filter weights
real scalar

When the input to this port is not zero, the block resets the filter weights to their initial values. When the input to this port is 0, the filter weights do not change.

Data Types: `single` | `double` | `Boolean` | `int16` | `int32` | `int64` | `int8` | `uint16` | `uint32` | `uint64` | `uint8`

Output

`Wts` — Filter weights
1-by-*n* row vector

The length of the filter weights vector is the value in the **Filter length** parameter.

Data Types: `single` | `double`

Parameters

`Algorithm` — LMS adaptive algorithm
`LMS` (default) | `Normalized LMS` | `Sign-Error LMS` | `Sign-Data LMS` | `Sign-Sign LMS`

The block uses one of the listed algorithms to compute the filter weights. For more details on the algorithms, see “Algorithms” on page 2-1076.

`Filter length` — Length of the filter

32 (default) | positive integer

Filter length specifies the length of the weights vector the block generates through the **Wts** output port.

`Step size source` — Method to specify the step size

`Property` (default) | `Input port`

- `Property` — Specify the filter adaptation size using the **Step size (mu)** parameter.
- `Input port` — Pass filter adaptation size using the **Mu** input port.

`Step size (mu)` — Size of the adaptation step

0.1 (default) | nonnegative scalar

Step size (mu) indicates the amount by which the filter weights are updated in each iteration. Choose an optimal step size so that the filter is stable and the convergence speed is optimal.

To enable this parameter, set **Step size source** to `Property`.

This parameter is tunable. You can change its value even during the simulation.

`Leakage factor (0 to 1)` — Leakage factor

1.0 (default) | real scalar in the range [0,1]

Leakage factor (0 to 1) prevents unbounded growth of the filter coefficients by reducing the drift of the coefficients from their optimum values. A leakage factor of `1.0` indicates no leakage. If you encounter coefficient drift, that is, large fluctuation about the optimum solution, decrease the leakage factor until the coefficient fluctuation becomes small.

This parameter is tunable. You can change its value even during the simulation.

`Initial value of filter weights` — Initial value of filter weights

0 (default) | real scalar

This parameter specifies the initial value of the filter weights, *w(n-1)*. The block uses this value to compute the weights, *w(n)*, when n = 1. For more details, see “Algorithms” on page 2-1076.

`Enable adapt input` — Update the filter weights
off (default) | on

When you select this check box, the **Adapt** input port appears on the block. When the input to this port is greater than 0, the block updates the filter weights. When the input to this port is less than or equal to 0, the filter weights do not change.

`Enable reset input` — Reset the filter weights
off (default) | on

When you select this check box, the **Reset** input port appears on the block. When the input to this port is greater than 0, the block resets the filter weights to their initial values. When the input to this port is less than or equal to 0, the filter weights do not change.

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Model Examples

Algorithms

The block computes filter weight estimates using $\mathbf{w}(n) = \alpha\mathbf{w}(n-1) + f(\mathbf{u}(n), e(n), \mu)$.

The function $f(\mathbf{u}(n), e(n), \mu)$ is defined according to the LMS algorithm you specify through the **Algorithm** parameter:

- **`LMS`** — $f(\mathbf{u}(n),e(n),\mu) = \mu e(n)\mathbf{u}^*(n)$
- **`Normalized LMS`** — $$f(\mathbf{u}(n),e(n),\mu) = \mu e(n)\frac{\mathbf{u}^*(n)}{\varepsilon + \mathbf{u}^H(n)\mathbf{u}(n)}$$

  In the `Normalized LMS` algorithm, ε is a small positive constant that overcomes the potential numerical instability in the update of weights.

  For double-precision floating-point inputs, ε is `2.2204460492503131e-016`. For single-precision floating-point inputs, ε is `1.192092896e-07`. For fixed-point input, ε is 0.
- **`Sign-Error LMS`** — $f(\mathbf{u}(n),e(n),\mu) = \mu\, sign(e(n))\mathbf{u}^*(n)$
- **`Sign-Data LMS`** — $f(\mathbf{u}(n),e(n),\mu) = \mu e(n)\, sign(\mathbf{u}(n))$, where **u**(n) is real
- **`Sign-Sign LMS`** — $f(\mathbf{u}(n),e(n),\mu) = \mu\, sign(e(n))\, sign(\mathbf{u}(n))$, where **u**(n) is real

In the previous equations:

- n — The current time index
- $\mathbf{u}(n)$ — The vector of buffered input samples at step n
- $\mathbf{u}^*(n)$ — The complex conjugate of the vector of buffered input samples at step n
- $\mathbf{w}(n)$ — The vector of filter weight estimates at step n
- $e(n)$ — The estimation error at step n
- μ — The adaptation step size
- α — The leakage factor ($0 \le \alpha \le 1$)

References

[1] Madisetti, Vijay, and Douglas Williams. "Introduction to Adaptive Filters." *The Digital Signal Processing Handbook.* Boca Raton, FL: CRC Press, 1999.

[2] Akhtar, M. T., M. Abe, M. Kawamata. "Modified-filtered-x LMS algorithm based active noise control systems with improved online secondary-path modeling." IEEE Symposium on Circuits and Systems, 2004.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Blocks
`Block LMS Filter` | `Fast Block LMS Filter` | `LMS Filter` | `RLS Filter`

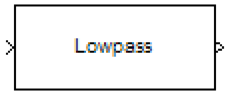
System Objects
dsp.AdaptiveLatticeFilter | dsp.BlockLMSFilter | dsp.KalmanFilter | dsp.LMSFilter | dsp.RLSFilter

Topics
“Adaptive Filters in Simulink”

Introduced in R2016b

Lowpass Filter

Design FIR or IIR lowpass filter



Library

Filtering / Filter Designs

`dspfdesign`

Description

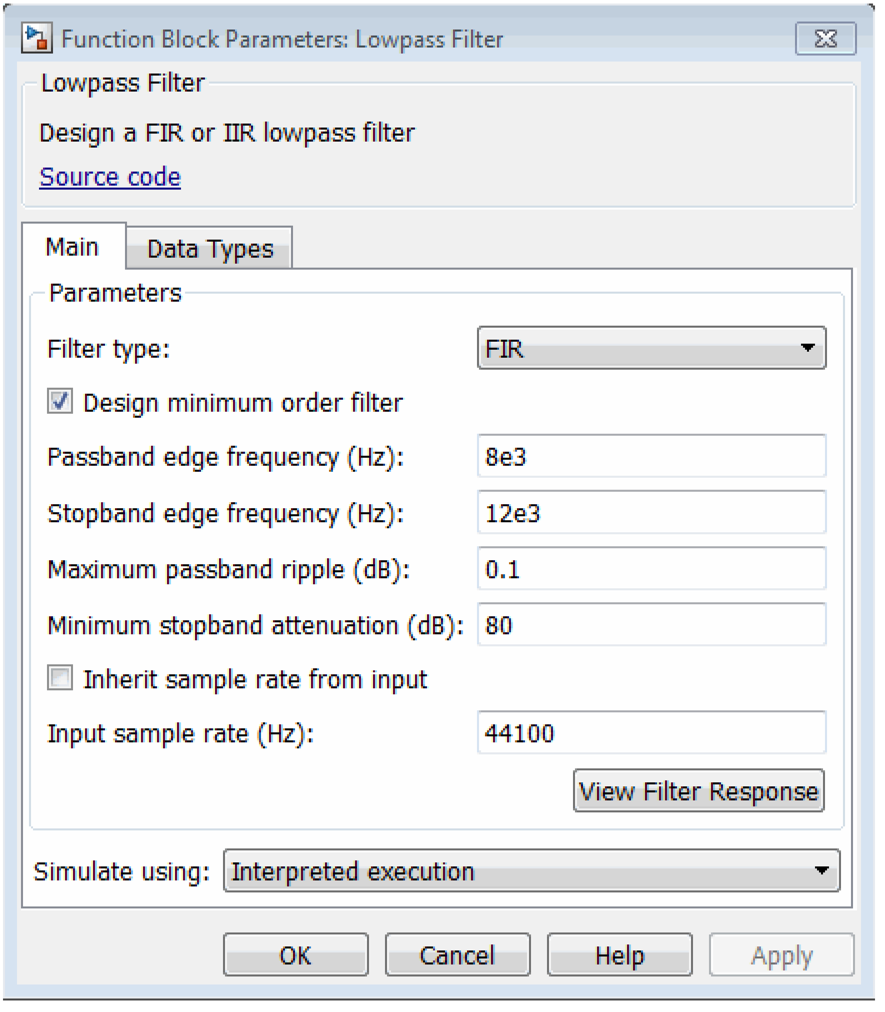
The Lowpass Filter block independently filters each channel of the input signal over time using the given design specifications.

You can set the **Filter type** parameter of the block to implement the block as an FIR or IIR lowpass filter. The block design the filter based on the parameters specified in the block dialog box.

The input signal can be a real- or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel.
The block supports fixed-point operations, HDL code generation, and ARM Cortex code generation. To learn more about ARM Cortex code generation, see "ARM Cortex-M and ARM Cortex-A Optimization"

Dialog Box

Main Tab



Parameters

Filter type

Type of lowpass filter.

- `FIR` (default) — Design an FIR lowpass filter.
- `IIR` — Design an IIR lowpass filter.

Design minimum order filter

When you select this check box, the block designs a filter with the minimum order and the specified passband, stopband frequency, passband ripple, and stopband attenuation. Set these specifications using the corresponding parameters. When you clear this check box, specify the order of the filter in **Filter order**.

By default, this check box is selected.

Filter order

Filter order of lowpass filter, specified as a positive scalar integer. You can specify the filter order only when **Design minimum order filter** check box is cleared. The default is `50`.

Passband edge frequency (Hz)

Passband edge frequency of the lowpass filter, specified as a real positive scalar in Hz. The passband edge frequency must be less than half the value of the **Input sample rate (Hz)**. The default is `8e3`.

Stopband edge frequency (Hz)

Stopband edge frequency of the lowpass filter, specified as a real positive scalar in Hz. The stopband edge frequency must be less than half the value of the **Input sample rate (Hz)**. You can specify the stopband edge frequency only when the **Design minimum order filter** is selected. The default is `12e3`.

Maximum passband ripple (dB)

Maximum ripple of the filter response in the passband, specified as a real positive scalar in dB. The default is `0.1`.

Minimum stopband attenuation (dB)

Minimum attenuation in the stopband, specified as a real positive scalar in dB. The default is `80`.

Inherit sample rate from input

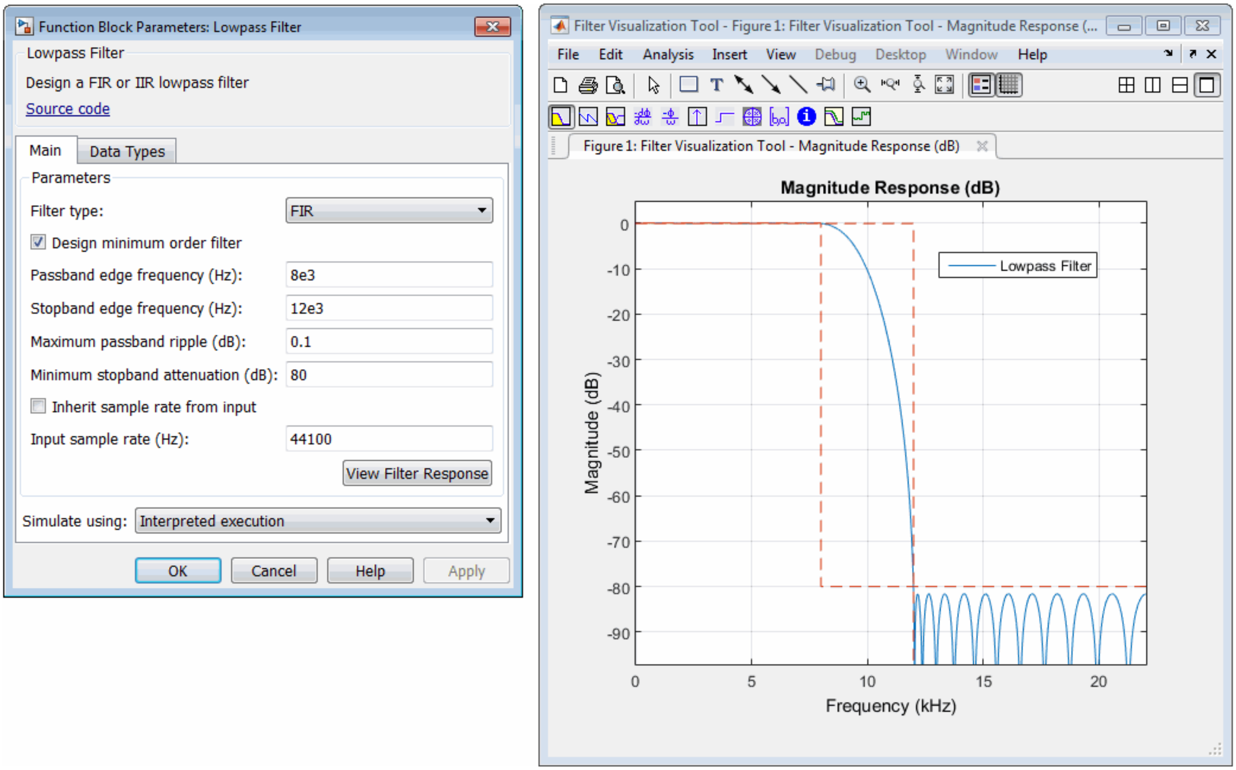
When you select this check box, the block inherits its sample rate from the input signal. When you clear this check box, you specify the sample rate in **Input sample rate (Hz)**.

Input sample rate (Hz)

Input sample rate, specified as a scalar in Hz. The default is `44100`.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the Lowpass Filter. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.



To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

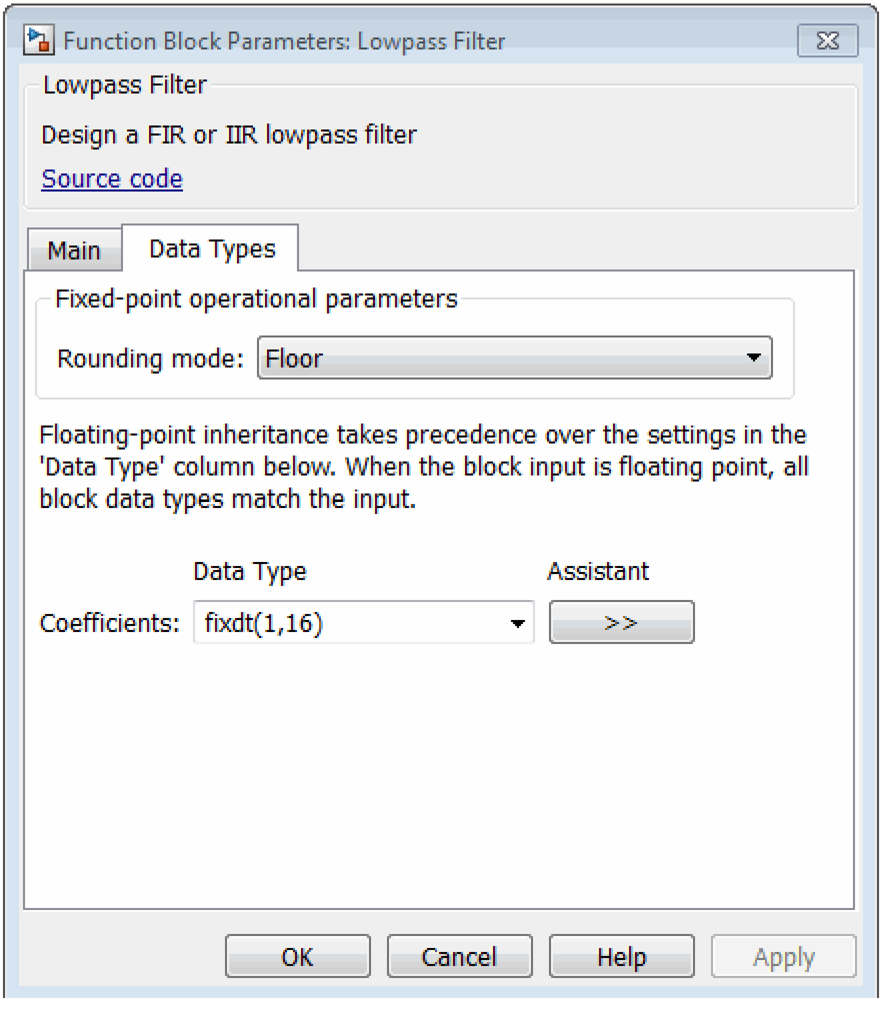
- `Interpreted execution` (default)

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

- `Code generation`

Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

Data Types Tab



Rounding mode

Rounding method for the output fixed-point operations. The rounding methods are `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, and `Zero`. The default is `Floor`.

Coefficients

Fixed-point data type of the coefficients, specified as one of the following:

- `fixdt(1,16)` (default) — Signed fixed-point data type of word length `16`, with binary point scaling. The block determines the fraction length automatically from the coefficient values in such a way that the coefficients occupy maximum representable range without overflowing.
- `fixdt(1,16,0)` — Signed fixed-point data type of word length `16` and fraction length, `0`. You can change the fraction length to any other integer value.
- `<data type expression>` — Specify the data type using an expression that evaluates to a data type object, for example, numeric type (`fixdt([ ],16, 15)`). Specify the sign mode of this data type as `[ ]` or `true`.
- `Refresh Data Type` — Refresh to the default data type.

Click the **Show data type assistant** button >> to display the data type assistant, which helps you set the stage input parameter.

See "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink) for more information.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed or unsigned)
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed or unsigned)

See Also

dsp.HighpassFilter	DSP System Toolbox
dsp.LowpassFilter	DSP System Toolbox
`Highpass Filter`	DSP System Toolbox

Algorithm

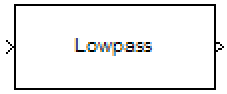
This block brings the capabilities of the dsp.LowpassFilter System object to the Simulink environment.

For information on the algorithms used by this block, see the Algorithms on page 4-1191 section of `dsp.LowpassFilter` system object.

Introduced in R2015b

Lowpass Filter (Obsolete)

Design lowpass filter



Note: The Lowpass Filter (Obsolete) block has been replaced by the `Lowpass Filter` block. Existing instances of the Lowpass Filter (Obsolete) block will continue to operate. For new models, use the Lowpass Filter block.

Library

Filtering / Filter Designs

`dspfdesign`

Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Parameters

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop-down list. `FIR` is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note: The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select `Minimum` (the default) or `Specify`. Selecting `Specify` enables the **Order** option so you can enter the filter order. When you set the **Impulse response** to `IIR`, you can specify different numerator and denominator orders. To specify a different denominator order, you must select the **Denominator order** check box.

Order

Enter the filter order. This option is enabled only if you set the **Order mode** to `Specify`.

Denominator order

Select this check box to specify a different denominator order. This option is enabled only if you set the **Impulse response** to `IIR` and the **Order mode** to `Specify`.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting `Sample-rate converter` activates both factors.

Decimation Factor

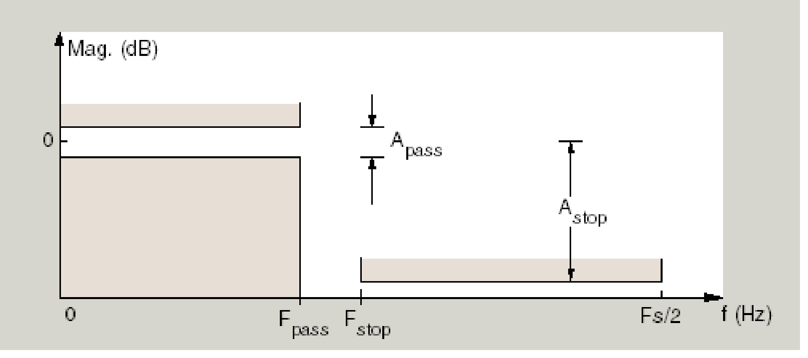
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to the one shown in the following figure.



In the figure, regions between specification values such as F_{pass} and F_{stop} represent transition regions where the filter response is not constrained.

Frequency constraints

When **Order mode** is `Specify`, select the filter features that the block uses to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and Stopband frequencies` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- `Passband frequency` — Define the filter by specifying the edge of the passband.
- `Stopband frequency` — Define the filter by specifying the edge of the stopband.

- `Hafband power (3dB) frequency` — Define the filter response by specifying the location of the 3 dB point. The 3 dB point is the frequency for the point three decibels below the passband value.
- `Cutoff (6dB) frequency` — For FIR filters, define the filter response by specifying the location of the 6 dB point. The 6 dB point is the frequency for the point six decibels below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0—1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input sample rate** parameter.

Input sample rate

Input sample rate, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Passband frequency

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Stopband frequency

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Half power (3dB) frequency

When **Frequency constraints** is `Half power (3dB) frequency`, specify the frequency of the 3 dB point. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Cutoff (6dB) frequency

When **Frequency constraints** is `Cutoff (6dB) frequency`, specify the frequency of the 6 dB point. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude constraints

This option is only available when you specify the order of your filter design. Depending on the setting of the **Frequency constraints** parameter, some combination of the following options will be available for the **Magnitude constraints** parameter: `Unconstrained`, and `Passband ripple and stopband attenuation`.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default)

Passband ripple

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Stopband attenuation

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually `Elliptic`, and the default FIR method is `Equiripple`.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Specify the phase constraint of the filter as `Linear`, `Maximum`, or `Minimum`.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent *n* in the relation $(1/f)^n$ to define the stopband decay. The block applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters (Impulse response: IIR).

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note: The `Inherited (this choice will be removed — see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

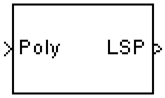
Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2006b

LPC to LSF/LSP Conversion

Convert linear prediction coefficients to line spectral pairs or line spectral frequencies



Library

Estimation / Linear Prediction

`dsplp`

Description

The LPC to LSF/LSP Conversion block takes a vector or matrix of linear prediction coefficients (LPCs) and converts it to a vector or matrix of line spectral pairs (LSPs) or line spectral frequencies (LSFs). When converting LPCs to LSFs, the block outputs match those of the `poly2lsf` function.

The block input must be either a matrix, a column vector, or an unoriented vector. Each channel of the input must have at least two samples.

The input LPCs for each channel, 1, a_1, a_2, ..., a_m, must be the denominator of the transfer function of a stable all-pole filter with the form given in the first equation of “Requirements for Valid Outputs” on page 2-1095. A length-M+1 input channel yields a length-M output channel.

See other sections of this reference page to learn about how to ensure that you get valid outputs, how to detect invalid outputs, how the block computes the LSF/LSP values, and more.

Requirements for Valid Outputs

To get valid outputs, your inputs and the **Root finding coarse grid points** parameter value must meet these requirements:

- The input LPCs for each channel, 1, a_1, a_2, ..., a_m, must come from the denominator of the following transfer function, *H(z)*, of a stable all-pole filter (all roots of *H(z)* must be inside the unit circle). Note that the first term in *H(z)*'s denominator must be 1. When the input LPCs do not come from a transfer function of the following form, the block outputs are invalid.

$$H(z) = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \ \ldots \ + a_m z^{-m}}$$

- The **Root finding coarse grid points** parameter value must be large enough so that the block can find all the LSP or LSF values. (The output LSFs and LSPs are roots of polynomials related to the input LPC polynomial; the block looks for these roots to produce the output. For details, see "LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding" on page 2-1101.) When you do not set **Root finding coarse grid points** to a high enough value relative to the number of LPCs, the block might not find all the LSPs or LSFs and yield invalid outputs as described in "Root Finding Method Limitations: Failure to Find Roots" on page 2-1105.

To learn about recognizing invalid inputs and outputs and parameters for dealing with them, see "Handling and Recognizing Invalid Inputs and Outputs" on page 2-1097.

Setting Outputs to LSFs or LSPs

Set the **Output** parameter to one of the following settings to determine whether the block outputs LSFs or LSPs:

- `LSF in radians (0 pi)` — Block outputs the LSF values between 0 and π radians in increasing order. The block does not output the guaranteed LSF values, 0 and π.
- `LSF normalized in range (0 0.5)` — Block outputs normalized LSF values in increasing order, computed by dividing the LSF values between 0 and π radians by 2π. The block does not output the guaranteed normalized LSF values, 0 and 0.5.
- `LSP in range (-1 1)` — Block outputs LSP values in decreasing order, equal to the cosine of the LSF values between 0 and π radians. The block does not output the guaranteed LSP values, -1 and 1.

Adjusting Output Computation Time and Accuracy with Root Finding Parameters

The values n and k determine the block's output computation time and accuracy, where

- n is the value of the **Root finding coarse grid points** parameter (choose this value with care; see the note below).
- k is the value of the **Root finding bisection refinement** parameter.
- Decreasing the values of n and k decreases the output computation time, but also decreases output accuracy:
  - The upper bound of block's computation time is proportional to $k \cdot (n-1)$.
  - Each LSP output is within $1/(n \cdot 2^k)$ of the actual LSP value.
  - Each LSF output is within ΔLSF of the actual LSF value, LSF_{act}, where

$$\Delta LSF = \left| a\cos\left(LSF_{act}\right) - a\cos\left(LSF_{act} + 1/\left(n \cdot 2^k\right)\right)\right|$$

Note When the value of the **Root finding coarse grid points** parameter is too small relative to the number of LPCs, the block might output invalid data as described in "Requirements for Valid Outputs" on page 2-1095. Also see "Handling and Recognizing Invalid Inputs and Outputs" on page 2-1097.

Notable Input and Output Properties

- To get valid outputs, your input LPCs and the value of the **Root finding coarse grid points** parameter must meet the requirements described in "Requirements for Valid Outputs" on page 2-1095.
- Length-L+1 input channel yields length-L output channel
- **Output** parameter determines the output type (see "Setting Outputs to LSFs or LSPs" on page 2-1096):
  - LSFs — frequencies, w_k, where $0 < w_k < \pi$ and $w_k < w_{k+1}$
  - Normalized LSFs — $w_k / 2\pi$
  - LSPs — $\cos(w_k)$

Handling and Recognizing Invalid Inputs and Outputs

The block outputs invalid data when your input LPCs and the value of the **Root finding coarse grid points** parameter do not meet the requirements described in

“Requirements for Valid Outputs” on page 2-1095. The following topics describe what invalid outputs look like, and how to set the block parameters provided for handling invalid inputs and outputs:

- “What Invalid Outputs Look Like” on page 2-1098
- “Parameters for Handling Invalid Inputs and Outputs” on page 2-1098

What Invalid Outputs Look Like

The channels of an invalid output have the same dimensions, sizes, and frame statues as the channels of a valid output. However, invalid output channels do not contain all the LSP or LSF values. Instead, they contain none or some of the LSP and LSF values and the rest of the output is filled with place holder values (-1, 0.5, or π) depending on the **Output** parameter setting).

In short, all invalid outputs in a channel end in one of the place holder values (-1, 0.5, or π) as illustrated in the following table. To learn how to use the block's parameters for handling invalid inputs and outputs, see the next section.

Output Parameter Setting	Place Holder	Sample Invalid Outputs
`LSF in radians (0 pi)`	π	$\begin{bmatrix} w_1 & w_2 & w_3 & \pi & \pi & \pi & \pi & \pi \end{bmatrix}$
`LSF normalized in range (0 0.5)`	`0.5`	$\begin{bmatrix} w_1 \\ w_2 \\ 0.5 \end{bmatrix}$
`LSP in range (-1 1)`	`-1`	$\begin{bmatrix} \cos(w_{13}) \\ \cos(w_{23}) \\ -1 \\ -1 \\ -1 \end{bmatrix}$

Parameters for Handling Invalid Inputs and Outputs

You must set how the block handles invalid inputs and outputs by setting these parameters:

- **Show output validity status (1=valid, 0=invalid)** — Set this parameter to activate a second output port that outputs a vector with one Boolean element per channel; `1` when the output of the corresponding channel is valid, and `0` when the output is invalid. The LSF and LSP outputs are invalid when the block fails to find all the LSF or LSP values or when the input LPCs are unstable (for details, see "Requirements for Valid Outputs" on page 2-1095). See the previous section to learn how to recognize invalid outputs.
- **If current output is invalid, overwrite with previous output** — Select this check box to cause the block to overwrite invalid outputs with the previous output. When you set this parameter you also need to consider these parameters:
  - **When first output is invalid, overwrite with user-defined values** — When the first input is unstable, you can overwrite the invalid first output with either
    - The default values, by clearing this check box
    - Values you specify, by selecting this check box

    The default initial overwrite values are the LSF or LSP representations of an all-pass filter. The vector that is used to overwrite invalid output is stored as an internal state.
  - **User-defined LSP/LSF values for overwriting invalid first output** — Specify a vector of values for overwriting an invalid first output if you selected the **When first output is invalid, overwrite with user-defined values** parameter. For multichannel inputs, provide a matrix with the same number of channels as the input, or one vector that will be applied to every channel. The vector or matrix of LSP/LSF values you specify should have the same dimension, size, and frame status as the other outputs.
- **If first input value is not 1** — The block output in any channel is invalid when the first coefficient in an LPC vector is not 1; this parameter determines what the block does when given such inputs:
  - `Ignore` — Proceed with computations as if the first coefficient is 1.
  - `Normalize` — Divide the input LPCs by the value of the first coefficient before computing the output.
  - `Normalize and warn` — In addition to `Normalize`, display a warning message at the MATLAB command line.
  - `Error` — Stop the simulation and display an error message at the MATLAB command line.

Parameters

Output

Specifies whether to convert the input linear prediction polynomial coefficients (LPCs) to `LSP in range (-1 1)`, `LSF in radians (0 pi)`, or `LSF normalized in range (0 0.5)`. See "Setting Outputs to LSFs or LSPs" on page 2-1096 for descriptions of the three settings.

Root finding coarse grid points

The value n, where the block divides the interval (-1, 1) into n subintervals of equal length, and looks for roots (LSP values) in each subinterval. You must pick n large enough or the block output might be invalid as described in "Requirements for Valid Outputs" on page 2-1095. To learn how the block uses this parameter to compute the output, see "LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding" on page 2-1101. Also see "Adjusting Output Computation Time and Accuracy with Root Finding Parameters" on page 2-1096. Tunable (Simulink).

Root finding bisection refinement

The value k, where each LSP output is within $1/(n \cdot 2^k)$ of the actual LSP value, where n is the value of the **Root finding coarse grid points** parameter. To learn how the block uses this parameter to compute the output, see "LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding" on page 2-1101. Also see "Adjusting Output Computation Time and Accuracy with Root Finding Parameters" on page 2-1096. Tunable (Simulink).

Show output validity status

Set this parameter to activate a second output port that outputs a vector with one Boolean element per channel; `1` when the output of the corresponding channel is valid, and `0` when the output is invalid. The LSF and LSP outputs are invalid when the block fails to find all the LSF or LSP values or when the input LPCs are unstable (for details, see "Requirements for Valid Outputs" on page 2-1095).

If current output is invalid, overwrite with previous output

Selecting this check box causes the block to overwrite invalid outputs with the previous output. Setting this parameter activates other parameters for taking care of initial overwrite values (when the very first output of the block is invalid). For more information, see "Parameters for Handling Invalid Inputs and Outputs" on page 2-1098.

When first output is invalid, overwrite with user-defined values

When the first input is unstable, you can overwrite the invalid first output with either

- The default values, by clearing this check box
- Values you specify, by selecting this check box

The default initial overwrite values are the LSF or LSP representations of an all-pass filter. The vector that is used to overwrite invalid output is stored as an internal state. For more information, see "Parameters for Handling Invalid Inputs and Outputs" on page 2-1098.

User-defined LSP/LSF values for overwriting invalid first output

Specify a vector of values for overwriting an invalid first output if you selected the **When first output is invalid, overwrite with user-defined values** parameter. For multichannel inputs, provide a matrix with the same number of channels as the input, or one vector that will be applied to every channel. The vector or matrix of LSP/LSF values you specify should have the same dimension, size, and frame status as the other outputs.

If first input value is not 1

Determines what the block does when the first coefficient of an input is not `1`. The block can either proceed with computations as when the first coefficient is 1 (`Ignore`); divide the input LPCs by the value of the first coefficient before computing the output (`Normalize`); in addition to `Normalize`, display a warning message at the MATLAB command line (`Normalize and warn`); stop the simulation and display an error message at the MATLAB command line (`Error`). For more information, see "Parameters for Handling Invalid Inputs and Outputs" on page 2-1098.

Theory

LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding

---

Note To learn the principles on which the block's LSP and LSF computation method is based, see the reference listed in "References" on page 2-1107.

To compute LSP outputs for each channel, the block relies on the fact that LSP values are the roots of two particular polynomials related to the input LPC polynomial; the block finds these roots using the Chebyshev polynomial root finding method, described next. To compute LSF outputs, the block computes the arc cosine of the LSPs, outputting values ranging from 0 to π radians.

Root Finding Method

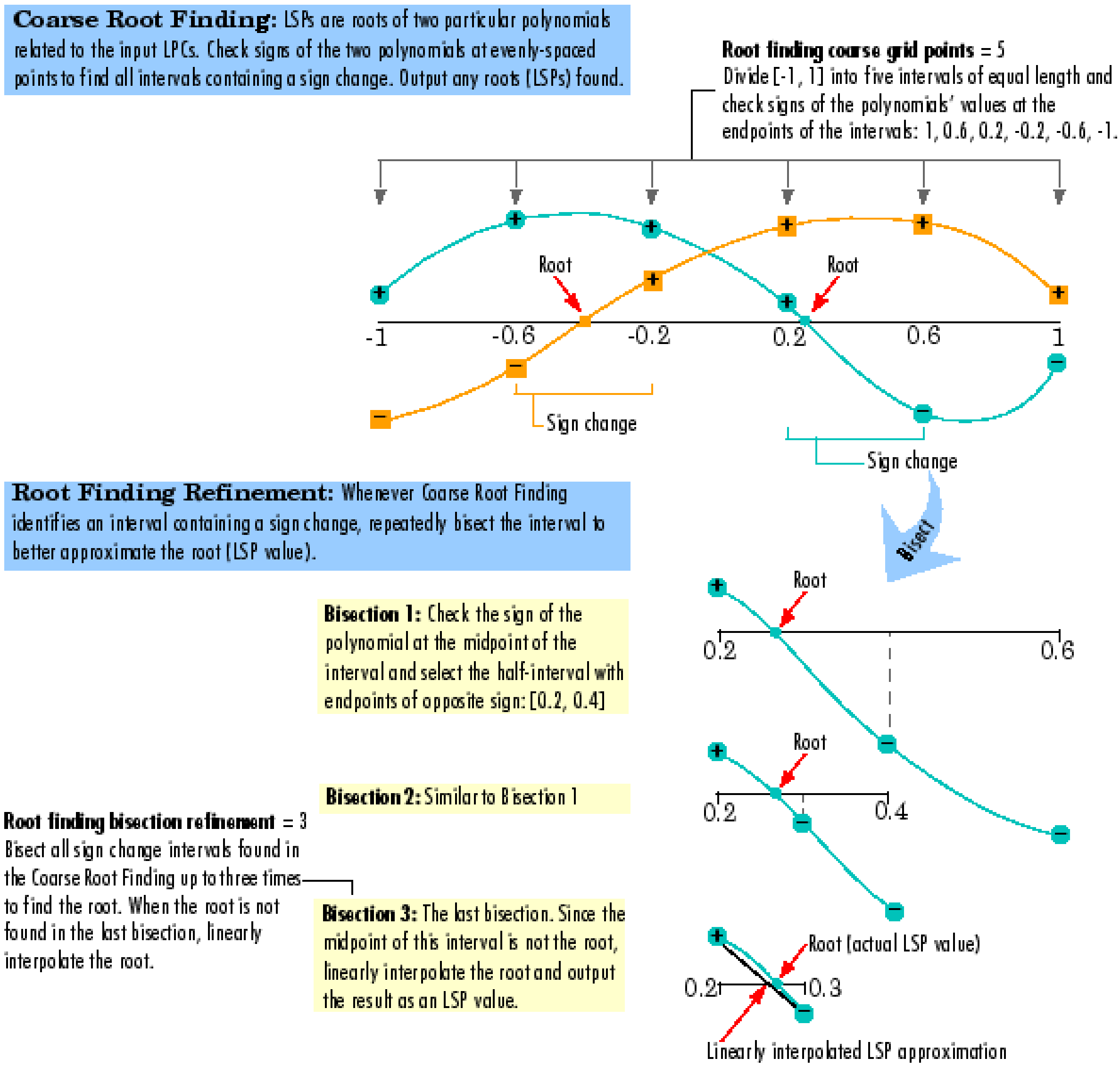
LSPs, which are the roots of two particular polynomials, always lie in the range (-1, 1). (The guaranteed roots at 1 and -1 are factored out.) The block finds the LSPs by looking for a sign change of the two polynomials' values between points in the range (-1, 1). The block searches a maximum of $k(n-1)$ points, where

- n is the value of the **Root finding coarse grid points** parameter.
- k is the value of the **Root finding bisection refinement** parameter.

The block's method for choosing which points to check consists of the following two steps:

1. **Coarse Root Finding** —- The block divides the interval [-1, 1] into n intervals, each of length $2/n$, and checks the signs of both polynomials' values at the endpoints of the intervals. The block starts checking signs at 1, and continues checking signs at $1 - 4/n$, $1 - 6/n$, and so on at steps of length $2/n$, outputting any point if it is a root. The block stops searching in these situations:
   a. The block finds a sign change of a polynomial's values between two adjacent points. An interval containing a sign change is guaranteed to contain a root, so the block further searches the interval as described in Step 2, Root Finding Refinement.
   b. The block finds and outputs all M roots (given a length-M+1 LPC input).
   c. The block fails to find all M roots and yields invalid outputs as described in "Handling and Recognizing Invalid Inputs and Outputs" on page 2-1097.
2. **Root Finding Refinement** — When the block finds a sign change in an interval, $[a, b]$, it searches for the root guaranteed to lie in the interval by following these steps:
   a. **Check if Midpoint Is a Root** — The block checks the sign of the midpoint of the interval $[a, b]$. The block outputs the midpoint if it is a root, and continues Step 1, Coarse Root Finding, at the next point, $a - 2/n$. Otherwise, the block selects the half-interval with endpoints of opposite sign (either $[a, (a + b)/2]$ or $[(a + b)/2, b]$) and executes Step 2b, Stop or Continue Root Finding Refinement.

b **Stop or Continue Root Finding Refinement** — When the block has repeated Step 2a k times (k is the value of the **Root finding bisection refinement** parameter), the block linearly interpolates the root by using the half-interval's endpoints, outputs the result as an LSP value, and returns to Step 1, Coarse Root Finding. Otherwise, the block repeats Step 2a using the half-interval.



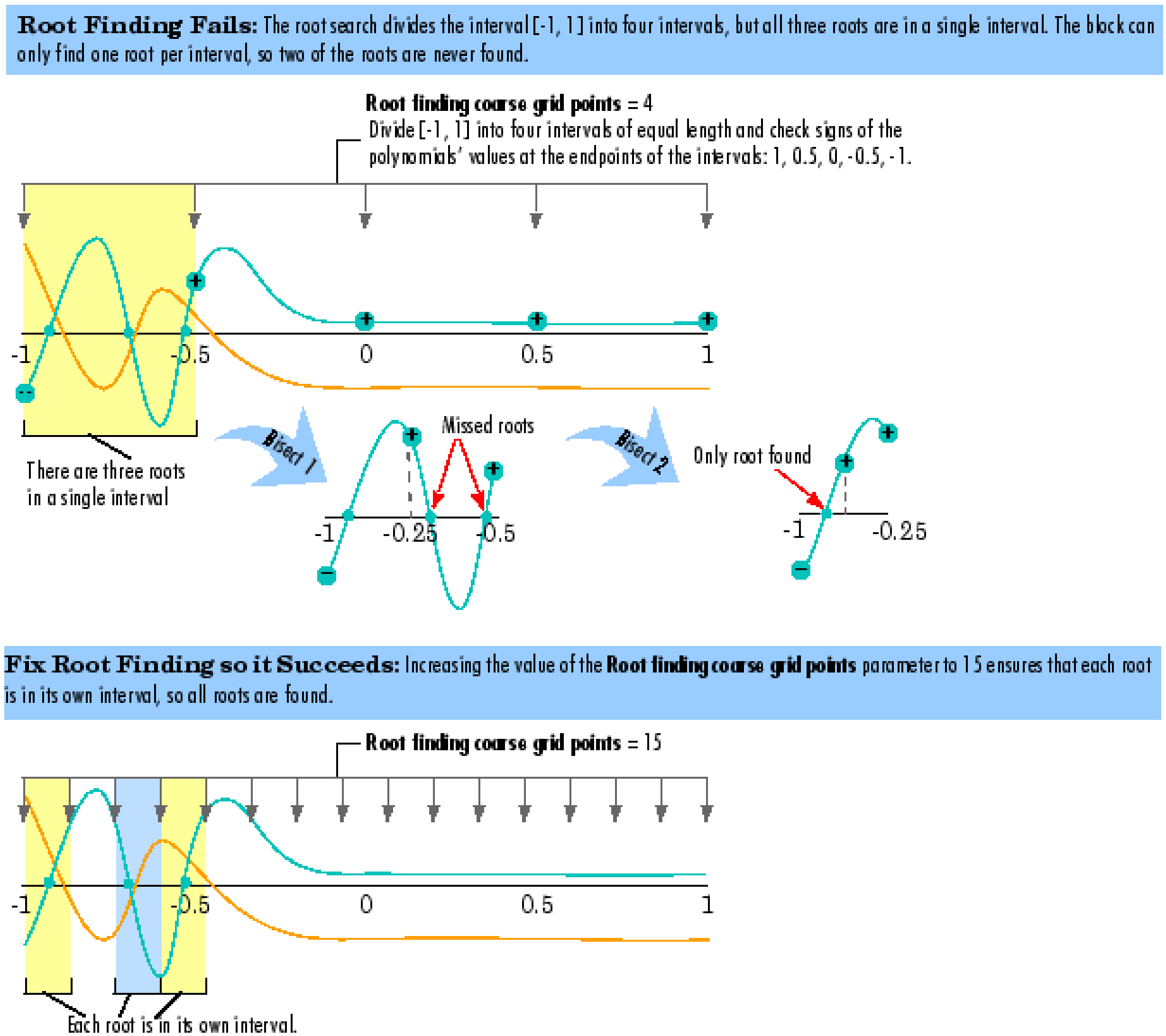
Coarse Root Finding and Root Finding Refinement

Root Finding Method Limitations: Failure to Find Roots

The block root finding method described above can fail, causing the block to produce invalid outputs (for details on invalid outputs, see "Handling and Recognizing Invalid Inputs and Outputs" on page 2-1097).

In particular, the block can fail to find some roots if the value of the **Root finding coarse grid points** parameter, *n*, is too small. If the polynomials oscillate quickly and have roots that are very close together, the root finding might be too coarse to identify roots that are very close to each other, as illustrated in Fixing a Failed Root Finding.

For higher-order input LPC polynomials, you should increase the **Root finding coarse grid points** value to ensure the block finds all the roots and produces valid outputs.



Fixing a Failed Root Finding

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean — Supported only by the optional output port that appears when you set the parameter, **Show output validity status (1=valid, 0=invalid)**

References

Kabal, P. and Ramachandran, R. "The Computation of Line Spectral Frequencies Using Chebyshev Polynomials."*IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-34 No. 6, December 1986. pp. 1419-1426.

See Also

`LSF/LSP to LPC Conversion`	DSP System Toolbox
`LPC to/from RC`	DSP System Toolbox
`LPC/RC to Autocorrelation`	DSP System Toolbox
`poly2lsf`	Signal Processing Toolbox

Introduced before R2006a

LSF/LSP to LPC Conversion

Convert line spectral frequencies or line spectral pairs to linear prediction coefficients

Library

Estimation / Linear Prediction

`dsplp`

Description

The LSF/LSP to LPC Conversion block takes a vector or matrix of line spectral pairs (LSPs) or line spectral frequencies (LSFs) and converts it to a vector or matrix of linear prediction polynomial coefficients (LPCs). When converting LSFs to LPCs, the block outputs match those of the `lsf2poly` function.

The block input can be an N-by-M matrix or an unoriented vector. Each column of the matrix is treated as a channel. When the input is an unoriented vector, the input is treated as one channel. Each input channel must be in the same format, which you specify in the **Input** parameter:

- `LSF in range (0 pi)` — Vector of LSF values between 0 and π radians in increasing order. Do not include the guaranteed LSF values, 0 and π.
- `LSF normalized in range (0 0.5)` — Vector of normalized LSF values in increasing order, (compute by dividing the LSF values between 0 and π radians by 2π). Do not include the guaranteed normalized LSF values, 0 and 0.5.
- `LSP in range (-1 1)` — Vector of LSP values in decreasing order, equal to the cosine of the LSF values between 0 and π radians. Do not include the guaranteed LSP values, -1 and 1.

Parameters

Input

Specifies whether to convert `LSP in range (-1 1)`, `LSF in range (0 pi)`, or `LSF normalized in range (0 0.5)` to linear prediction coefficients (LPCs).

Supported Data Types

- Double-precision floating point
- Single-precision floating point

References

Kabal, P. and Ramachandran, R. "The Computation of Line Spectral Frequencies Using Chebyshev Polynomials." *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-34 No. 6, December 1986. pp. 1419-1426.

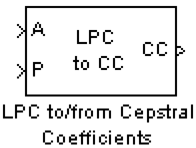
See Also

`LPC to LSF/LSP Conversion`	DSP System Toolbox
`LPC to/from RC`	DSP System Toolbox
`LPC/RC to Autocorrelation`	DSP System Toolbox
`lsf2poly`	Signal Processing Toolbox

Introduced before R2006a

LPC to/from Cepstral Coefficients

Convert linear prediction coefficients to cepstral coefficients or cepstral coefficients to linear prediction coefficients



Library

Estimation / Linear Prediction

`dsplp`

Description

The LPC to/from Cepstral Coefficients block either converts linear prediction coefficients (LPCs) to cepstral coefficients (CCs) or cepstral coefficients to linear prediction coefficients. Set the **Type of conversion** parameter to `LPCs to cepstral coefficients` or `Cepstral coefficients to LPCs` to select the domain into which you want to convert your coefficients. The LPC port corresponds to LPCs, and the CC port corresponds to the CCs. For more information, see "Algorithm" on page 2-1111.

The block input can be an N-by-M matrix or an unoriented vector. Each column of the matrix is treated as a channel. When the input is an unoriented vector, the input is treated as one channel.

Consider a signal $x(n)$ as the input to an FIR analysis filter represented by LPCs. The output of this analysis filter, $e(n)$, is known as the prediction error signal. The power of this error signal is denoted by P, the prediction error power.

When you select `LPCs to cepstral coefficients` from the **Type of conversion** list, you can specify the prediction error power in two ways. From the **Specify P** list, choose `via input port` to input the prediction error power using input port P. The input to the port must be a vector with length equal to the number of input channels. Select `assume P equals 1` to set the prediction error power equal to 1 for all channels.

When you select `LPCs to cepstral coefficients` from the **Type of conversion** list, the **Output size same as input size** check box appears. When you select this check box, the length of the input vector of LPCs is equal to the output vector of CCs. When you do not select this check box, enter a positive scalar for the **Length of output cepstral coefficients** parameter.

When you select `LPCs to cepstral coefficients` from the **Type of conversion** list, you can use the **If first input value is not 1** parameter to specify the behavior of the block when the first coefficient of the LPC vector is not 1. The following options are available:

- `Replace it with 1` —- Changes the first value of the coefficient vector to 1. The other coefficient values are unchanged.
- `Normalize` — Divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC vector is 1.
- `Normalize and Warn` — Divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC vector is 1. The block displays a warning message telling you that your vector of coefficients has been normalized.
- `Error` — Displays an error telling you that the first coefficient of the LPC vector is not 1.

When you select `Cepstral coefficients to LPCs` from the **Type of conversion** list, the **Output P** check box appears on the block. Select this check box when you want to output the prediction error power from output port P.

Algorithm

The cepstral coefficients are the coefficients of the Fourier transform representation of the logarithm magnitude spectrum. Consider a sequence, $x(n)$, having a Fourier transform $X(\omega)$. The cepstrum, $c_x(n)$, is defined by the inverse Fourier transform of $C_x(\omega)$, where $C_x(\omega) = \log_e X(\omega)$. See the Real Cepstrum block reference page for information on computing cepstrum coefficients from time-domain signals.

LPC to CC

When in this mode, this block uses a recursion technique to convert LPCs to CCs. The LPC vector is defined by $\begin{bmatrix} a_0 & a_1 & a_2 & \dots & a_p \end{bmatrix}$ and the CC vector is defined by $\begin{bmatrix} c_0 & c_1 & c_2 & \dots & c_p & \dots & c_{n-1} \end{bmatrix}$. The recursion is defined by the following equations:

$$c_0 = \log_e P$$

$$c_m = -a_m + \frac{1}{m} \sum_{k=1}^{m-1} \left[-(m-k) \cdot a_k \cdot c_{(m-k)} \right], 1 \le m \le p$$

$$c_m = \sum_{k=1}^{p} \left[\frac{-(m-k)}{m} \cdot a_k \cdot c_{(m-k)} \right], p < m < n$$

CC to LPC

When in this mode, this block uses a recursion technique to convert CCs to LPCs. The CC vector is defined by $\begin{bmatrix} c_0 & c_1 & c_2 & \dots & c_p & \dots & c_n \end{bmatrix}$ and the LPC vector is defined by $\begin{bmatrix} a_0 & a_1 & a_2 & \dots & a_p \end{bmatrix}$. The recursion is defined by the following equations

$$a_m = -c_m - \frac{1}{m} \sum_{k=1}^{m-1} \left[(m-k) \cdot c_{(m-k)} \cdot a_k \right]$$

$$P = \exp(C_0)$$

where $m = 1, 2, \dots, p$.

Parameters

Type of conversion

Choose `LPCs to cepstral coefficients` or `Cepstral coefficients to LPCs` to specify the domain into which you want to convert your coefficients.

Specify P

Choose `via input port` to input the values of prediction error power using input port P. Select `assume P equals 1` to set the prediction error power equal to 1.

Output size same as input size

When you select this check box, the length of the input vector of LPCs is equal to the output vector of CCs.

Length of output cepstral coefficients

Enter a positive scalar that is the length of each output channel of CCs.

If first input value is not 1

Select what you would like the block to do when the first coefficient of the LPC vector is not 1. You can choose `Replace it with 1`, `Normalize`, `Normalize and Warn`, and `Error`.

Output P

Select this check box to output the prediction error power for each channel from output port P.

References

Papamichalis, Panos E. *Practical Approaches to Speech Coding*. Englewood Cliffs, NJ: Prentice Hall, 1987.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

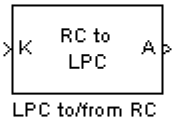
`Levinson-Durbin`	DSP System Toolbox
`LPC to LSF/LSP Conversion`	DSP System Toolbox

`LSF/LSP to LPC Conversion`	DSP System Toolbox
`LPC to/from RC`	DSP System Toolbox
`LPC/RC to Autocorrelation`	DSP System Toolbox
`Real Cepstrum`	DSP System Toolbox
`Complex Cepstrum`	DSP System Toolbox

Introduced before R2006a

LPC to/from RC

Convert linear prediction coefficients to reflection coefficients or reflection coefficients to linear prediction coefficients



Library

Estimation / Linear Prediction

`dsplp`

Description

The LPC to/from RC block either converts linear prediction coefficients (LPCs) to reflection coefficients (RCs) or reflection coefficients to linear prediction coefficients. Set the **Type of conversion** parameter to `LPC to RC` or `RC to LPC` to select the domain into which you want to convert your coefficients. The A port corresponds to LPC coefficients, and the K port corresponds to the RC coefficients. For more information, see "Algorithm" on page 2-1116.

The block input can be an N-by-M matrix or an unoriented vector. Each column of the matrix is treated as a channel. When the input is an unoriented vector, the input is treated as one channel.

Consider a signal $x(n)$ as the input to an FIR analysis filter represented by LPC coefficients. The output of the analysis filter, $e(n)$, is known as the prediction error signal. The power of this error signal is denoted by P. When the zero lag autocorrelation coefficient of $x(n)$ is one, the autocorrelation sequence and prediction error power are said to be normalized.

Select the **Output normalized prediction error power** check box to enable port P. The normalized prediction error power output at `P` is a vector with one element per input channel. Each element varies between zero and one.

Select the **Output LPC filter stability** check box to output the stability of the filter represented by the LPCs or RCs. The synthesis filter represented by the LPCs is stable when the absolute value of each of the roots of the LPC polynomial is less than one. The lattice filter represented by the RCs is stable when the absolute value of each reflection coefficient is less than 1. When the filter is stable, the block outputs a Boolean value of `1` for each input channel at the S port. When the filter is unstable, the block outputs a Boolean value of `0` for each input channel at the S port.

If first input value is not 1 parameter specifies the behavior of the block when the first coefficient of the LPC coefficient vector in any channel is not 1. The following options are available:

- `Replace it with 1` — Changes the first value of the coefficient channel to 1. The other coefficient values are unchanged.
- `Normalize` — Divides the entire channel of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1.
- `Normalize and Warn` — Divides the entire channel of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1. The block displays a warning message telling you that your vector of coefficients has been normalized.
- `Error` — Displays an error telling you that the first coefficient of the LPC coefficient channel is not 1.

Algorithm

LPC to RC

When in this mode, this block uses backward Levinson recursion to convert linear prediction coefficients (LPCs) to reflection coefficients (RCs). For a given Nth order LPC vector $LPC_N = [\; 1 \quad a_{N1} \quad a_{N2} \quad \ldots \quad a_{NN} \;]$, the block calculates the Nth reflection coefficient value using the formula $\gamma_N = -a_{NN}$. The block then finds the lower order LPC vectors , LPC_{N-1}, LPC_{N-2}, ..., LPC_1, using the following recursion.

for $p = N, N-1, \ldots, 2,$

$$\gamma_p = a_{pp}$$
$$F = 1 - {\gamma_p}^2$$
$$a_{p-1,m} = \frac{a_{p,m}}{F} - \frac{\gamma_p a_{p,p-m}}{F}, \quad 1 \le m < p$$

end

Finally, $\gamma_1 = -a_{11}$. The reflection coefficient vector is $[\gamma_1, \quad \gamma_2, \quad \ldots, \quad \gamma_N]$.

RC to LPC

When in this mode, this block uses Levinson recursion to convert reflection coefficients (RCs) to linear prediction coefficients (LPCs). In this case, the input to the block is $RC = [\gamma_1 \quad \gamma_2 \quad \ldots \quad \gamma_N]$. The zeroth order LPC vector term is 1. Starting with this term, the block uses recursion to calculate the higher order LPC vectors, $LPC_2, LPC_3, \ldots LPC_N$, until it has calculated the entire LPC matrix.

$$LPC_{matrix} = \begin{bmatrix} LPC_0 \\ LPC_1 \\ LPC_2 \\ \ldots \\ LPC_N \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & a_{11} & 0 & 0 & \cdots & 0 \\ 1 & a_{21} & a_{22} & 0 & \cdots & 0 \\ 1 & a_{31} & a_{32} & a_{33} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & a_{N1} & a_{N2} & a_{N3} & \cdots & a_{NN} \end{bmatrix}$$

This LPC matrix consists of LPC vectors of order 0 through N found by using the Levinson recursion. The following are the formulas for the recursion steps, for $p = 0$, 1, ...,$N - 1$.

$$a_{p+1,m} = a_{p,m} + \gamma_{p+1} a_{p,p+1-m}, \quad 1 \le m \le p$$
$$a_{p+1,p+1} = \gamma_{p+1}$$

Parameters

Type of conversion

Select `LPC to RC` or `RC to LPC` to select the domain into which you want to convert your coefficients.

Output normalized prediction error power

Select this check box to output the normalized prediction error power at port P.

Output LPC filter stability

Select this check box to output the stability of the filter. When the filter represented by the LPCs or RCs is stable, the block outputs a Boolean value of `1` for each input channel at the S port. When the filter represented by the LPCs or RCs is unstable, the block outputs a Boolean value of `0` for each input channel at the S port.

If first input value is not 1

Select what you would like the block to do when the first coefficient of the LPC coefficient vector is not 1. You can choose `Replace it with 1`, `Normalize`, `Normalize and Warn`, and `Error`.

References

Makhoul, J *Linear Prediction: A tutorial review.* Proc. IEEE. 63, 63, 56 (1975).

Markel, J.D. and A. H. Gray, Jr., *Linear Prediction of Speech.* New York, Springer-Verlag, 1976.

Supported Data Types

- Double-precision floating-point
- Single-precision floating-point

See Also

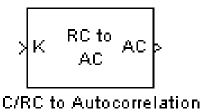
`Levinson-Durbin`	DSP System Toolbox
`LPC to LSF/LSP Conversion`	DSP System Toolbox

`LSF/LSP to LPC Conversion`	DSP System Toolbox
`LPC/RC to Autocorrelation`	DSP System Toolbox

Introduced before R2006a

LPC/RC to Autocorrelation

Convert linear prediction coefficients or reflection coefficients to autocorrelation coefficients



Library

Estimation / Linear Prediction

`dsplp`

Description

The LPC/RC to Autocorrelation block either converts linear prediction coefficients (LPCs) to autocorrelation coefficients (ACs) or reflection coefficients (RCs) to autocorrelation coefficients (ACs). Set the **Type of conversion** parameter to `LPC to autocorrelation` or `RC to autocorrelation` to select the domain from which you want to convert your coefficients. The A port corresponds to LPC coefficients, and the K port corresponds to the RC coefficients.

The block input can be an N-by-M matrix or an unoriented vector. Each column of the matrix is treated as a channel. When the input is an unoriented vector, the input is treated as one channel.

Use the **Specify P** parameter to set the value of the prediction error power. You can set this parameter to 1 by selecting `Assume P=1`. When you select `Via input port`, a P port appears on the block. You can use this port to input the value of the actual, non-unity prediction error power for each channel. The length of this vector must equal the number of channels in the input.

The **If first input value is not 1** parameter specifies the behavior of the block when the first coefficient of the LPC coefficient vector is not 1. The following options are available:

- `Replace it with 1` — The block changes the first value of the coefficient vector to 1. The rest of the coefficient values are unchanged.

- `Normalize` — The block divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1.
- `Normalize and Warn` — The block divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1. The block displays a warning message telling you that your vector of coefficients has been normalized.
- `Error` — The block displays an error telling you that the first coefficient of the LPC coefficient vector is not 1.

Parameters

Type of conversion

From the list select `LPC to autocorrelation` or `RC to autocorrelation` to specify the domain from which you want to convert your coefficients.

Specify P

From the list select `Assume P=1` or `Via input port` to specify the value of prediction error power.

If first input value is not 1

Select what you would like the block to do when the first coefficient of the LPC coefficient vector is not 1. You can choose `Replace it with 1`, `Normalize`, `Normalize and Warn`, and `Error`.

References

Orfanidis, S.J. *Optimum Signal Processing*. New York, McGraw-Hill, 1988.

Makhoul, J. *Linear Prediction: A tutorial review*. Proc. IEEE. 63, 63, 56 (1975).

Markel, J.D. and A. H. Gray, Jr., *Linear Prediction of Speech*. New York, Springer-Verlag, 1976.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

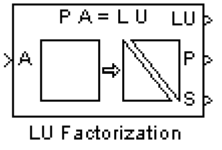
See Also

`Levinson-Durbin`	DSP System Toolbox
`LPC to LSF/LSP Conversion`	DSP System Toolbox
`LSF/LSP to LPC Conversion`	DSP System Toolbox
`LPC to/from RC`	DSP System Toolbox

Introduced before R2006a

LU Factorization

Factor square matrix into lower and upper triangular components



Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

`dspfactors`

Description

The LU Factorization block factors a row-permuted version of the square input matrix A as $A_p = L*U$, where L is a unit-lower triangular matrix, U is an upper triangular matrix, and A_p contains the rows of A permuted as indicated by the permutation index vector P. The block uses the pivot matrix A_p instead of the exact input matrix A because it improves the numerical accuracy of the factorization. You can determine the singularity of the input matrix A by enabling the optional output port S. When A is singular, the block outputs a `1` at port S; when A is nonsingular, it outputs a `0`.

To improve efficiency, the output of the LU Factorization block at port LU is a composite matrix containing both the lower triangle elements of L and the upper triangle elements of U. Thus, the output is in a different format than the output of the MATLAB `lu` function, which returns L and U as separate matrices. To convert the output from the block's `LU` port to separate L and U matrices, use the following code:

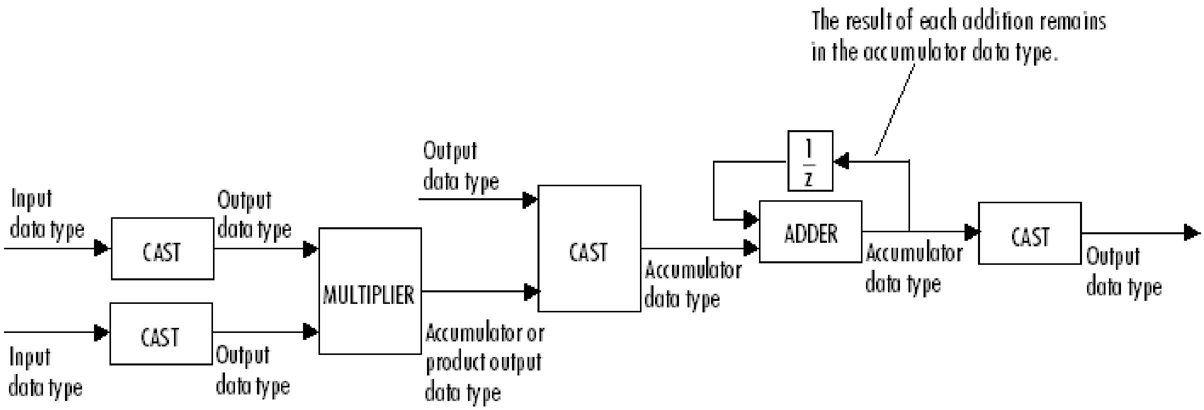
```
L = tril(LU,-1)+eye(size(LU));
U = triu(LU);
```

If you compare the results produced by these equations to the actual output of the MATLAB `lu` function, you may see slightly different values. These differences are due to rounding error, and are expected.

See the `lu` function reference page in the MATLAB documentation for more information about LU factorizations.

Fixed-Point Data Types

The following diagram shows the data types used within the LU Factorization block for fixed-point signals.



You can set the product output, accumulator, and output data types in the block dialog as discussed below.

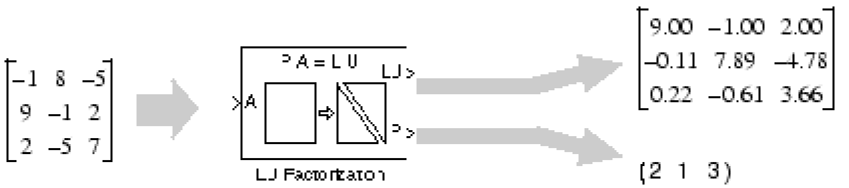
The output of the multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”.

Examples

The row-pivoted matrix A_p and permutation index vector P computed by the block are shown below for 3-by-3 input matrix A.

$$A = \begin{bmatrix} -1 & 8 & -5 \\ 9 & -1 & 2 \\ 2 & -5 & 7 \end{bmatrix} \qquad P = \begin{pmatrix} 2 & 1 & 3 \end{pmatrix} \qquad A_P = \begin{bmatrix} 9 & -1 & 2 \\ -1 & 8 & -5 \\ 2 & -5 & 7 \end{bmatrix}$$

The LU output is a composite matrix whose lower subtriangle forms *L* and whose upper triangle forms *U*.

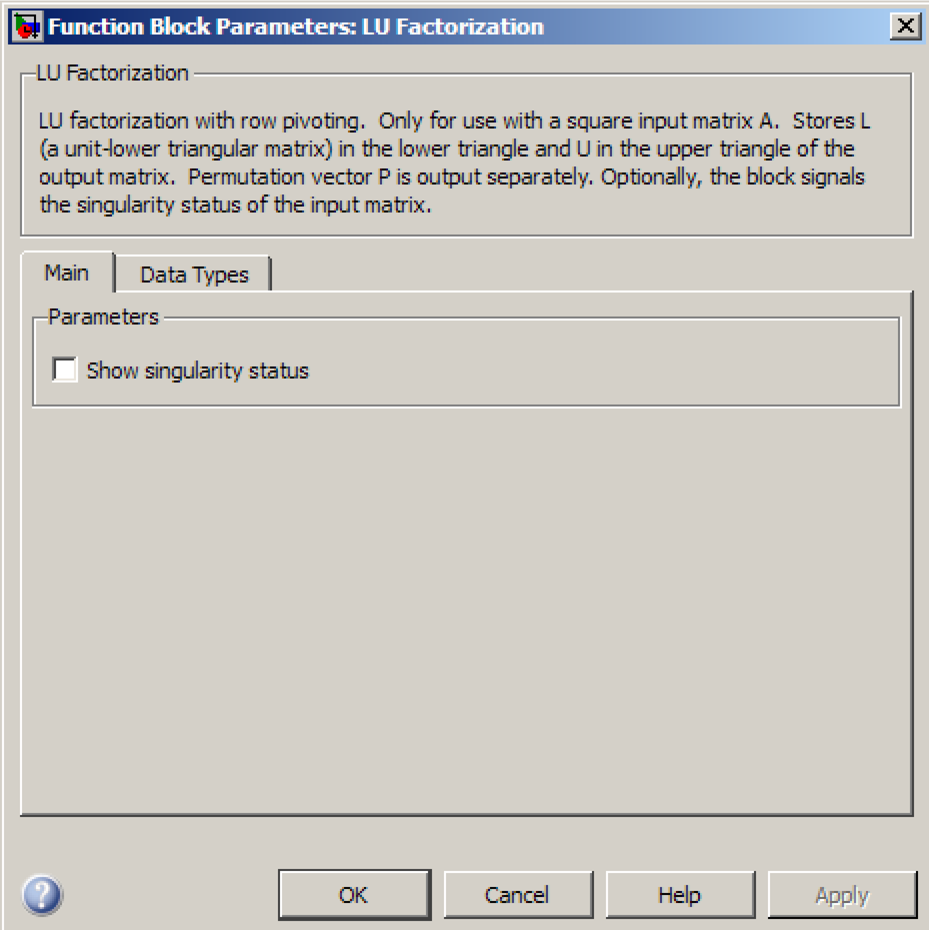


$$L = \begin{bmatrix} 1 & 0 & 0 \\ -0.11 & 1 & 0 \\ 0.22 & -0.61 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 9.00 & -1.00 & 2.00 \\ 0 & 7.89 & -4.78 \\ 0 & 0 & 3.66 \end{bmatrix}$$

See “Matrix Factorizations” in the *DSP System Toolbox User's Guide* for another example using the LU Factorization block.

Dialog Box

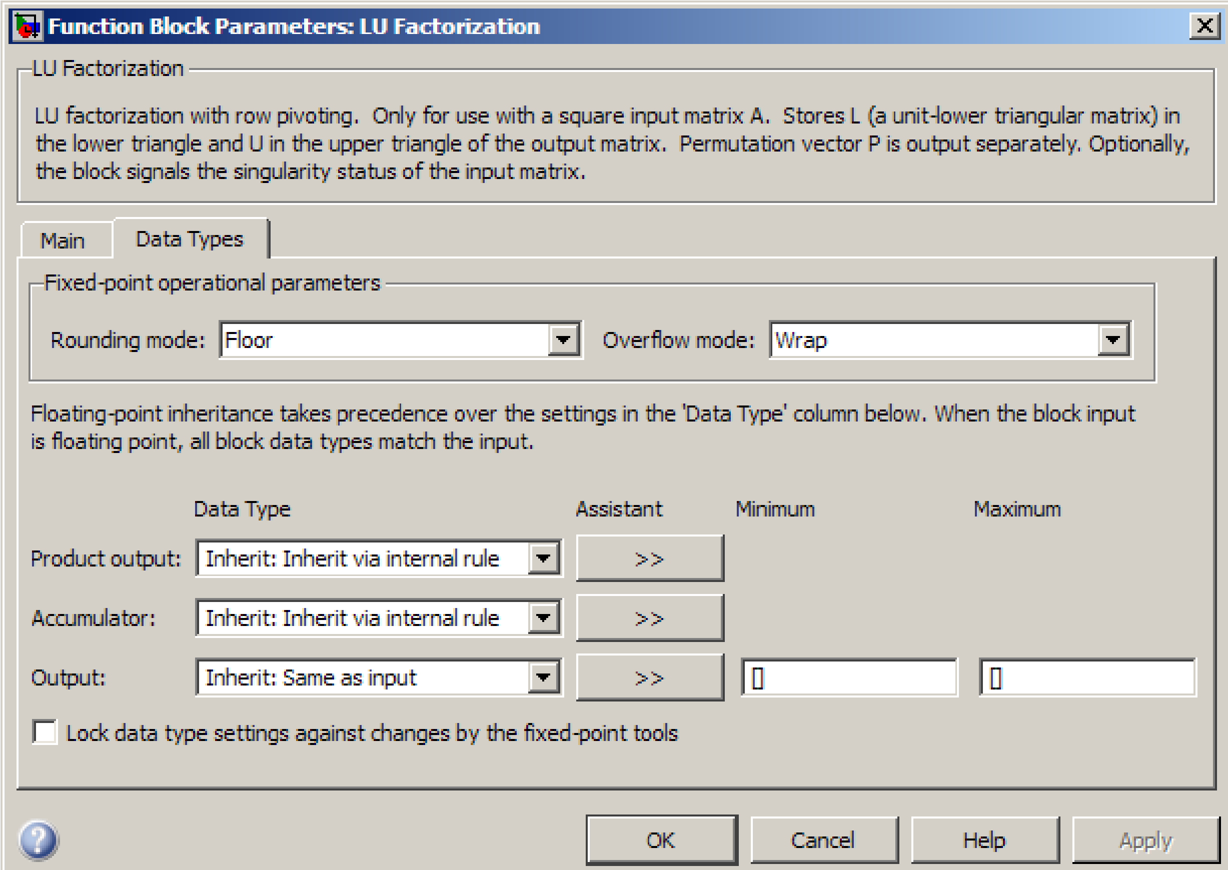
The **Main** pane of the LU Factorization block dialog appears as follows.



Show singularity status

Select to output the singularity of the input at port S, which outputs Boolean data type values of 1 or 0. An output of 1 indicates that the current input is singular, and an output of 0 indicates the current input is nonsingular.

The **Data Types** pane of the LU Factorization block dialog appears as follows.



Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-1124 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-1124 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-1124 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button  to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
LU	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
P	• Double-precision floating point • Single-precision floating point • 32-bit unsigned integers
S	• Boolean

See Also

`Autocorrelation LPC`	DSP System Toolbox
`Cholesky Factorization`	DSP System Toolbox
`LDL Factorization`	DSP System Toolbox

`LU Inverse`	DSP System Toolbox
`LU Solver`	DSP System Toolbox
`Permute Matrix`	DSP System Toolbox
`QR Factorization`	DSP System Toolbox
`lu`	MATLAB

See “Matrix Factorizations” for related information.

Introduced before R2006a

LU Inverse

Compute inverse of square matrix using LU factorization

Library

Math Functions / Matrices and Linear Algebra / Matrix Inverses

`dspinverses`

Description

The LU Inverse block computes the inverse of the square input matrix A by factoring and inverting row-pivoted variant A_p.

$$A_p^{-1} = (LU)^{-1}$$

L is a lower triangular square matrix with unity diagonal elements, and U is an upper triangular square matrix. The block outputs the inverse matrix A^{-1}.

Examples

See “Matrix Inverses” for an example that uses the LU Inverse block.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

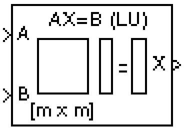
`Cholesky Inverse`	DSP System Toolbox
`LDL Inverse`	DSP System Toolbox
`LU Factorization`	DSP System Toolbox
`LU Solver`	DSP System Toolbox
`inv`	MATLAB

See “Matrix Inverses” for related information.

Introduced before R2006a

LU Solver

Solve AX=B for X when A is square matrix



Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

`dspsolvers`

Description

The LU Solver block solves the linear system AX=B by applying LU factorization to the M-by-M matrix at the A port. The input to the B port is the right side M-by-N matrix, B. The M-by-N matrix output X is the unique solution of the equations.

The block treats length-M unoriented vector input to the input port B as an M-by-1 matrix.

Algorithm

The LU algorithm factors a row-permuted variant (A_p) of the square input matrix A as

$$A_p = LU$$

where L is a lower triangular square matrix with unity diagonal elements, and U is an upper triangular square matrix.

The matrix factors are substituted for A_p in

$$A_pX = B_p$$

where B_p is the row-permuted variant of B, and the resulting equation

$$LUX = B_p$$

is solved for X by making the substitution $Y = UX$, and solving two triangular systems.

$$LY = B_p$$
$$UX = Y$$

Examples

See “Linear System Solvers” for an example that uses the LU Solver block.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

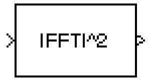
`Autocorrelation LPC`	DSP System Toolbox
`Cholesky Solver`	DSP System Toolbox
`LDL Solver`	DSP System Toolbox
`Levinson-Durbin`	DSP System Toolbox
`LU Factorization`	DSP System Toolbox
`LU Inverse`	DSP System Toolbox
`QR Solver`	DSP System Toolbox

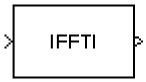
See “Linear System Solvers” for related information.

Introduced before R2006a

Magnitude FFT

Compute nonparametric estimate of spectrum using periodogram method





Library

- Estimation / Power Spectrum Estimation

  `dspspect3`
- Transforms

  `dspxfrm3`

Description

The Magnitude FFT block computes a nonparametric estimate of the spectrum using the periodogram method.

When the **Output** parameter is set to `Magnitude squared`, the block output for an M-by-N input u is equivalent to

```
y = abs(fft(u,nfft)).^2       % M ≤ nfft
```

When the **Output** parameter is set to `Magnitude`, the block output for an input u is equivalent to

```
y = abs(fft(u,nfft))          % M ≤ nfft
```

When $M > N_{fft}$, the block wraps the input to N_{fft} before computing the FFT using one of the above equations:

```
y(:,k)=datawrap(u(:,k),nfft)     % 1 ≤ k ≤ N
```

When $M > N_{fft}$, the block can also truncate the input:

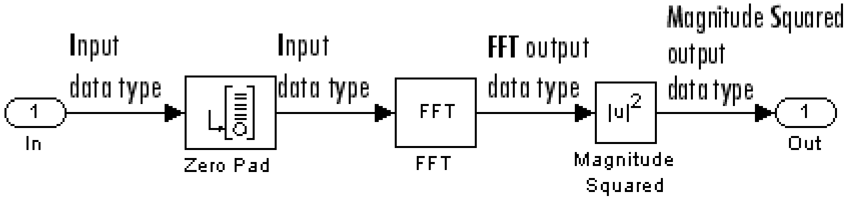
```
y(:,k)=abs(fft(u,nfft))        % 1 ≤ k ≤ N
```

The block treats an M-by-N matrix input as M sequential time samples from N independent channels. The block computes a separate estimate for each of the N independent channels and generates an N_{fft}-by-N matrix output. Each column of the output matrix contains the estimate of the corresponding input column's power spectral density at N_{fft} equally spaced frequency points in the range $[0,F_s)$, where F_s represents the signal's sample frequency. The block always outputs sample–based data.

The Magnitude FFT block supports real and complex floating-point inputs. The block also supports real fixed-point inputs in both `Magnitude` and `Magnitude squared` modes, and complex fixed-point inputs in the `Magnitude squared` mode.

Fixed-Point Data Types

The following diagram shows the data types used within the Magnitude FFT subsystem block for fixed-point signals.



The settings for the fixed-point parameters of the FFT block in the diagram above are as follows:

- Sine table — `Same word length as input`
- Integer rounding mode — `Floor`
- Saturate on integer overflow — unchecked
- Product output — `Inherit via internal rule`
- Accumulator — `Inherit via internal rule`
- Output — `Inherit via internal rule`

The settings for the fixed-point parameters of the Magnitude Squared block in the diagram above are as follows:

- Integer rounding mode — `Floor`
- Saturate on integer overflow — checked
- Output — `Inherit via internal rule`

Examples

The dspsacomp example compares the periodogram method with several other spectral estimation methods.

Parameters

Output

Specify whether the block computes the magnitude FFT or magnitude-squared FFT of the input.

FFT implementation

Set this parameter to `FFTW` to support an arbitrary length input signal. The block restricts generated code with FFTW implementation to MATLAB host computers.

Set this parameter to `Radix-2` for bit-reversed processing, fixed or floating-point data, or for portable C-code generation using the Simulink Coder. The first dimension M, of the input matrix must be a power of two. To work with other input sizes, use the `Pad` block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW algorithm.

Set this parameter to `Auto` to let the block choose the FFT implementation. For non-power-of-two transform lengths, the block restricts generated code to MATLAB host computers.

Inherit FFT length from input dimensions

Select to use the input frame size as the number of data points, on which to perform the FFT. When you select this check box, this number must be a power of two. When you do not select this check box, the **FFT length** parameter specifies the number of data points.

FFT length

Enter the number of data points on which to perform the FFT, N_{fft}. When N_{fft} is larger than the input frame size, each frame is zero-padded as needed. When N_{fft} is smaller than the input frame size, each frame is wrapped as needed. This parameter is enabled when you clear the **Inherit FFT length from input dimensions** check box.

When you set the **FFT implementation** parameter to `Radix-2`, this value must be a power of two.

Wrap input data when FFT length is shorter than input length

Choose to wrap or truncate the input, depending on the **FFT length**. If this parameter is checked, modulo-length data wrapping occurs before the FFT operation, given **FFT length** is shorter than the input length. If this property is unchecked, truncation of the input data to the FFT length occurs before the FFT operation. The default is checked.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

References

[1] FFTW (http://www.fftw.org)

[2] Frigo, M. and S. G. Johnson, “FFTW: An Adaptive Software Architecture for the FFT,”*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

[3] Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

[4] Orfanidis, S. J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[5] Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- When the following conditions apply, the executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB:

  - **FFT implementation** is set to `FFTW`.
  - **Inherit FFT length from input dimensions** is cleared, and **FFT length** is set to a value that is not a power of two.

  Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- When the FFT length is a power of two, you can generate standalone C and C++ code from this block.

See Also

See Also

Functions

`pwelch`

Blocks
`Burg Method` | `Short-Time FFT` | `Spectrum Analyzer` | `Yule-Walker Method`

Topics
“Spectral Analysis”

Introduced before R2006a

Matrix 1-Norm

Compute 1-norm of matrix

Matrix
1-Norm

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

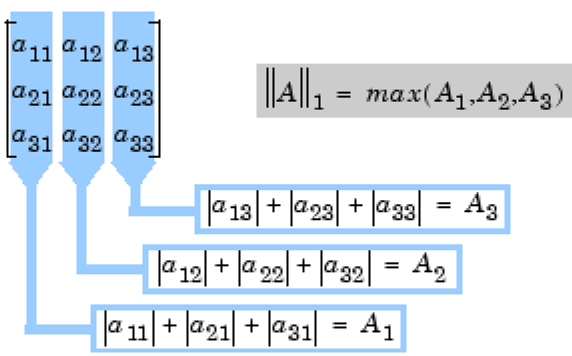
Description

The Matrix 1-Norm block computes the 1-norm, or maximum column-sum, of an M-by-N input matrix, A.

$$y = \|A\|_1 = \max_{1 \le j \le N} \sum_{i=1}^{M} |a_{ij}|$$

This is equivalent to

```
y = max(sum(abs(A)))      % Equivalent
MATLAB code
```

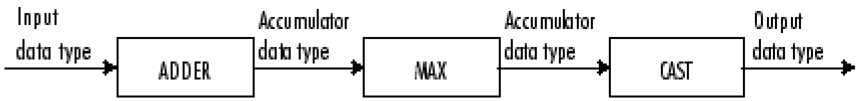


The block treats length-M unoriented vector input as an M-by-1 matrix. The output, y, is always a scalar.

The Matrix 1-Norm block supports real and complex floating-point inputs, and real fixed-point inputs.

Fixed-Point Data Types

The following diagram shows the data types used within the Matrix 1-Norm block for fixed-point signals.

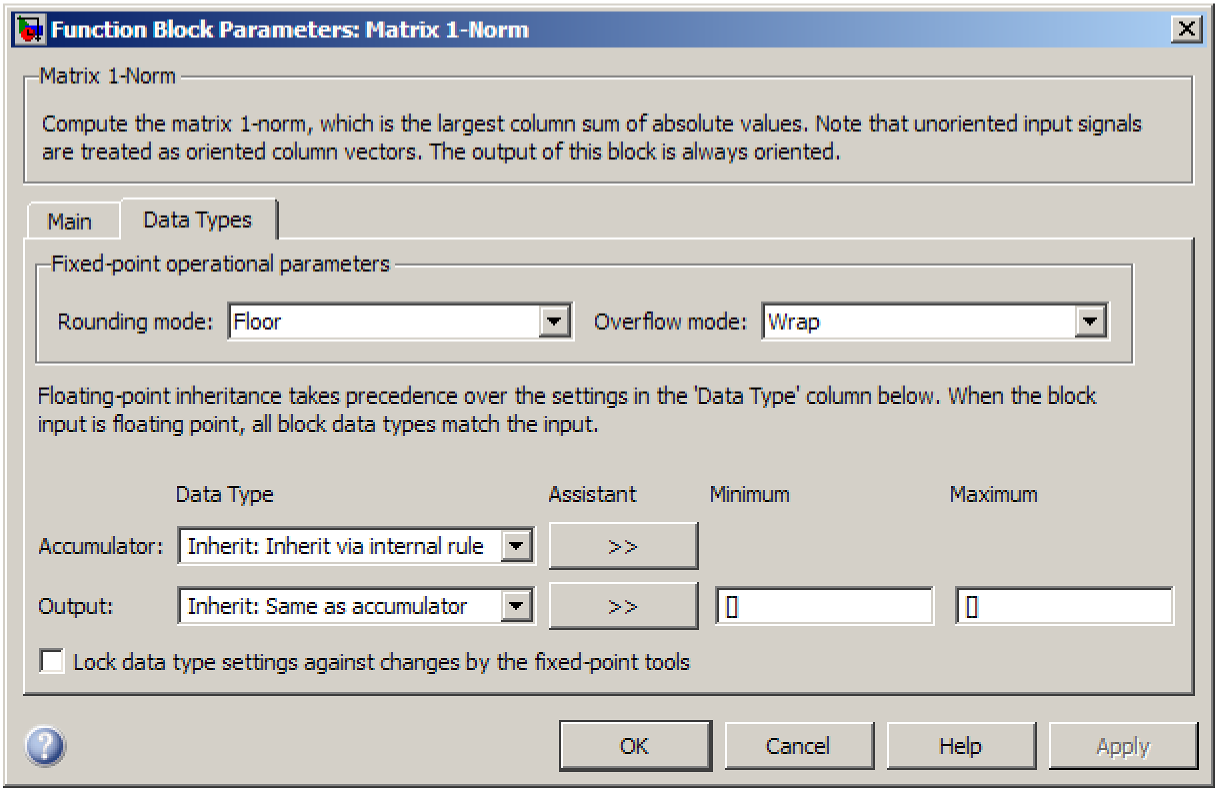


The block calculations are all done in the accumulator data type until the `max` is performed. The result is then cast to the output data type. You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 2-1142 below.

Dialog Box

There are no parameters on the **Main** pane of this dialog.

The **Data types** pane of the Matrix 1-Norm block dialog appears as follows.



Note: Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-1142 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-1142 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

Normalization	DSP System Toolbox
Reciprocal Condition	DSP System Toolbox
`norm`	MATLAB

Introduced before R2006a

Matrix Concatenate

Concatenate input signals of same data type to create contiguous output signal

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

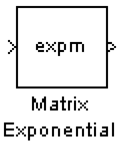
Description

The Matrix Concatenate block is an implementation of the Simulink Matrix Concatenate block. See Matrix Concatenate for more information.

Introduced in R2008a

Matrix Exponential

Compute matrix exponential



Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

Description

The Matrix Exponential block computes the matrix exponential using a scaling and squaring algorithm with a Pade approximation. The input matrix must be square.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

`Array-Vector Multiply`	DSP System Toolbox
`expm`	MATLAB
Dot Product	Simulink
`Matrix Product`	DSP System Toolbox
Product	Simulink

Introduced before R2006a

Matrix Multiply

Multiply or divide inputs

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

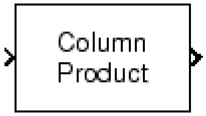
Description

The Matrix Multiply block is an implementation of the Simulink Product block. See Product for more information.

Introduced before R2006a

Matrix Product

Multiply matrix elements along rows, columns, or entire input



Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

Description

The Matrix Product block multiplies the elements of an M-by-N input matrix u along its rows, its columns, or over all its elements.

When the **Multiply over** parameter is set to `Rows`, the block multiplies across the elements of each row and outputs the resulting M-by-1 matrix. The block treats length-N unoriented vector input as a 1-by-N matrix.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} \left(\prod_{j=1}^{3} u_{1j} \right) \\ \left(\prod_{j=1}^{3} u_{2j} \right) \\ \left(\prod_{j=1}^{3} u_{3j} \right) \end{bmatrix}$$

When the **Multiply over** parameter is set to `Columns`, the block multiplies down the elements of each column and outputs the resulting 1-by-N matrix. The block treats length-M unoriented vector input as an M-by-1 matrix.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}$$

$$\Downarrow$$

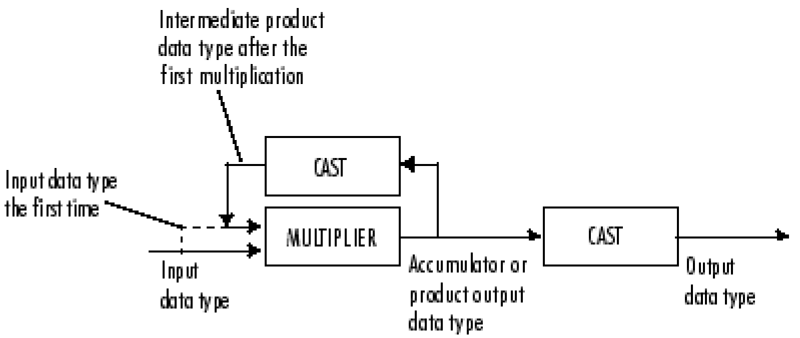
$$\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = \left[\left(\prod_{i=1}^{3} u_{i1} \right) \left(\prod_{i=1}^{3} u_{i2} \right) \left(\prod_{i=1}^{3} u_{i3} \right) \right]$$

When the **Multiply over** parameter is set to `Entire input`, the block multiplies all the elements of the input together and outputs the resulting scalar.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \Rightarrow y = \left(\prod_{i=1}^{3} \prod_{j=1}^{3} u_{ij} \right)$$

Fixed-Point Data Types

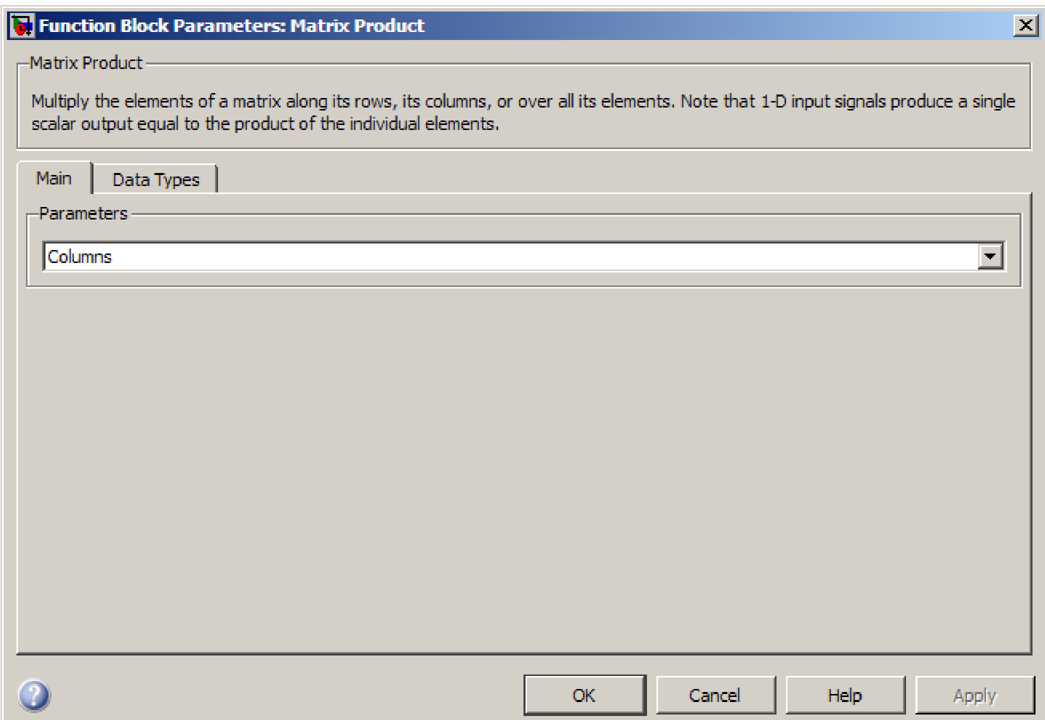
The following diagram shows the data types used within the Matrix Product block for fixed-point signals.



The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see “Multiplication Data Types”. You can set the accumulator, product output, intermediate product, and output data types in the block dialog as discussed in "Dialog Box" on page 2-1152 below.

Dialog Box

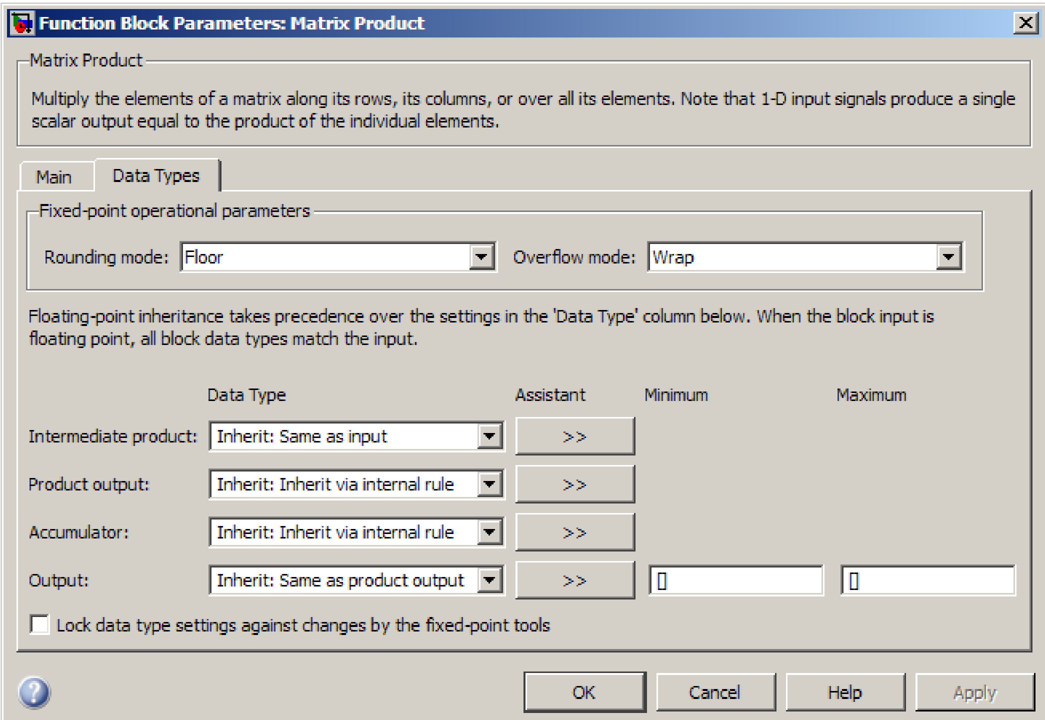
The **Main** pane of the Matrix Product block dialog appears as follows.



Multiply over

Indicate whether to multiply together the elements of each row, each column, or the entire input.

The **Data Types** pane of the Matrix Product block dialog appears as follows.



Note: Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Intermediate product

Specify the intermediate product data type. As shown in “Fixed-Point Data Types” on page 2-1151, the output of the multiplier is cast to the intermediate product data type before the next element of the input is multiplied into it. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-1151 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-1151 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-1151 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
	• Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

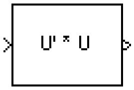
See Also

`Array-Vector Multiply`	DSP System Toolbox
`Matrix Square`	DSP System Toolbox
`Matrix Sum`	DSP System Toolbox
`prod`	MATLAB

Introduced before R2006a

Matrix Square

Compute square of input matrix



Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

Description

The Matrix Square block computes the square of an M-by-N input matrix, u, by premultiplying with the Hermitian transpose.

```
y = u' * u     % Equivalent MATLAB code
```

The block treats length-M unoriented vector inputs as an M-by-1 matrix. When the input is an M-by-N matrix, the output of the block is an N-by-N matrix.

Applications

The Matrix Square block is useful in a variety of applications:

- General matrix squares — The Matrix Square block computes the output matrix, y, without explicitly forming u'. It is therefore more efficient than other methods for computing the matrix square.
- Sum of squares — When the input is a column vector (N=1), the block's operation is equivalent to a multiply-accumulate (MAC) process, or inner product. The output is the sum of the squares of the input, and is always a real scalar.
- Correlation matrix — When the input is a row vector (M=1), the output, y, is the symmetric autocorrelation matrix, or outer product.

Parameters

Output minimum

Specify the minimum value the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Lock output data type setting against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the **Output data type** you specify on the block mask.

Integer rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select this check box to have overflows saturate to the maximum or minimum value that the data type can represent. If you clear this check box, the block wraps all overflows. See overflow mode for more information.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

`Matrix Multiply`	DSP System Toolbox
`Matrix Product`	DSP System Toolbox
`Matrix Sum`	DSP System Toolbox

Introduced before R2006a

Matrix Sum

Sum matrix elements along rows, columns, or entire input

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

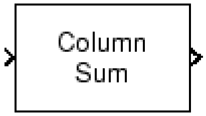
Description

The Matrix Sum block is an implementation of the Simulink Sum block. See Sum for more information.

Introduced before R2006a

Matrix Sum (Obsolete)

Sum matrix elements along rows, columns, or entire input



Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspobslib`

Description

The Matrix Sum block sums the elements of an M-by-N input matrix u along its rows, its columns, or over all its elements.

When the **Sum over** parameter is set to `Rows`, the block sums across the elements of each row and outputs the resulting M-by-1 matrix. A length-N 1-D vector input is treated as a 1-by-N matrix.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} \left(\sum_{j=1}^{3} u_{1j} \right) \\ \left(\sum_{j=1}^{3} u_{2j} \right) \\ \left(\sum_{j=1}^{3} u_{3j} \right) \end{bmatrix}$$

When the **Sum over** parameter is set to `Columns`, the block sums down the elements of each column and outputs the resulting 1-by-N matrix. A length-M 1-D vector input is treated as a M-by-1 matrix.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = \left[\left(\sum_{i=1}^{3} u_{i1} \right) \left(\sum_{i=1}^{3} u_{i2} \right) \left(\sum_{i=1}^{3} u_{i3} \right) \right]$$

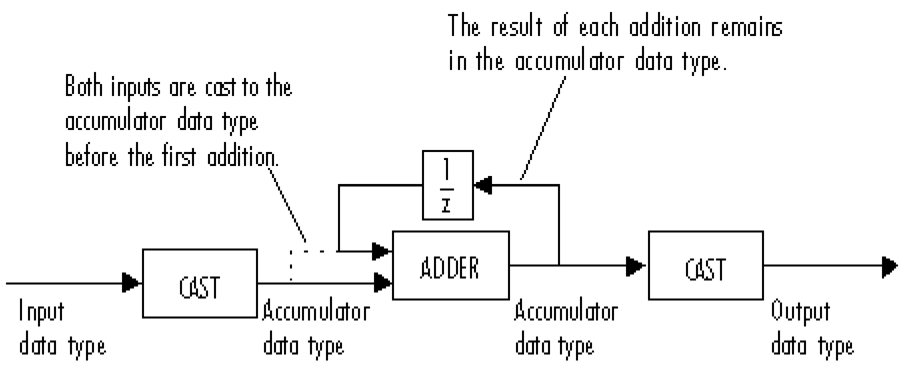
When the **Sum over** parameter is set to `Entire input`, the block sums all the elements of the input together and outputs the resulting scalar.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \Rightarrow y = \left(\sum_{i=1}^{3} \sum_{j=1}^{3} u_{ij} \right)$$

The output of the Matrix Sum block has the same frame status as the input. This block accepts real and complex fixed-point and floating-point inputs except for complex unsigned fixed-point inputs.

Fixed-Point Data Types

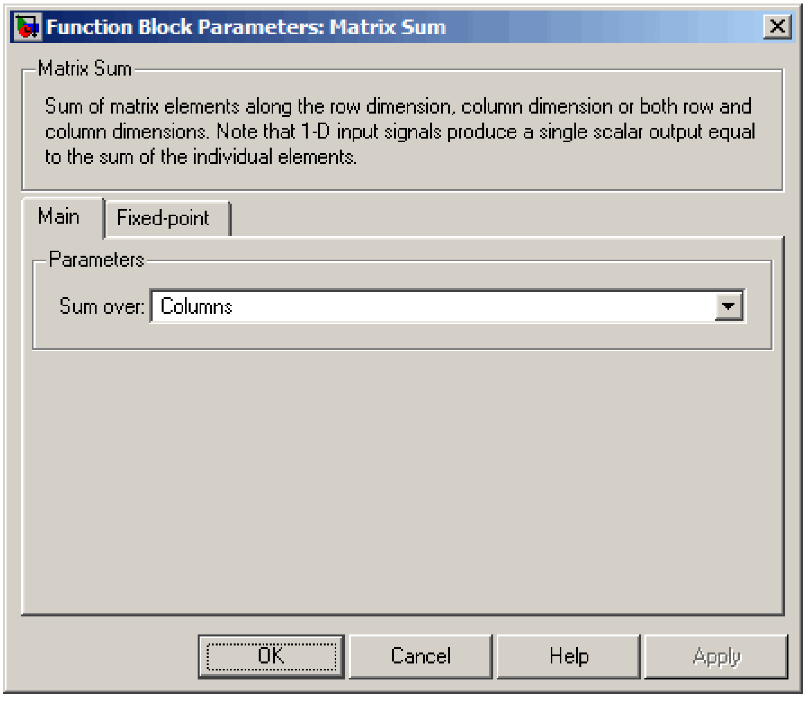
The following diagram shows the data types used within the Matrix Sum block for fixed-point signals.



You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 2-1163 below.

Dialog Box

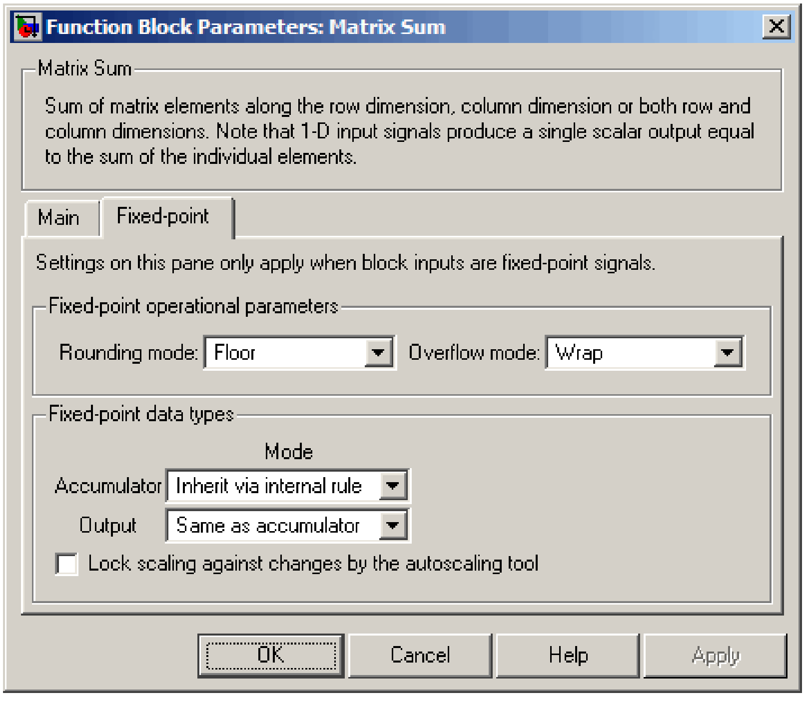
The **Main** pane of the Matrix Sum block dialog appears as follows.



Sum over

Indicate whether to sum the elements of each row, each column, or of the entire input.

The **Fixed-point** pane of the Matrix Sum block dialog appears as follows.



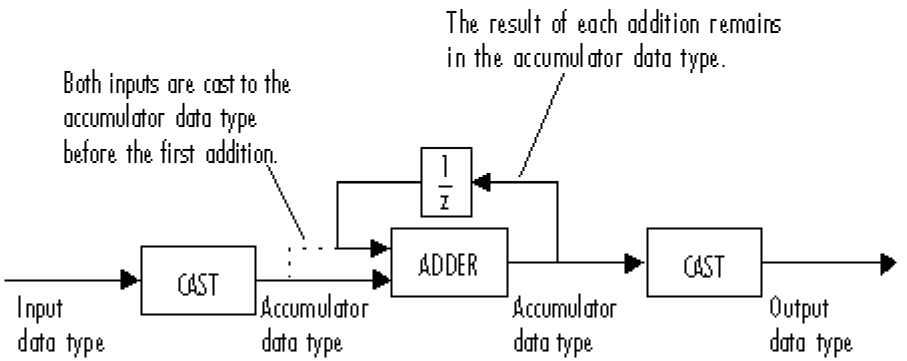
Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Accumulator



As depicted above, the elements of the block input are cast to the accumulator data type before they are added together. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how you would like to designate this accumulator word and fraction lengths:

- When you select `Inherit via internal rule`, the accumulator word length and fraction length are calculated automatically. For information about how the accumulator word and fraction lengths are calculated when an internal rule is used, see "Inherit via Internal Rule".
- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you specify the output word length and fraction length:

- When you select `Same as accumulator`, these characteristics match those of the accumulator.
- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool

Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overridden by the autoscaling feature of the Fixed-Point Tool. See the `fxptdlg` reference page for more information.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

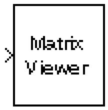
See Also

`Matrix Product`	DSP System Toolbox
`Matrix Multiply`	DSP System Toolbox
`sum`	MATLAB

Introduced in R2008b

Matrix Viewer

Display matrices as color images



Library

Sinks

`dspsnks4`

Description

The Matrix Viewer block displays an *M*-by-*N* matrix input by mapping the matrix element values to a specified range of colors. The display is updated as each new input is received. This block treats an unoriented length *M* vector input as an *M*-by-1 matrix.

You can use the Matrix Viewer block in models running in Normal or Accelerator simulation modes. The software does not support this block in models running in Rapid Accelerator or External mode. For more information about these modes, see “How Acceleration Modes Work” (Simulink) in the *Simulink User's Guide*.

Image Properties

Select the **Image Properties** tab to show the image property parameters, which control the colormap and display.

You specify the mapping of matrix element values to colors in the **Colormap matrix**, **Minimum input value**, and **Maximum input value** parameters. For a colormap with *L* colors, the colormap matrix has dimension *L*-by-3, with one row for each color and one column for each element of the RGB triple that defines the color. Examples of RGB triples are

```
[ 1   0   0 ]   (red)
[ 0   0   1 ]   (blue)
[0.8 0.8 0.8]   (light gray)
```

See the `ColorSpec` property in the MATLAB documentation for complete information about defining RGB triples.

MATLAB provides a number of functions for generating predefined colormaps, such as `hot`, `cool`, `bone`, and `autumn`. Each of these functions accepts the colormap size as an argument, and can be used in the **Colormap matrix** parameter. For example, when you specify `gray(128)` for the **Colormap matrix** parameter, the matrix is displayed in 128 shades of gray. The color in the first row of the colormap matrix represents the value specified by the **Minimum input value** parameter, and the color in the last row represents the value specified by the **Maximum input value** parameter. Values between the minimum and maximum are quantized and mapped to the intermediate rows of the colormap matrix.

The documentation for the MATLAB `colormap` function provides complete information about specifying colormap matrices, and includes a complete list of the available colormap functions.

Axis Properties

Select the **Axis Properties** tab to show the axis property parameters, which control labeling and positioning.

The **Axis origin** parameter determines where the first element of the input matrix, $U(1,1)$, is displayed. When you specify `Upper left corner`, the matrix is displayed in matrix orientation, with $U(1,1)$ in the upper-left corner.

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ U_{21} & U_{22} & U_{23} & U_{24} \\ U_{31} & U_{32} & U_{33} & U_{34} \\ U_{41} & U_{42} & U_{43} & U_{44} \end{bmatrix}$$

When you specify `Lower left corner`, the matrix is flipped vertically to image orientation, with $U(1,1)$ in the lower-left corner.

$$\begin{bmatrix} U_{41} & U_{42} & U_{43} & U_{44} \\ U_{31} & U_{32} & U_{33} & U_{34} \\ U_{21} & U_{22} & U_{23} & U_{24} \\ U_{11} & U_{12} & U_{13} & U_{14} \end{bmatrix}$$

Axis zoom, when selected, causes the image display to completely fill the figure window. Axis titles are not displayed. This option can also be selected from the pop-up menu that is displayed when you right-click in the figure window. When **Axis zoom** is cleared, the axis labels and titles are displayed in a gray border surrounding the image axes.

To customize the axis tick range, set **Axis tick mode** to `User-defined` and specify a two-element row vector, [`Minvalue Maxvalue`], in **X-tick range** and **Y-tick range**. The matrix data does not change even when the axes ticks change. For example, consider displaying a matrix on axes with ticks ranging from 0 to 10. When you set **X-tick range** and **Y-tick range** to [0 1], the range of ticks changes from [0 10] to [0 1].

Figure Window

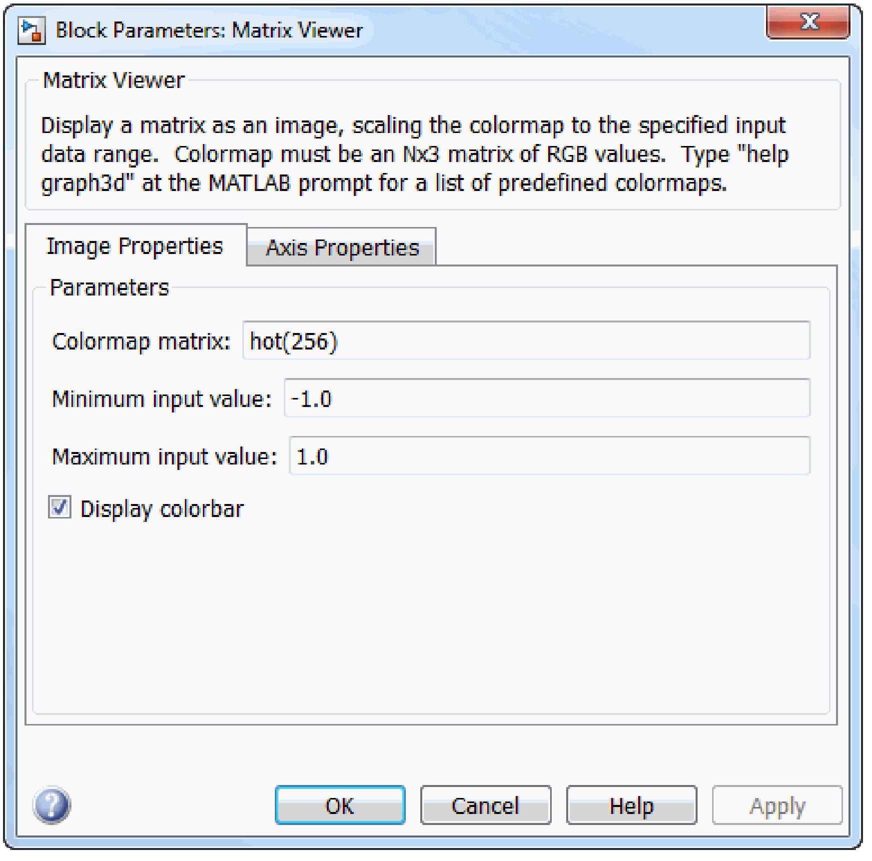
The image title in the figure title bar is the same as the block title. The axis tick marks reflect the size of the input matrix; the x-axis is numbered from 1 to N (number of columns), and the y-axis is numbered from 1 to M (number of rows).

Right-click the image in the figure window to access the following menu items:

- **Refresh** erases all data on the scope display except for the most recent image.
- **Autoscale** recomputes the minimum and maximum input values to fit the range of values observed in a series of 10 consecutive inputs. The numerical limits selected by the autoscale feature are shown in the **Minimum input value** and **Maximum input value** parameters, where you can make further adjustments to them manually.
- **Axis zoom**, when selected, causes the image to completely fill the figure window. Axis titles are not displayed. When **Axis zoom** is cleared, the axis labels and titles are displayed in a gray border surrounding the scope axes. This option can also be set in the Axis Properties pane of the parameter dialog box.
- **Colorbar**, when selected, displays a bar with the specified colormap to the right of the image axes.
- **Save Position** automatically updates the **Figure position** parameter in the **Axis Properties** pane to reflect the figure window's current position and size on the screen. To make the scope window open at a particular location on the screen when the simulation runs, drag the window to the desired location, resize it, and select **Save Position**. The parameter dialog box must be closed when you select **Save Position** for the **Figure position** parameter to be updated.

Dialog Box

The **Image Properties** pane of the Matrix Viewer block appears as follows.



Colormap matrix

A 3-column matrix defining the colormap as a set of RGB triples, or a call to a colormap-generating function such as `hot` or `spring`. See the `ColorSpec` property for complete information about defining RGB triples, and the MATLAB `colormap` function for a list of colormap-generating functions. Tunable (Simulink).

Minimum input value

The input value to be mapped to the color defined in the first row of the colormap matrix. Right-click in the figure window and select `Autoscale` from pop-up menu to set this parameter to the minimum value observed in a series of 10 consecutive matrix inputs. Tunable (Simulink).

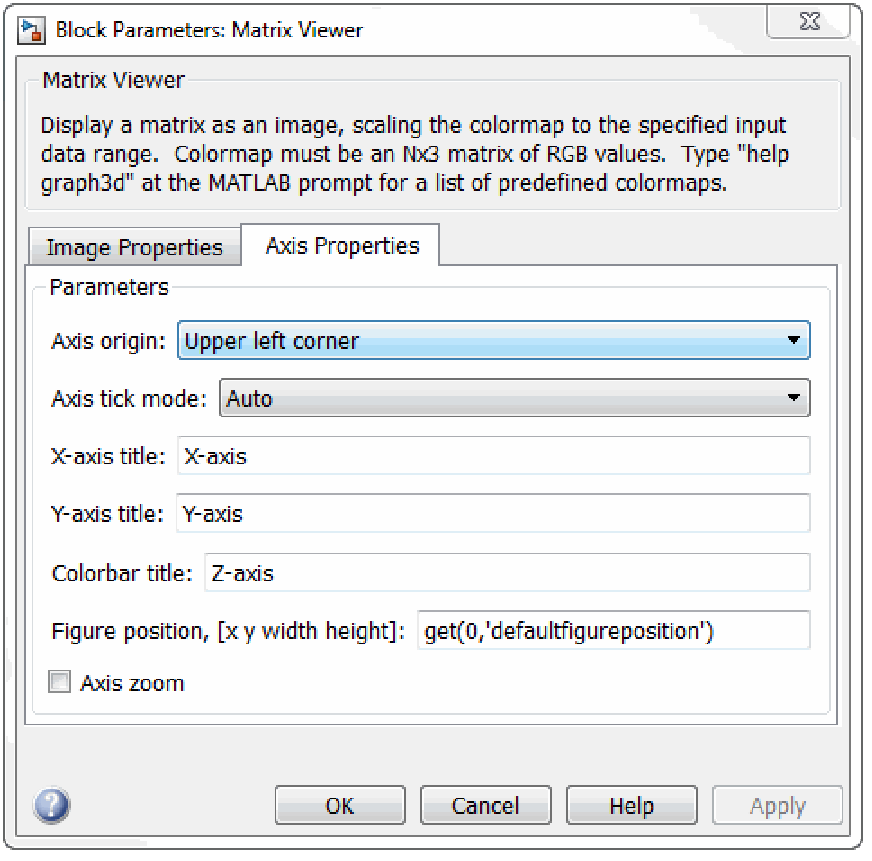
Maximum input value

The input value to be mapped to the color defined in the last row of the colormap matrix. Right-click in the figure window and select `Autoscale` from the pop-up menu to set this parameter to the maximum value observed in a series of 10 consecutive matrix inputs. Tunable (Simulink).

Display colorbar

Select to display a bar with the selected colormap to the right of the image axes. Tunable (Simulink).

The **Axis Properties** pane of the Matrix Viewer block appears as follows.



Axis origin

The position within the axes where the first element of the input matrix, $U(1,1)$, is plotted; bottom left or top left. Tunable (Simulink).

Axis tick mode

The mode used to specify the axis tick range. When you select `Auto`, the *x*-axis tick range and *y*-axis tick range are chosen based on the input signal. When you select `User-defined`, the axis tick range is set based on the values you specify in **X-tick range** and **Y-tick range**. The default is `Auto`.

X-tick range

x-axis tick range, specified as a two-element row vector of real scalars, [X_{min} X_{max}]. This parameter applies only when you set **Axis tick mode** to `User-defined`. When you change **X-tick range** during simulation, the axis range on the Matrix Viewer block updates in real time. The default is `[0 1]`.

Y-tick range

y-axis tick range, specified as a two-element row vector of real scalars, [Y_{min} Y_{max}]. This parameter applies only when you set **Axis tick mode** to `User-defined`. When you change **Y-tick range** during simulation, the axis range on the Matrix Viewer block updates in real time. The default is `[0 1]`.

X-axis title

The text to be displayed below the *x*-axis. Tunable (Simulink).

Y-axis title

The text to be displayed to the left of the *y*-axis. Tunable (Simulink).

Colorbar title

The text to be displayed to the right of the color bar, when **Display colorbar** is currently selected. Tunable (Simulink).

Figure position, [x y width height]

A 4-element vector of the form `[x y width height]` specifying the position of the figure window, where `(0,0)` is the lower-left corner of the display. Tunable (Simulink).

Axis zoom

Resizes the image to fill the figure window. Tunable (Simulink).

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

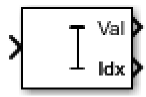
See Also

`Spectrum Analyzer`	DSP System Toolbox
`Vector Scope`	DSP System Toolbox
`colormap`	MATLAB
`ColorSpec`	MATLAB
`image`	MATLAB

Introduced before R2006a

Maximum

Maximum values of input or sequence of inputs
Library: DSP System Toolbox / Statistics



Description

The Maximum block identifies the value and position of the largest element in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the maximum value of the entire input. The Maximum block can also track the maximum values in a sequence of inputs over a period of time. The **Mode** parameter specifies the block's mode of operation and can be set to one of the following:

- `Value` — The block outputs the maximum values in the specified dimension.
- `Index` — The block outputs the index array of the maximum values in the specified dimension.
- `Value and Index` — The block outputs the maximum values and the corresponding index array in the specified dimension.
- `Running` — The block tracks the maximum values in a sequence of inputs over a period of time.

You can specify the dimension using the **Find the maximum value over** parameter.

Note: The **Running** mode in the Maximum block will be removed in a future release. To compute the running maximum in Simulink, use the `Moving Maximum` block instead.

Ports

Input

`In` — Data input
vector | matrix | *N*-D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input can be floating-point, fixed-point, or Boolean. Real fixed-point inputs can be either signed or unsigned. Complex fixed-point inputs must be signed.

This port is unnamed until you set the **Mode** parameter to `Running` and the **Reset port** parameter to any option other than `None`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed_point`
Complex Number Support: Yes

`Rst` — Reset port
scalar

Specify the reset event that causes the block to reset the running maximum. The sample time of the **Rst** input must be a positive integer multiple of the input sample time.

Dependencies

To enable this port, set the **Mode** parameter to `Running` and the **Reset port** parameter to any option other than `None`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Val` — Maximum values along the specified dimension
scalar | vector | matrix | *N*-D array

The data type of the maximum value matches the data type of the input.

When the **Mode** parameter is set to either `Value and Index` or `Value`, the following applies:

- The size of the dimension for which the block computes the maximum value is 1. The sizes of all other dimensions match those of the input array. For example, when the input is an M-by-N-by-P array, with the dimension set to `1`, the block outputs a 1-by-N-by-P array. When the dimension is set to `3`, the block outputs a two-dimensional M-by-N matrix.

- When the input is an M-by-N matrix, with the dimension set to 1, the block outputs a 1-by-N matrix.

If you specify the block to compute the maximum value over the entire input, the block outputs a scalar.

When the **Mode** parameter is set `Running`, the block tracks the maximum value of each channel in a time sequence of M-by-N inputs. In this mode, you must also specify the **Input processing** parameter as one of the following:

- `Elements as channels (sample based)` — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size M-by-N-by-P, the block outputs an M-by-N-by-P array. Each y_{ijk} element of the output contains the maximum value observed in element u_{ijk} for all inputs since the last reset.

  When a reset event occurs, the running maximum y_{ijk} in the current frame is reset to the element u_{ijk}.
- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel. This option does not support an N-dimensional input signal, where N > 2. For a two-dimensional input signal of size M-by-N, the block outputs an M-by-N matrix. Each element y_{ij} of the output contains the maximum value observed in the jth column of all inputs since the last reset, up to and including element u_{ij} of the current input.

  When a reset event occurs, the running maximum for each channel becomes the maximum value of all the samples in the current input frame, up to and including the current input sample.

The block resets the running maximum whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

Dependencies

To enable this port, set the **Mode** parameter to either `Value and Index` or `Value`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed_point`
Complex Number Support: Yes

`Idx` — Index of the maximum values along the specified dimension
scalar | vector | matrix | *N*-D array

When the input is `double`, the index values are also `double`. Otherwise, the index values are `uint32`.

Dependencies

To enable this port, set the **Mode** parameter to either `Value and Index` or `Index`.

Data Types: `double` | `uint32`

Parameters

Main Tab

`Mode` — Mode in which the block operates
`Value and Index` (default) | `Value` | `Index` | `Running`

When the **Mode** parameter is set to:

- `Value` — The block computes the maximum value in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the maximum value of the entire input at each sample time, and outputs the array, *y*. Each element in the output is the maximum value in the corresponding column, row, vector, or entire input. The output *y* depends on the setting of the **Find the maximum value over** parameter. Consider a three dimensional input signal of size *M*-by-*N*-by-*P*. Set **Find the maximum value over** to:

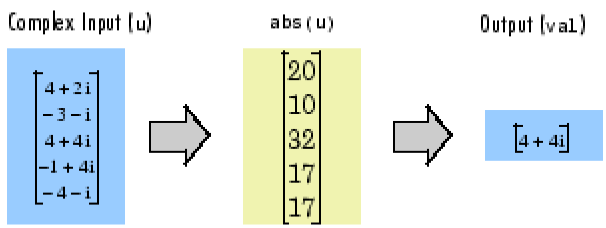
  - `Each row` — The output *y* at each sample time consists of an *M*-by-1-by-*P* array, where each element contains the maximum value of each vector over the second dimension of the input. For an *M*-by-*N* matrix input, the output at each sample time is an *M*-by-1 column vector.
  - `Each column` — The output *y* at each sample time consists of a 1-by-*N*-by-*P* array, where each element contains the maximum value of each vector over the first dimension of the input. For an *M*-by-*N* matrix input, the output at each sample time is a 1-by-*N* row vector.

In this mode, the block treats length-M unoriented vector inputs as M-by-1 column vectors.

- `Entire input` — The output y at each sample time is a scalar that contains the maximum value in the M-by-N-by-P input matrix.
- `Specified dimension` — The output y at each sample time depends on **Dimension**. If **Dimension** is set to `1`, the output is the same as when you select `Each column`. If **Dimension** is set to `2`, the output is the same as when you select `Each row`. If **Dimension** is set to `3`, the output at each sample time is an M-by-N matrix containing the maximum value of each vector over the third dimension of the input.

Complex Inputs

For complex inputs, the block selects the value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input that has the maximum magnitude squared as shown in the following figure. For complex value $u = a + bi$, the magnitude squared is $a^2 + b^2$.



- `Index` — The block computes the maximum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input, and outputs the index array I. Each element in I is an integer indexing the maximum value in the corresponding column, row, vector, or entire input. The output I depends on the setting of the **Find the maximum value over** parameter. Consider a three-dimensional input signal of size M-by-N-by-P:
  - `Each row` — The output I at each sample time consists of an M-by-1-by-P array, where each element contains the index of the maximum value of each vector over the second dimension of the input. For an input that is an M-by-N matrix, the output at each sample time is an M-by-1 column vector.
  - `Each column` — The output I at each sample time consists of a 1-by-N-by-P array, where each element contains the index of the maximum value of each vector over

the first dimension of the input. For an input that is an *M*-by-*N* matrix, the output at each sample time is a 1-by-*N* row vector.

In this mode, the block treats length-*M* unoriented vector inputs as *M*-by-1 column vectors.

- `Entire input` — The output *I* at each sample time is a 1-by-3 vector that contains the location of the maximum value in the *M*-by-*N*-by-*P* input matrix. For an input that is an *M*-by-*N* matrix, the output is a 1-by-2 vector.
- `Specified dimension` — The output *I* at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select `Each column`. If **Dimension** is set to 2, the output is the same as when you select `Each row`. If **Dimension** is set to 3, the output at each sample time is an *M*-by-*N* matrix containing the indices of the maximum values of each vector over the third dimension of the input.

When a maximum value occurs more than once, the computed index corresponds to the first occurrence. For example, when the input is the column vector `[3 2 1 2 3]'`, the computed one-based index of the maximum value is `1`, rather than `5` when `Each column` is selected.

- `Value and Index` — The block outputs the maximum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input, and the corresponding index array *I*.
- `Running` — The block tracks the maximum value of each channel in a time sequence of *M*-by-*N* inputs. In this mode, you must also specify the **Input processing** parameter as one of the following:

  - `Elements as channels (sample based)` — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size *M*-by-*N*-by-*P*, the block outputs an *M*-by-*N*-by-*P* array. Each y_{ijk} element of the output contains the maximum value observed in element u_{ijk} for all inputs since the last reset.

    When a reset event occurs, the running maximum y_{ijk} in the current frame is reset to the element u_{ijk}.
  - `Columns as channels (frame based)` — The block treats each column of the input as a separate channel. This option does not support an N-dimensional input signal, where N > 2. For a two-dimensional input signal of size *M*-by-*N*, the block outputs an *M*-by-*N* matrix. Each element y_{ij} of the output contains the

maximum value observed in the *j*th column of all inputs since the last reset, up to and including element u_{ij} of the current input.

When a reset event occurs, the running maximum for each channel becomes the maximum value of all the samples in the current input frame, up to and including the current input sample.

The block resets the running maximum whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

Running Mode for Variable-Size Inputs

When the input is a variable-size signal, and you set the **Mode** to `Running`, then:

- If you set the **Input processing** parameter to `Elements as channels (sample based)`, the state is reset.
- If you set the **Input processing** parameter to `Columns as channels (frame based)`, then:
  - When the input size difference is in the number of channels (columns), the state is reset.
  - When the input size difference is in the length of channels (rows), there is no reset and the running operation is carried out as usual.

`Index base` — Base of the maximum value index

`One` (default) | `Zero`

Specify whether the index of the maximum value is reported using one-based or zero-based numbering.

Dependencies

To enable this parameter, set **Mode** to either `Index` or `Value and Index`.

`Find the maximum value over` — Dimension over which the block computes the maximum value

`Each column` (default) | `Each row` | `Entire input` | `Specified dimension`

- `Each column` — The block outputs the maximum value over each column.

- `Each row` — The block outputs the maximum value over each row.
- `Entire input` — The block outputs the maximum value over the entire input.
- `Specified dimension` — The block outputs the maximum value over the dimension, specified in the **Dimension** parameter.

Dependencies

To enable this parameter, set **Mode** to `Value and Index`, `Value`, or `Index`.

`Dimension` — Custom dimension

`1` (default) | scalar

Specify the dimension (one-based value) of the input signal over which the block computes the maximum. The value of this parameter must be greater than 0 and less than the number of dimensions in the input signal.

Dependencies

To enable this parameter, set **Find the maximum value over** to `Specified dimension`.

`Input processing` — Method to process the input in running mode

`Columns as channels (frame based)` (default) | `Elements as channels (sample based)`

- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel. This option does not support an N-dimensional input signal, where N > 2. For a two-dimensional input signal of size *M*-by-*N*, the block outputs an *M*-by-*N* matrix. Each element y_{ij} of the output contains the maximum value observed in the *j*th column of all inputs since the last reset, up to and including element u_{ij} of the current input.

  When a reset event occurs, the running maximum for each channel becomes the maximum value of all the samples in the current input frame, up to and including the current input sample.
- `Elements as channels (sample based)` — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size *M*-by-*N*-by-*P*, the block outputs an *M*-by-*N*-by-*P* array. Each y_{ijk} element of the output

contains the maximum value observed in element u_{ijk} for all inputs since the last reset.

When a reset event occurs, the running maximum y_{ijk} in the current frame is reset to the element u_{ijk}.

Dependencies

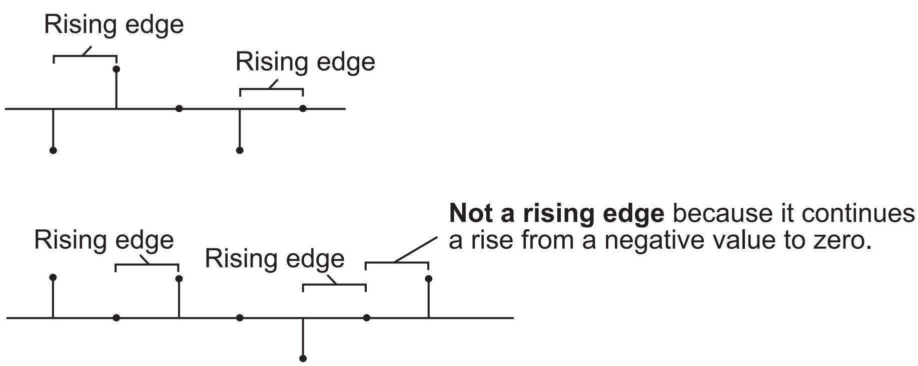
To enable this parameter, set **Mode** to `Running`.

`Reset port` — Reset event

`None` (default) | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero sample`

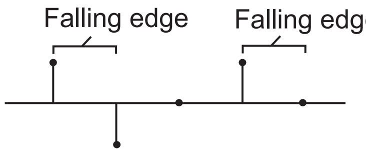
The block resets the running maximum whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer, which is a multiple of the input sample time.

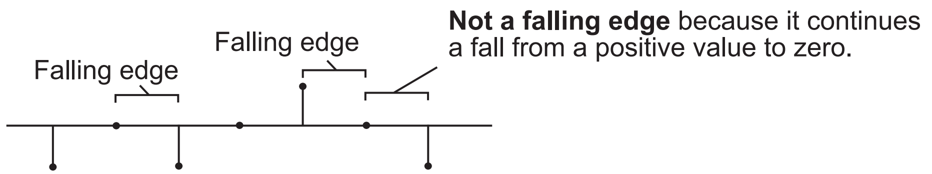
- `None` — Disables the **Rst** port.
- `Rising edge` — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.



- `Falling edge` — Triggers a reset operation when the **Rst** input does one of the following:

- Falls from a positive value to a negative value or zero.
- Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.





- `Either edge` — Triggers a reset operation when the **Rst** input is a `Rising edge` or `Falling edge`.
- `Non-zero sample` — Triggers a reset operation at each sample time that the **Rst** input is not zero.

Note: When running simulations in the Simulink `MultiTasking` mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Time-Based Scheduling and Code Generation" (Simulink Coder).

Dependencies

To enable this parameter, set **Mode** to `Running`.

Data Types Tab

Note To use these parameters, the data input must be complex and fixed-point.

`Rounding mode` — Method of rounding operation

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Select the rounding mode for fixed-point operations.

Note: The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numeric results when all these conditions are met:

- **Product output** data type is `Inherit: Inherit via internal rule`.
- **Accumulator** data type is `Inherit: Inherit via internal rule`.

With these data type settings, the block operates in full-precision mode.

`Saturate on integer overflow` — Saturate on integer overflow

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations.

Note: The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numeric results when all these conditions are met:

- **Product output** data type is `Inherit: Inherit via internal rule`.
- **Accumulator** data type is `Inherit: Inherit via internal rule`.

With these data type settings, the block operates in full-precision mode.

`Product output` — Product output data type

`Inherit: Same as input` (default) | `fixdt([],16,0)`

Product output specifies the data type of the output of a product operation in the Maximum block. For more information on the product output data type, see “Multiplication Data Types”.

- `Inherit: Same as input` — The block specifies the product output data type to be the same as the input data type.

- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink).

`Accumulator` — Accumulator data type

`Inherit: Same as product output` (default) | `Inherit: Same as input` | `fixdt([],16,0)`

Accumulator specifies the data type of the output of an accumulation operation in the Maximum block.

- `Inherit: Same as product output` — The block specifies the accumulator data type to be the same as the product output data type.
- `Inherit: Same as input` — The block specifies the accumulator data type to be the same as the input data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink).

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block dialog box.

Model Examples

Algorithms

Maximum

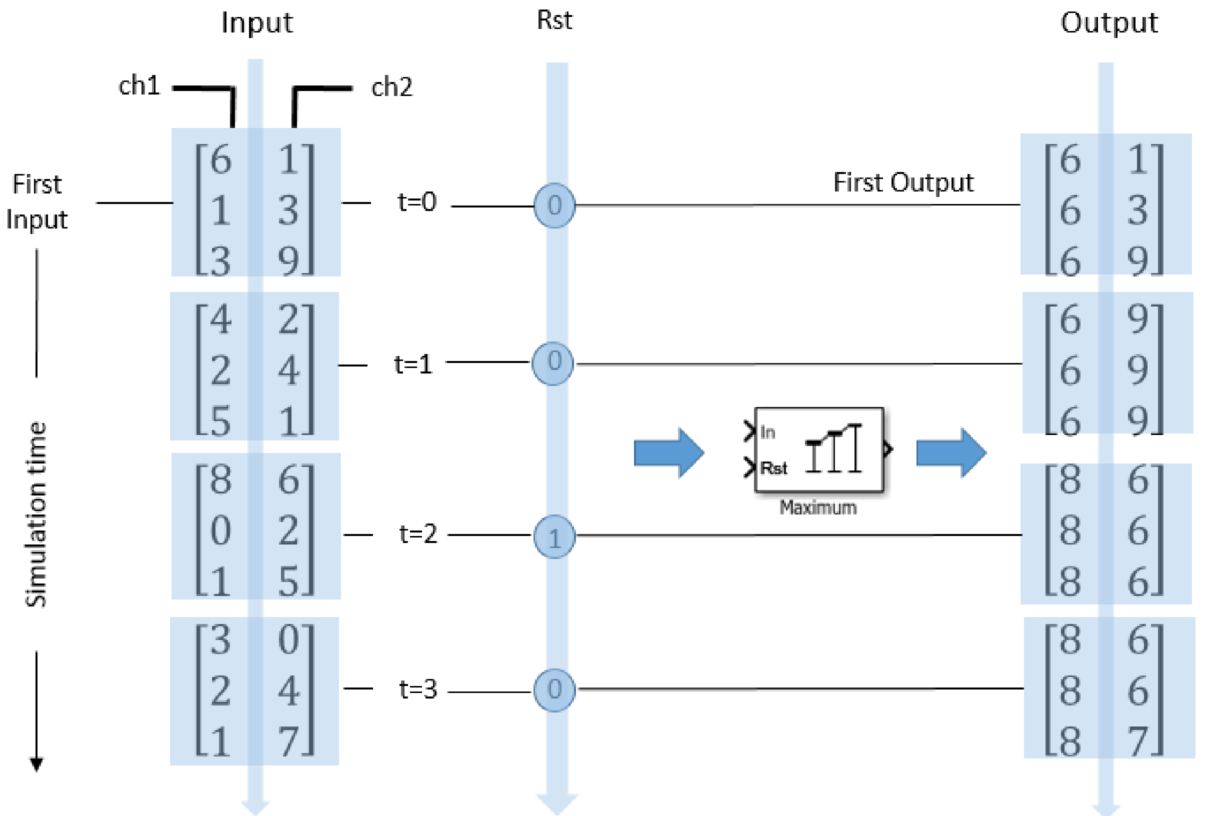
When you set **Mode** to one of `Value`, `Index`, or `Value and Index`, and specify a dimension, the block produces results identical to the MATLAB `max` function, when it is called as `[y,I] = max(u,[],D)`.

- `u` is the data input.
- `D` is the dimension.
- `y` is the maximum value.
- `I` is the index of the maximum value.

The maximum value along the entire input is identical to calling the `max` function as `[y,I] = max(u(:))`.

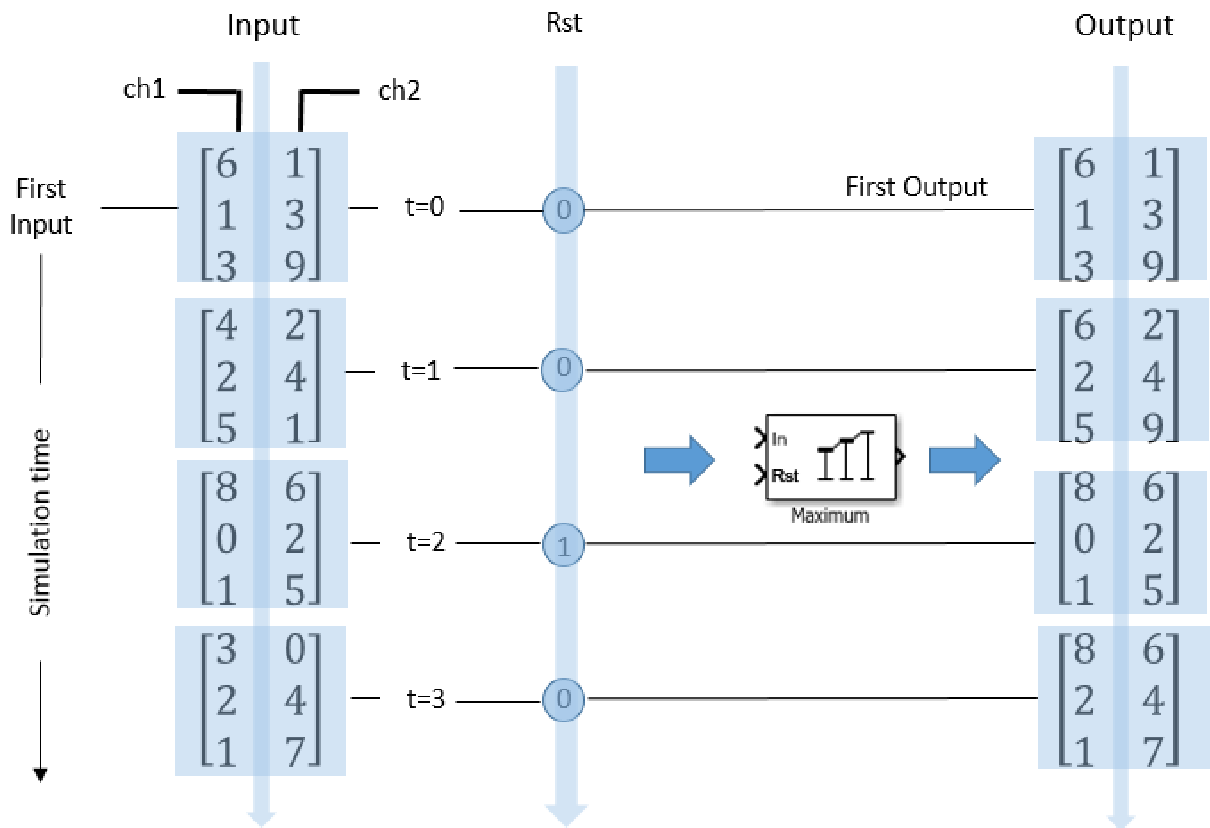
Running Maximum

When you set **Mode** to `Running`, and **Input processing** to `Columns as channels (frame based)`, the block treats each column of the input as a separate channel. In this example, the block processes a two-channel signal with a frame size of three under these settings.



The block outputs the maximum value over each channel since the last reset. At $t = 2$, the reset event occurs. The maximum value in the second column changes to 6, even though 6 is less than 9, which was the maximum value since the previous reset event.

When you set **Mode** to `Running`, and **Input processing** to `Elements as channels (sample based)`, the block treats each element of the input as a separate channel. In this example, the block processes a two-channel signal with a frame size of three under these settings.



Each y_{ij} element of the output contains the maximum value observed in element u_{ij} for all inputs since the last reset. The reset event occurs at $t = 2$. When a reset event occurs, the running maximum, y_{ij}, in the current frame is reset to element u_{ij}.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Maximum.

Fixed-Point Conversion

Design and simulate fixed-point algorithms using Fixed-Point Designer™.

The parameters on the **Data Types Tab** of the block are used only for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described under the 'Mode' parameter in “Main Tab” on page 2-1177. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

See Also

See Also

Functions

`cummax` | `max`

System Objects

dsp.Minimum | dsp.Maximum | dsp.MovingMinimum | dsp.MovingMaximum

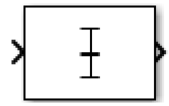
Blocks

`Mean` | `Minimum` | `Moving Maximum` | `Moving Minimum`

Introduced before R2006a

Mean

Find mean value of input or sequence of inputs
Library: DSP System Toolbox / Statistics



Description

The Mean block computes the mean of each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the mean of the entire input. You can specify the dimension using the **Find the mean value over** parameter. The Mean block can also track the mean value in a sequence of inputs over a period of time. To track the mean value in a sequence of inputs, select the **Running mean** parameter.

Note: The **Running** mode in the Mean block will be removed in a future release. To compute the running mean in Simulink, use the `Moving Average` block instead.

Ports

Input

`I` — Data input
vector | matrix | *N*-D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input data type must be double precision, single precision, integer, or fixed point with power-of-two slope and zero bias.

This port is unnamed until you select the **Running mean** parameter and set the **Reset port** parameter to any option other than `None`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed_point`

Complex Number Support: Yes

`Rst` — Reset port
scalar

Specify the reset event that causes the block to reset the running mean. The sample time of the **Rst** input must be a positive integer multiple of the input sample time.

Dependencies

To enable this port, select the **Running mean** parameter and set the **Reset port** parameter to any option other than `None`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Port_1` — Mean value along the specified dimension
scalar | vector | matrix | *N*-D array

The data type of the output matches the data type of the input.

When you do not select the **Running mean** parameter, the block computes the mean value in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the mean of the entire input at each individual sample time. Each element in the output array `y` is the mean value of the corresponding column, row, or entire input. The output array `y` depends on the setting of the **Find the mean value over** parameter. Consider a three-dimensional input signal of size *M*-by-*N*-by-*P*. When you set **Find the mean value over** to:

- `Entire input` — The output at each sample time is a scalar that contains the mean value of the *M*-by-*N*-by-*P* input matrix.
- `Each row` — The output at each sample time consists of an *M*-by-1-by-*P* array, where each element contains the mean value of each vector over the second dimension of the input. For an *M*-by-*N* matrix input, the output at each sample time is an *M*-by-1 column vector.
- `Each column` — The output at each sample time consists of a 1-by-*N*-by-*P* array, where each element contains the mean value of each vector over the first dimension

of the input. For an M-by-N matrix input, the output at each sample time is a 1-by-N row vector.

In this mode, the block treats length-M unoriented vector inputs as M-by-1 column vectors.

- `Specified dimension` — The output at each sample time depends on the value of the **Dimension** parameter. If you set the **Dimension** to `1`, the output is the same as when you select `Each column`. If you set the **Dimension** to `2`, the output is the same as when you select `Each row`. If you set the **Dimension** to `3`, the output at each sample time is an M-by-N matrix containing the mean value of each vector over the third dimension of the input.

When you select **Running mean**, the block tracks the mean value of each channel in a time sequence of inputs. In this mode, you must also specify a value for the **Input processing** parameter.

- `Elements as channels (sample based)` — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size M-by-N-by-P, the block outputs an M-by-N-by-P array. Each element y_{ijk} of the output contains the mean value of the element u_{ijk} for all inputs since the last reset.

  When a reset event occurs, the running mean y_{ijk} in the current frame is reset to the element u_{ijk}.
- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel. This option does not support an N-dimensional input signal, where N > 2. For a two-dimensional input signal of size M-by-N, the block outputs an M-by-N matrix. Each element y_{ij} of the output contains the mean of the values in the jth column of all inputs since the last reset, up to and including the element u_{ij} of the current input.

  When a reset event occurs, the running mean for each channel becomes the mean value of all the samples in the current input frame, up to and including the current input sample.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed_point`
Complex Number Support: Yes

Parameters

Main Tab

`Running mean` — Option to select running mean

off (default) | on

When you select the **Running mean** parameter, the block tracks the mean value of each channel in a time sequence of inputs.

`Find the mean value over` — Dimension over which the block computes the mean value

`Each column` (default) | `Entire input` | `Each row` | `Specified dimension`

- `Each column` — The block outputs the mean value over each column.
- `Each row` — The block outputs the mean value over each row.
- `Entire input` — The block outputs the mean value over the entire input.
- `Specified dimension` — The block outputs the mean value over the dimension, specified in the **Dimension** parameter.

Dependencies

To enable this parameter, clear the **Running mean** parameter.

`Dimension` — Custom dimension

1 (default) | scalar

Specify the dimension (one-based value) of the input signal over which the mean is computed. The value of this parameter must be greater than 0 and less than the number of dimensions in the input signal.

Dependencies

To enable this parameter, set **Find the mean value over** to `Specified dimension`.

`Input processing` — Method to process the input in running mode

`Columns as channels (frame based)` (default) | `Elements as channels (sample based)`

- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel. This option does not support an N-dimensional input signal, where N > 2. For a two-dimensional input signal of size *M*-by-*N*, the block outputs an *M*-by-*N* matrix. Each element y_{ij} of the output contains the mean of the values in the *j*th column of all inputs since the last reset, up to and including the element u_{ij} of the current input.

  When a reset event occurs, the running mean for each channel becomes the mean value of all the samples in the current input frame, up to and including the current input sample.
- `Elements as channels (sample based)` — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size *M*-by-*N*-by-*P*, the block outputs an *M*-by-*N*-by-*P* array. Each element y_{ijk} of the output contains the mean value of the element u_{ijk} for all inputs since the last reset.

  When a reset event occurs, the running mean y_{ijk} in the current frame is reset to the element u_{ijk}.

  Variable-Size Inputs

  When your inputs are of variable size, and you select the **Running mean** parameter, then:

  - If you set the **Input processing** parameter to `Elements as channels (sample based)`, the state is reset.
  - If you set the **Input processing** parameter to `Columns as channels (frame based)`, then:
    - When the input size difference is in the number of channels (number of columns), the state is reset.
    - When the input size difference is in the length of channels (number of rows), there is no reset and the running operation is carried out as usual.

Dependencies

To enable this parameter, select the **Running mean** parameter.

`Reset port` — Reset event

`None` (default) | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero sample`

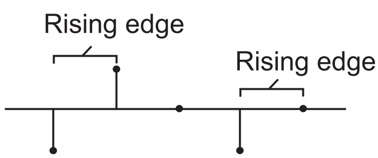
The block resets the running mean whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

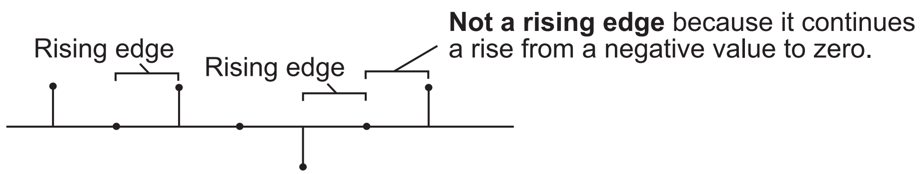
When a reset event occurs while the **Input processing** parameter is set to `Elements as channels (sample based)`, the running mean for each channel is initialized to the value in the corresponding channel of the current input. Similarly, when the **Input processing** parameter is set to `Columns as channels (frame based)`, the running mean for each channel becomes the mean value of all the samples in the current input frame, up to and including the current input sample.

Use this parameter to specify the reset event.

- `None` — Disables the **Rst** port.
- `Rising edge` — Triggers a reset operation when the **Rst** input does one of the following:

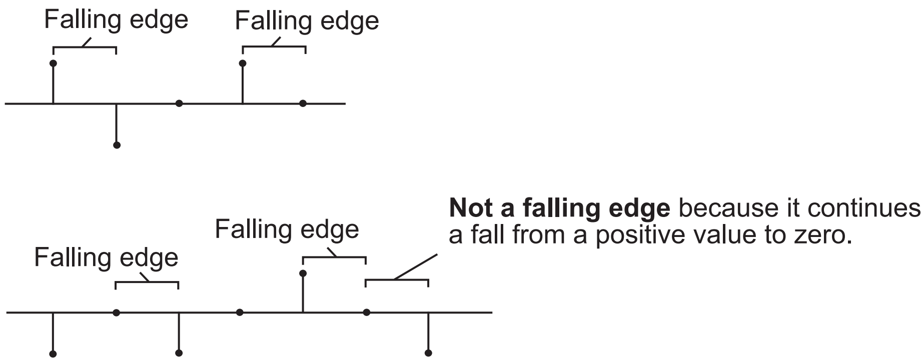
  - Rises from a negative value to either a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.





- `Falling edge` — Triggers a reset operation when the **Rst** input does one of the following:

  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.



- `Either edge` — Triggers a reset operation when the **Rst** input is a `Rising edge` or `Falling edge`.
- `Non-zero sample` — Triggers a reset operation at each sample time, when the **Rst** input is not zero.

Note: When running simulations in the Simulink multitasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Time-Based Scheduling and Code Generation" (Simulink Coder).

Dependencies

To enable this parameter, select the **Running mean** parameter.

Data Types Tab

Note: To use these parameters, the data input must be fixed-point. For all other inputs, the parameters on the **Data Types** tab are ignored.

`Rounding mode` — Method of rounding operation

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Select the rounding mode for fixed-point operations.

`Overflow mode` — Method of overflow action

`Wrap` (default) | `Saturate`

Select the overflow mode for fixed-point operations.

`Accumulator` — Accumulator data type

`Inherit: Same as input` (default) | `fixdt([],16,0)`

Accumulator specifies the data type of the output of an accumulation operation in the Mean block. See “Fixed Point” on page 2- for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- `Inherit: Same as input` — The block specifies the accumulator data type to be the same as the input data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button 

.

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink).

`Output` — Output data type

`Inherit: Same as accumulator` (default) | `fixdt([],16,0)`

Output specifies the data type of the output of the Mean block. See “Fixed Point” on page 2- for illustrations depicting the use of the output data type in this block. You can set it to:

- `Inherit: Same as accumulator` — The block specifies the output data type to be the same as the accumulator data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button 

.

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink).

`Output Minimum` — Minimum value the block can output

[ ] (default) | scalar

Specify the minimum value that the block can output. The default value is `[]` (unspecified). Simulink uses this value to perform:

- Simulation range checking. See “Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

`Output Maximum` — Maximum value the block can output

[ ] (default) | scalar

Specify the maximum value that the block can output. The default value is `[]` (unspecified). Simulink uses this value to perform:

- Simulation range checking. See “Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block.

Model Examples

Algorithms

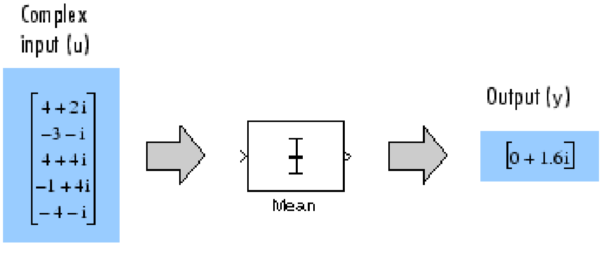
Mean

When you clear the **Running mean** parameter and specify a dimension, the block produces results identical to the MATLAB `mean` function, when it is called as `y = mean(u,D)`.

- u is the data input.
- D is the dimension.
- y is the mean value.

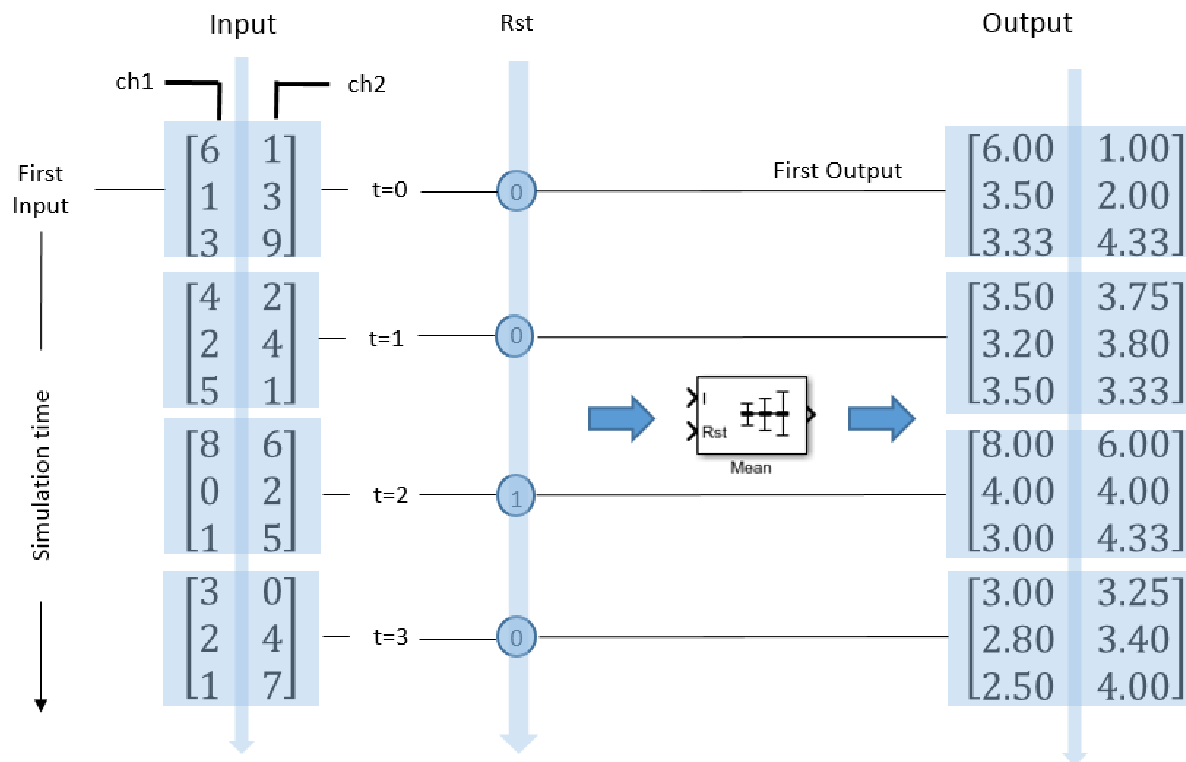
The mean value along the entire input is identical to calling the mean function as y = mean(u(:)).

The mean of a complex input is computed independently for the real and imaginary components.



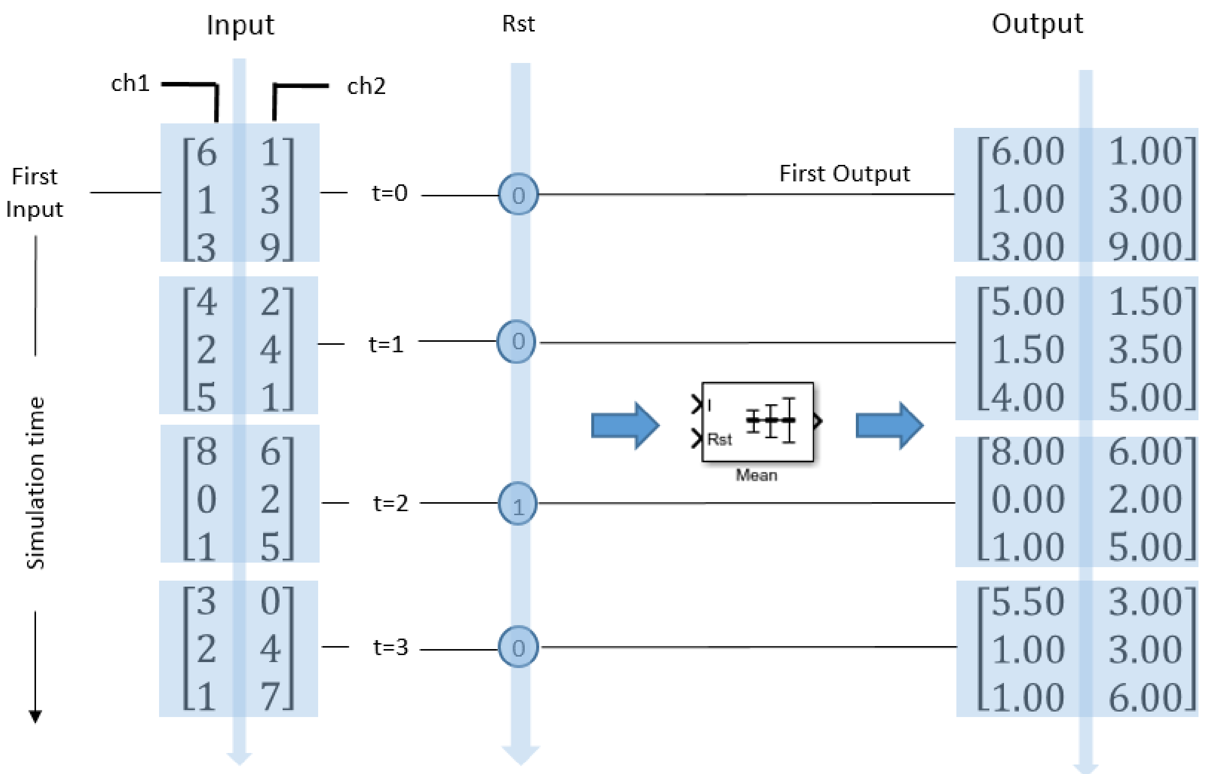
Running Mean

When you select the **Running mean** parameter, and set the **Input processing** parameter to Columns as channels (frame based), the block treats each column of the input as a separate channel. In this example, the block processes a two-channel signal with a frame size of three under these settings.



The block outputs the mean value over each channel since the last reset. At $t = 2$, the reset event occurs. The window of data in the second column now contains only 6.

When you select the **Running mean** parameter, and set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats each element of the input as a separate channel. In this example, the block processes a two-channel signal with a frame size of three under these settings.



Each y_{ij} element of the output contains the mean value observed in element u_{ij} for all inputs since the last reset. The reset event occurs at $t = 2$. When a reset event occurs, the running mean, y_{ij}, in the current frame is reset to element u_{ij}.

Extended Capabilities

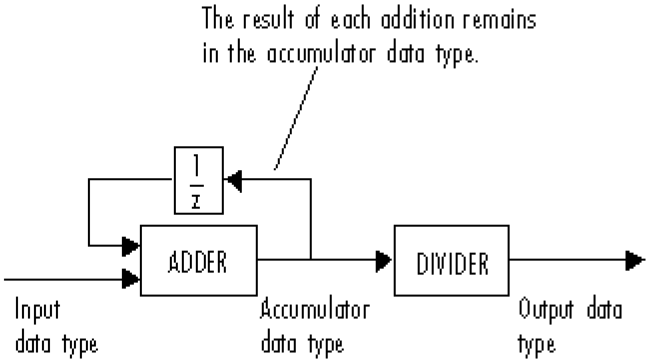
C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion

Design and simulate fixed-point algorithms using Fixed-Point Designer™.

The following diagram shows the data types used within the Mean block for fixed-point signals.



See Also

See Also

Functions

`mean`

System Objects

dsp.Mean | dsp.MovingAverage | dsp.MedianFilter

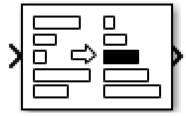
Blocks

`Maximum` | `Median` | `Median Filter` | `Minimum` | `Moving Average`

Introduced before R2006a

Median

Median value of input
Library: DSP System Toolbox / Statistics



Description

The Median block computes the median of each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the median of the entire input. You can specify the dimension using the **Find the median value over** parameter. While computing the median, the block first sorts the input values. If the number of values is odd, the median is the middle value. If the number of values is even, the median is the average of the two middle values. To sort the data, you can specify the **Sort algorithm** parameter as either `Quick sort` or `Insertion sort`. The block sorts complex inputs according to their magnitude.

Ports

Input

`Port_1` — Data input

vector | matrix | *N*-D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input data type must be double precision, single precision, integer, or fixed point, with power-of-two slope and zero bias.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed_point`
Complex Number Support: Yes

Output

`Port_1` — Median value along the specified dimension

vector | matrix | *N*-D array

The data type of the output matches the data type of the input.

The block computes the median value in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the median of the entire input. Each element in the output array `y` is the median value of the corresponding column, row, or entire input. The output array `y` depends on the setting of the **Find the median value over** parameter. Consider a three-dimensional input signal of size *M*-by-*N*-by-*P*. When you set **Find the median value over** to:

- `Entire input` — The output at each sample time is a scalar that contains the median value of the *M*-by-*N*-by-*P* input matrix.
- `Each row` — The output at each sample time consists of an *M*-by-1-by-*P* array, where each element contains the median value of each vector over the second dimension of the input. For an *M*-by-*N* matrix input, the output is an *M*-by-1 column vector.
- `Each column` — The output at each sample time consists of a 1-by-*N*-by-*P* array, where each element contains the median value of each vector over the first dimension of the input. For an *M*-by-*N* matrix input, the output at each sample time is a 1-by-*N* row vector.

  In this mode, the block treats length-*M* unoriented vector inputs as *M*-by-1 column vectors.
- `Specified dimension` — The output at each sample time depends on the value of the **Dimension** parameter. If you set the **Dimension** to `1`, the output is the same as when you select `Each column`. If you set the **Dimension** to `2`, the output is the same as when you select `Each row`. If you set the **Dimension** to `3`, the output at each sample time is an *M*-by-*N* matrix containing the median value of each vector over the third dimension of the input.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed_point`
Complex Number Support: Yes

Parameters

Main Tab

`Sort algorithm` — Sort method
`Quick sort` (default) | `Insertion sort`

Specify the sorting algorithm as either `Quick sort` or `Insertion sort`.

`Find the median value over` — Dimension over which the block computes the median value
`Each column` (default) | `Entire input` | `Each row` | `Specified dimension`

- `Each column` — The block outputs the median value over each column.
- `Each row` — The block outputs the median value over each row.
- `Entire input` — The block outputs the median value over the entire input.
- `Specified dimension` — The block outputs the median value over the dimension specified in the **Dimension** parameter.

`Dimension` — Custom dimension
`1` (default) | scalar

Specify the dimension (one-based value) of the input signal over which the block computes the median. The value of this parameter must be greater than 0 and less than the number of dimensions in the input signal.

Dependencies

To enable this parameter, set **Find the median value over** to `Specified dimension`.

Data Types Tab

Note: To use these parameters, the data input must be fixed point. For all other inputs, the parameters on the **Data Types** tab are ignored.

`Rounding mode` — Method of rounding operation
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Select the rounding mode for fixed-point operations.

`Overflow mode` — Method of overflow action
`Wrap` (default) | `Saturate`

Select the overflow mode for fixed-point operations.

`Product output` — Product output data type

`Inherit: Same as input` (default) | `fixdt([],16,0)`

Product output specifies the data type of the output of a product operation in the Median block. For more information, see “Fixed Point” on page 2- and “Multiplication Data Types”.

- `Inherit: Same as input` — The block specifies the product output data type to be the same as the input data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink).

`Accumulator` — Accumulator data type

`Inherit: Same as product output` (default) | `Inherit: Same as input` | `fixdt([],16,0)`

Accumulator specifies the data type of the output of an accumulation operation in the Median block. For more details, see “Fixed Point” on page 2- .

You can set this parameter to:

- `Inherit: Same as product output` — The block specifies the accumulator data type to be the same as the product output data type.
- `Inherit: Same as input` — The block specifies the accumulator data type to be the same as the input data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink).

`Output` — Output data type

`Inherit: Same as accumulator` (default) | `Inherit: Same as input` | `Inherit: Same as product output` | `fixdt([],16,0)`

Output specifies the data type of the output of the Median block. For more details, see “Fixed Point” on page 2- .

You can set this parameter to:

- `Inherit: Same as accumulator` — The block specifies the output data type to be the same as the accumulator data type.
- `Inherit: Same as input` — The block specifies the output data type to be the same as the input data type.
- `Inherit: Same as product output` — The block specifies the output data type to be the same as the product output data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink).

`Output Minimum` — Minimum value the block can output

`[]` (default) | scalar

Specify the minimum value that the block can output. The default value is `[]` (unspecified). Simulink uses this value to perform:

- Simulation range checking. See “Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

`Output Maximum` — Maximum value the block can output

`[]` (default) | scalar

Specify the maximum value that the block can output. The default value is `[]` (unspecified). Simulink uses this value to perform:

- Simulation range checking. See “Signal Ranges” (Simulink).
- Automatic scaling of fixed-point data types.

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block.

Algorithms

Median

The median of a set of data is calculated using the following steps:

1. The values are sorted using the specified sorting algorithm.
2. If the number of values is odd, the median is the middle value.
3. If the number of values is even, the median is the average of the two middle values.

The block produces results identical to the MATLAB `median` function, when it is called as `y = median(u,D)`.

- `u` is the data input.
- `D` is the dimension.
- `y` is the median value.

The median value along the entire input is identical to calling the `median` function as `y = median(u(:))`.

When the input is complex, the block sorts the data according to the magnitude. The magnitude is computed by taking the sum of the squares of the real and imaginary components of the complex input.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion

Design and simulate fixed-point algorithms using Fixed-Point Designer™.

For fixed-point inputs, you can specify the **Accumulator**, **Product output**, and **Output** data types in the block dialog box. Not all these fixed-point parameters are applicable for all types of fixed-point inputs. The following table shows when each kind of data type and scaling is used.

M is the length of the sorted data along the specified dimension. X indicates that the particular data type is applicable.

	Output data type	Accumulator data type	Product output data type
Even *M*	X	X	Not Applicable
Odd *M*	X	Not Applicable	Not Applicable
Odd *M* and complex	X	X	X
Even *M* and complex	X	X	X

When *M* is even, the **Accumulator** and **Output** data types and scalings are used for fixed-point signals. While calculating the average of the two central rows of the input matrix, the result of the sum is stored in the **Accumulator** data type and scaling. The total result of the average which is the median of the data is stored in the **Output** data type and scaling.

When the fixed-point inputs are complex, both the **Accumulator** and the **Product output** data types are used in addition to the **Output** data type. Before sorting the data, the block computes the sum of the squares of the real and imaginary components of the complex input. The results of the squares are stored in the **Product output** data type and scaling. The result of the sum of the squares is stored in the **Accumulator** data type and scaling.

For fixed-point inputs that are both complex and have even M, **Accumulator** data type in addition stores the sum of the two central rows of the input matrix. The average of the two central rows which is the median of the data is stored in the **Output** data type.

See Also

See Also

Functions
`median`

System Objects
dsp.Median | dsp.MedianFilter | dsp.HampelFilter | dsp.MovingAverage

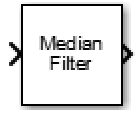
Blocks
`Hampel Filter` | `Median Filter` | `Minimum` | `Sort` | `Standard Deviation` | `Variance`

Introduced before R2006a

Median Filter

Median filter

Library: DSP System Toolbox / Filtering / Filter Designs
DSP System Toolbox / Statistics



Description

The Median Filter block computes the moving median of the input signal along each channel independently over time. The block uses the sliding window method to compute the moving median. In this method, a window of specified length moves over each channel sample by sample, and the block computes the median of the data in the window. This block performs median filtering on the input data over time. For more details, see “Algorithms” on page 2-1212.

Ports

Input

`Port_1` — Data input
column vector | row vector | matrix

Data over which the block computes moving median. The block accepts real-valued or complex-valued multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$, and $n \geq 1$. The block also accepts variable-size inputs. During simulation, you can change the size of each input channel. However, the number of channels cannot change.

Data Types: `single` | `double`
Complex Number Support: Yes

Output

`Port_1` — Moving median output
column vector | row vector | matrix

The size of the moving median output matches the size of the input. The block uses the sliding window method to compute the moving median. For more details, see “Algorithms” on page 2-1212.

Data Types: `single` | `double`
Complex Number Support: Yes

Parameters

`Window length` — Length of the sliding window
5 (default) | positive scalar integer

Window length specifies the length of the sliding window.

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

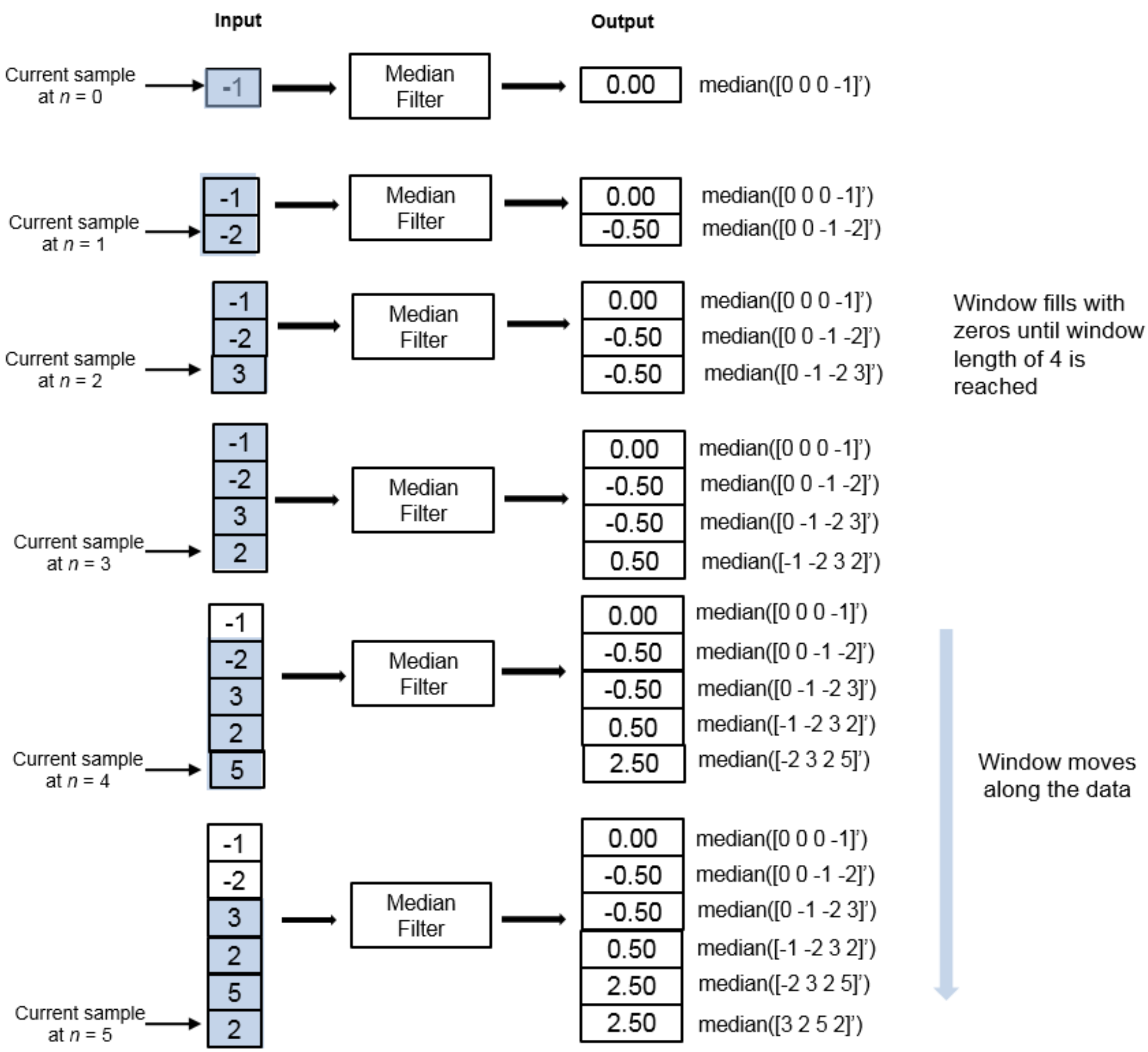
  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Algorithms

Sliding Window Method

In the sliding window method, the output for each input sample is the median of the current sample and the *Len* - 1 previous samples. *Len* is the length of the window. To compute the first *Len* - 1 outputs, when the window does not have enough data yet, the algorithm fills the window with zeros. As an example, to compute the median value when the second input sample comes in, the algorithm fills the window with *Len* - 2 zeros. The data vector, *x*, is then the two data samples followed by *Len* - 2 zeros. This object performs median filtering on the input data over time.

Consider an example of computing the moving median of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Blocks

`Median` | `Moving Average` | `Moving Maximum` | `Moving Minimum` | `Moving RMS` | `Moving Standard Deviation` | `Moving Variance`

System Objects

dsp.MedianFilter | dsp.Median | dsp.MovingRMS | dsp.MovingMaximum | dsp.MovingMinimum | dsp.MovingStandardDeviation | dsp.MovingVariance | dsp.MovingAverage

Topics

"What Are Moving Statistics?"
"Signal Statistics"
"Remove High-Frequency Noise from Gyroscope Data"

Introduced in R2016b

MIDI Controls

Output values from controls on MIDI control surface

Note: The MIDI Controls block will be removed from DSP System Toolbox in a future release. Existing instances of the block continue to run. For new code, use the MIDI Controls block from Audio System Toolbox instead.

Library

Sources

`dspsrcs4`

Description

The MIDI Controls block outputs values from controls on a MIDI control surface in real time.

Use the **MIDI device** parameter to specify the name of the MIDI control surface device from which to receive control values. You can choose:

- `Default`
- `Specify other`

If you choose `Default`, the block looks for a MATLAB preference with a group named `midi` and preference named `DefaultDevice`. You can set this preference using the MATLAB `setpref` function. For example, if the desired device is named `BCF2000`, you can type the following command at the MATLAB command line:

```
>> setpref('midi', 'DefaultDevice', 'BCF2000');
```

If the block does not find this preference, it then attempts to choose a device using an algorithm that is unspecified and platform dependent.

If you choose `Specify other`, then a **MIDI device name** edit box appears for you to enter a MATLAB expression for the device name. Enter any MATLAB expression that can evaluate to a string. Literal names must be enclosed in quotes, (for example, `'BCF2000'`).

You can determine the name of your MIDI device using the MATLAB function `midiid`, discussed in “Identifying MIDI Device Names and Control Numbers” on page 2-1217.

Use the **MIDI controls** parameter to specify the controls on the MIDI device to which the block should respond. This parameter also determines the size of the block output port. You can choose:

- `Respond to any control`
- `Respond to specified controls`

If you choose `Respond to any control`, then the block output will be a scalar. This scalar outputs the value from any and all controls that are manipulated on the MIDI device. Use this option in simple cases when you need only a single control value and the control to which it responds is unimportant.

If you choose `Respond to specified controls`, then a **MIDI control numbers** edit box opens. In this box, enter a MATLAB expression for the device control numbers. Enter any MATLAB expression that can evaluate to a row vector of real double-precision values. The block outputs a 1-D vector with one element corresponding to the output of each specified control.

Use the **Initial values** parameter to specify the value of the block output when simulation starts. The MIDI protocol transmits control values only when a control changes. This protocol provides no means for the block to query the current value of a control. Thus, the block must have some initial value to output until it receives a control change from the device.

Use the **Send initial values to device at start** check box to synchronize the device controls with the block outputs when simulation starts. Some MIDI control surfaces are bidirectional, meaning that they not only send control values but can also receive them. For example, some devices have motorized controls that move to the appropriate position when they receive a control value. If you have such a bidirectional device, select this check box. The block attempts to send the initial values to the device when the simulation starts. No diagnostic message appears if the attempt fails.

The generated code for this block relies on prebuilt .dll files. You can run this code outside the MATLAB environment, or redeploy. However you must account for these extra .dll files when doing so. The packNGo function creates a single .zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

Output Port

The MIDI Controls block output is a vector whose width is determined by the **MIDI controls** and **MIDI control numbers** parameters previously described. The output data type can be either real double-precision floating point, or uint8 integer if the output mode is `'Raw MIDI'`. The output values range from 0.0 to 1.0, inclusively, and in the raw mode, they range from 0 to 127, inclusively. The output port back inherits its sample time.

Identifying MIDI Device Names and Control Numbers

To specify a particular control on a particular MIDI device, you must know the name assigned to the device by the operating system. In addition, a number is always associated with the control. You can interactively discover this information using the MATLAB function, `midiid`. Follow these steps to identify device names and control numbers:

1. Verify that MIDI control surface device is correctly connected to the host computer running MATLAB.

   Note: For the most consistent behavior, MathWorks recommends that you connect your MIDI control surface device to your computer before starting MATLAB. In some circumstances MATLAB may not be able to find your device if you connect it after starting your MATLAB session. Also, it may not find your device if you disconnect it and reconnect it during your MATLAB session.

   ---

2. Type the following command at the MATLAB command line.

   ```
   >> [ctlnum devname] = midiid
   ```

   You are prompted to move the control in which you are interested.

   ```
   >> [ctlnum devname] = midiid
   Move the control you wish to identify; type ^C to abort.
   Waiting for control message ...
   ```

3 Move the control. `midiid` detects the movement and returns the device name and control number.

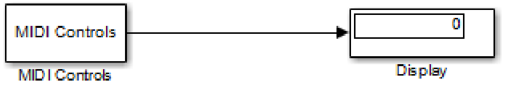
```
>> [ctlnum devname] = midiid
Move the control you wish to identify; type ^C to abort.
Waiting for control message ... done
ctlnum =
        1081
devname =
        BCF2000
>>
```

4 Use the device name in the block dialog, or set it as the default device using `setpref`. Then, enter the control number in the block dialog. Concatenate the number with other control numbers as needed.

Examples

How to Output Values from Controls

Use this example to familiarize yourself with how to set controls in the MIDI Controls block as it interacts with the MIDI control surface.



Open the ex_simplemidi model, and follow these steps:

Connect a MIDI device to the computer.

Use `midiid` to determine the name of the device, and set it on the MIDI Controls block.

Verify that any control changes the display value.

Use `midiid` to determine the number of a particular control, and set that on the MIDI Controls block.

Verify that a particular control changes the display value and that other controls do not.

Use `midiid` to determine the number of a few more controls, and set those on the MIDI Controls block.

Verify that the display block shows the correct number of values. Also verify that the controls you specified change the appropriate display values and that the other controls do not change the values.

Set each control to have a unique initial value.

Verify that the correct initial values appear on the display when the model starts.

If your MIDI device is bidirectional, on the MIDI Controls block, select the **Send initial values to device at start** check box.

Verify that the controls are set to the correct initial values when the model starts.

Parameters

MIDI device

Specify whether to use a default MIDI device, or specify a particular device by name.

MIDI device name

Specify the name of a particular MIDI control surface device from which to receive control values.

MIDI controls

Specify whether to respond to any control on the MIDI device or respond to particular specified controls.

MIDI control numbers

Specify particular controls to which the block should respond.

Initial values

Specify initial values to output when simulation starts.

Send initial values to device at start

Select this check box to attempt to synchronize a bidirectional MIDI device with block initial values when simulation starts.

Output mode

Specify the mode in which the control values are generated. When you set **Output mode** to `Normalized (0-1)`, the block generates control values in the range [`0`

1]. In this mode, control values are represented as a fraction of a full-scale. Hence, you can easily scale this range to your particular application. When you set **Output mode** to `RAW MIDI (0-127)`, the block generates byte-oriented MIDI control values in the range [`0 127`]. By default, this parameter is set to `Normalized (0-1)`.

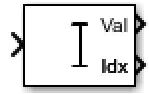
Supported Data Types

Port	Supported Data Types
Output	• Double-precision floating point, uint8 integer

Introduced in R2012a

Minimum

Minimum values of input or sequence of inputs
Library: DSP System Toolbox / Statistics



Description

The Minimum block identifies the value and position of the smallest element in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the minimum value of the entire input. The Minimum block can also track the minimum values in a sequence of inputs over a period of time. The **Mode** parameter specifies the block's mode of operation and can be set to one of the following:

- `Value` — The block outputs the minimum values in the specified dimension.
- `Index` — The block outputs the index array of the minimum values in the specified dimension.
- `Value and Index` — The block outputs the minimum values and the corresponding index array in the specified dimension.
- `Running` — The block tracks the minimum values in a sequence of inputs over a period of time.

You can specify the dimension using the **Find the minimum value over** parameter.

Note: The **Running** mode in the Minimum block will be removed in a future release. To compute the running minimum in Simulink, use the `Moving Minimum` block instead.

Ports

Input

`In` — Data input
vector | matrix | *N*-D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs. The input can be floating-point, fixed-point, or Boolean. Real fixed-point inputs can be either signed or unsigned. Complex fixed-point inputs must be signed.

This port is unnamed until you set the **Mode** parameter to `Running` and the **Reset port** parameter to any option other than `None`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed_point`
Complex Number Support: Yes

`Rst` — Reset port
scalar

Specify the reset event that causes the block to reset the running minimum. The sample time of the **Rst** input must be a positive integer multiple of the input sample time.

Dependencies

To enable this port, set the **Mode** parameter to `Running` and the **Reset port** parameter to any option other than `None`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Val` — Minimum values along the specified dimension
scalar | vector | matrix | *N*-D array

The data type of the minimum value matches the data type of the input.

When the **Mode** parameter is set to either `Value and Index` or `Value`, the following applies:

- The size of the dimension for which the block computes the minimum value is 1. The sizes of all other dimensions match those of the input array. For example, when the input is an *M*-by-*N*-by-*P* array, with the dimension set to `1`, the block outputs a 1-by-*N*-by-*P* array. When the dimension is set to `3`, the block outputs a two-dimensional *M*-by-*N* matrix.

- When the input is an M-by-N matrix, with the dimension set to `1`, the block outputs a 1-by-N matrix.

If you specify the block to compute the minimum value over the entire input, the block outputs a scalar.

When the **Mode** parameter is set `Running`, the block tracks the minimum value of each channel in a time sequence of M-by-N inputs. In this mode, you must also specify the **Input processing** parameter as one of the following:

- `Elements as channels (sample based)` — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size M-by-N-by-P, the block outputs an M-by-N-by-P array. Each y_{ijk} element of the output contains the minimum value observed in element u_{ijk} for all inputs since the last reset.

  When a reset event occurs, the running minimum y_{ijk} in the current frame is reset to the element u_{ijk}.
- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel. This option does not support an N-dimensional input signal, where N > 2. For a two-dimensional input signal of size M-by-N, the block outputs an M-by-N matrix. Each element y_{ij} of the output contains the minimum value observed in the jth column of all inputs since the last reset, up to and including element u_{ij} of the current input.

  When a reset event occurs, the running minimum for each channel becomes the minimum value of all the samples in the current input frame, up to and including the current input sample.

The block resets the running minimum whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

Dependencies

To enable this port, set the **Mode** parameter to either `Value and Index` or `Value`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed_point`
Complex Number Support: Yes

`Idx` — Index of the minimum values along the specified dimension
scalar | vector | matrix | *N*-D array

When the input is `double`, the index values are also `double`. Otherwise, the index values are `uint32`.

Dependencies

To enable this port, set the **Mode** parameter to either `Value and Index` or `Index`.

Data Types: `double` | `uint32`

Parameters

Main Tab

`Mode` — Mode in which the block operates
`Value and Index` (default) | `Value` | `Index` | `Running`

When the **Mode** parameter is set to:

- `Value`— The block computes the minimum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input at each sample time, and outputs the array *y*. Each element in *y* is the minimum value in the corresponding column, row, vector, or entire input. The output *y* depends on the setting of the **Find the minimum value over** parameter. Consider a three dimensional input signal of size *M*-by-*N*-by-*P*. Set **Find the minimum value over** to:

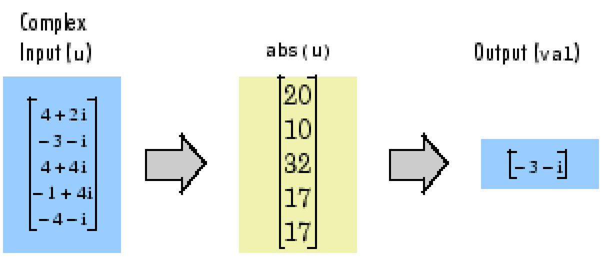
  - `Each row` — The output *y* at each sample time consists of an *M*-by-1-by-*P* array, where each element contains the minimum value of each vector over the second dimension of the input. For an input that is an *M*-by-*N* matrix, the output at each sample time is an *M*-by-1 column vector.
  - `Each column` — The output *y* at each sample time consists of a 1-by-*N*-by-*P* array, where each element contains the minimum value of each vector over the first dimension of the input. For an input that is an *M*-by-*N* matrix, the output at each sample time is a 1-by-*N* row vector.

In this mode, the block treats length-M unoriented vector inputs as M-by-1 column vectors.

- `Entire input` — The output y at each sample time is a scalar that contains the minimum value in the M-by-N-by-P input matrix.
- `Specified dimension` — The output y at each sample time depends on **Dimension**. If **Dimension** is set to `1`, the output is the same as when you select `Each column`. If **Dimension** is set to `2`, the output is the same as when you select `Each row`. If **Dimension** is set to `3`, the output at each sample time is an M-by-N matrix containing the minimum value of each vector over the third dimension of the input.

Complex Inputs

For complex inputs, the block selects the value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input that has the minimum magnitude squared in the following figure. For complex value $u = a + bi$, the magnitude squared is $a^2 + b^2$.



- `Index` — The block computes the minimum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input, and outputs the index array I. Each element in I is an integer indexing the minimum value in the corresponding column, row, vector, or entire input. The output I depends on the setting of the **Find the minimum value over** parameter. Consider a three dimensional input signal of size M-by-N-by-P:

  - `Each row` — The output I at each sample time consists of an M-by-1-by-P array, where each element contains the index of the minimum value of each vector over the second dimension of the input. For an input that is an M-by-N matrix, the output at each sample time is an M-by-1 column vector.

    - `Each column` — The output I at each sample time consists of a 1-by-N-by-P array, where each element contains the index of the minimum value of each vector over the first dimension of the input. For an input that is an M-by-N matrix, the output at each sample time is a 1-by-N row vector.

      In this mode, the block treats length-M unoriented vector inputs as M-by-1 column vectors.
    - `Entire input` — The output I at each sample time is a 1-by-3 vector that contains the location of the minimum value in the M-by-N-by-P input matrix. For an input that is an M-by-N matrix, the output is a 1-by-2 vector.
    - `Specified dimension` — The output I at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select `Each column`. If **Dimension** is set to 2, the output is the same as when you select `Each row`. If **Dimension** is set to 3, the output at each sample time is an M-by-N matrix containing the indices of the minimum values of each vector over the third dimension of the input.

  When a minimum value occurs more than once, the computed index corresponds to the first occurrence. For example, when the input is the column vector `[3 2 1 2 3]'`, the computed one-based index of the minimum value is `1`, rather than `5` when `Each column` is selected.
- `Value and Index` — The block outputs the minimum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input, and the corresponding index array I.
- `Running` — The block tracks the minimum value of each channel in a time sequence of M-by-N inputs. The block resets the running minimum whenever a reset event is detected at the optional `Rst` port. The reset sample time must be a positive integer multiple of the input sample time. In this mode, you must also specify the **Input processing** parameter as one of the following:

    - `Elements as channels (sample based)` — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size M-by-N-by-P, the block outputs an M-by-N-by-P array. Each y_{ijk} element of the output contains the minimum value observed in element u_{ijk} for all inputs since the last reset.

      When a reset event occurs, the running minimum y_{ijk} in the current frame is reset to the element u_{ijk}.

- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel. This option does not support an N-dimensional input signal, where N > 2. For a two-dimensional input signal of size *M*-by-*N*, the block outputs an *M*-by-*N* matrix. Each element y_{ij} of the output contains the minimum value observed in the *j*th column of all inputs since the last reset, up to and including element u_{ij} of the current input.

  When a reset event occurs, the running minimum for each channel becomes the minimum value of all the samples in the current input frame, up to and including the current input sample.

The block resets the running minimum whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

Running Mode for Variable-Size Inputs

When the input is a variable-size signal, and you set the **Mode** to `Running`, then:

- If you set the **Input processing** parameter to `Elements as channels (sample based)`, the state is reset.
- If you set the **Input processing** parameter to `Columns as channels (frame based)`, then:
  - When the input size difference is in the number of channels (columns), the state is reset.
  - When the input size difference is in the length of channels (rows), there is no reset and the running operation is carried out as usual.

`Index base` — Base of the minimum value index

`One` (default) | `Zero`

Specify whether the index of the minimum value is reported using one-based or zero-based numbering.

Dependencies

To enable this parameter, set **Mode** to either `Index` or `Value and Index`.

`Find the minimum value over` — Dimension over which the block computes the minimum value
`Each column` (default) | `Each row` | `Entire input` | `Specified dimension`

- `Each column` — The block outputs the minimum value over each column.
- `Each row` — The block outputs the minimum value over each row.
- `Entire input` — The block outputs the minimum value over the entire input.
- `Specified dimension` — The block outputs the minimum value over the dimension specified in the **Dimension** parameter.

Dependencies

To enable this parameter, set **Mode** to `Value and Index`, `Value`, or `Index`.

`Dimension` — Custom dimension
1 (default) | scalar

Specify the dimension (one-based value) of the input signal over which the block computes the minimum. The value of this parameter must be greater than 0 and less than the number of dimensions in the input signal.

Dependencies

To enable this parameter, set **Find the minimum value over** to `Specified dimension`.

`Input processing` — Method to process the input in running mode
`Columns as channels (frame based)` (default) | `Elements as channels (sample based)`

- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel. This option does not support an N-dimensional input signal, where N > 2. For a two-dimensional input signal of size M-by-N, the block outputs an M-by-N matrix. Each element y_{ij} of the output contains the minimum value observed in the jth column of all inputs since the last reset, up to and including element u_{ij} of the current input.

When a reset event occurs, the running minimum for each channel becomes the minimum value of all the samples in the current input frame, up to and including the current input sample.

- `Elements as channels (sample based)` — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size *M*-by-*N*-by-*P*, the block outputs an *M*-by-*N*-by-*P* array. Each y_{ijk} element of the output contains the minimum value observed in element u_{ijk} for all inputs since the last reset.

  When a reset event occurs, the running minimum y_{ijk} in the current frame is reset to the element u_{ijk}.

Dependencies

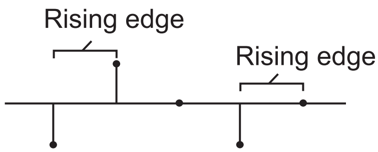
To enable this parameter, set **Mode** to `Running`.

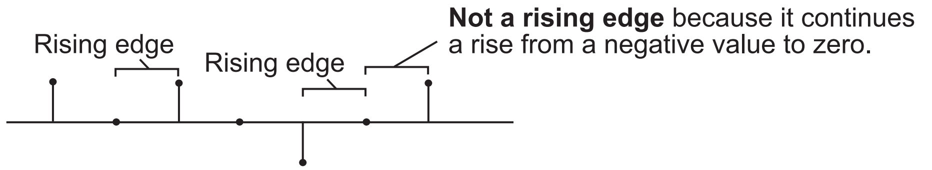
`Reset port` — Reset event

`None` (default) | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero sample`

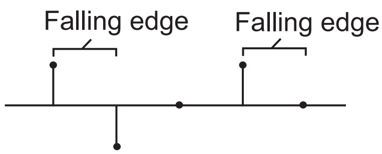
The block resets the running minimum whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer which is a multiple of the input sample time.

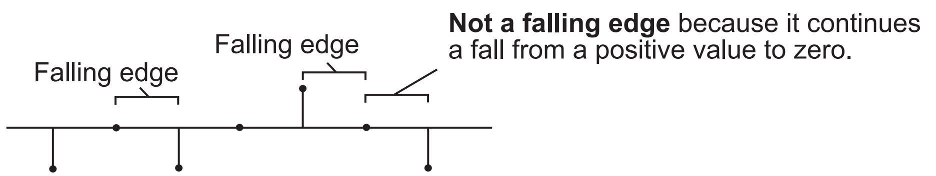
- `None` — Disables the **Rst** port.
- `Rising edge` — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.





- `Falling edge` — Triggers a reset operation when the **Rst** input does one of the following:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.





- `Either edge` — Triggers a reset operation when the **Rst** input is a `Rising edge` or `Falling edge`.
- `Non-zero sample` — Triggers a reset operation at each sample time that the **Rst** input is not zero.

Note: When running simulations in the Simulink `MultiTasking` mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

Dependencies

To enable this parameter, set **Mode** to `Running`.

Data Types Tab

Note To use these parameters, the data input must be complex fixed-point.

`Rounding mode` — Method of rounding operation

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Select the rounding mode for fixed-point operations.

Note: The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numeric results when all these conditions are met:

- **Product output** data type is `Inherit: Inherit via internal rule`.
- **Accumulator** data type is `Inherit: Inherit via internal rule`.

With these data type settings, the block operates in full-precision mode.

`Saturate on integer overflow` — Saturate on integer overflow

off (default) | on

When you select this parameter, the block saturates the result of its fixed-point operation. When you clear this parameter, the block wraps the result of its fixed-point operation. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations.

Note: The **Rounding mode** and **Saturate on integer overflow** parameters have no effect on numeric results when all these conditions are met:

- **Product output** data type is `Inherit: Inherit via internal rule`.
- **Accumulator** data type is `Inherit: Inherit via internal rule`.

With these data type settings, the block operates in full-precision mode.

`Product output` — Product output data type

`Inherit: Same as input` (default) | `fixdt([],16,0)`

Product output specifies the data type of the output of a product operation in the Minimum block. For more information on the product output data type, see "Multiplication Data Types".

- `Inherit: Same as input` — The block specifies the product output data type to be the same as the input data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink).

`Accumulator` — Accumulator data type

`Inherit: Same as product output` (default) | `Inherit: Same as input` | `fixdt([],16,0)`

Accumulator specifies the data type of the output of an accumulation operation in the Minimum block.

- `Inherit: Same as product output` — The block specifies the accumulator data type to be the same as the product output data type.
- `Inherit: Same as input` — The block specifies the accumulator data type to be the same as the input data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink).

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block dialog box.

Model Examples

Algorithms

Minimum

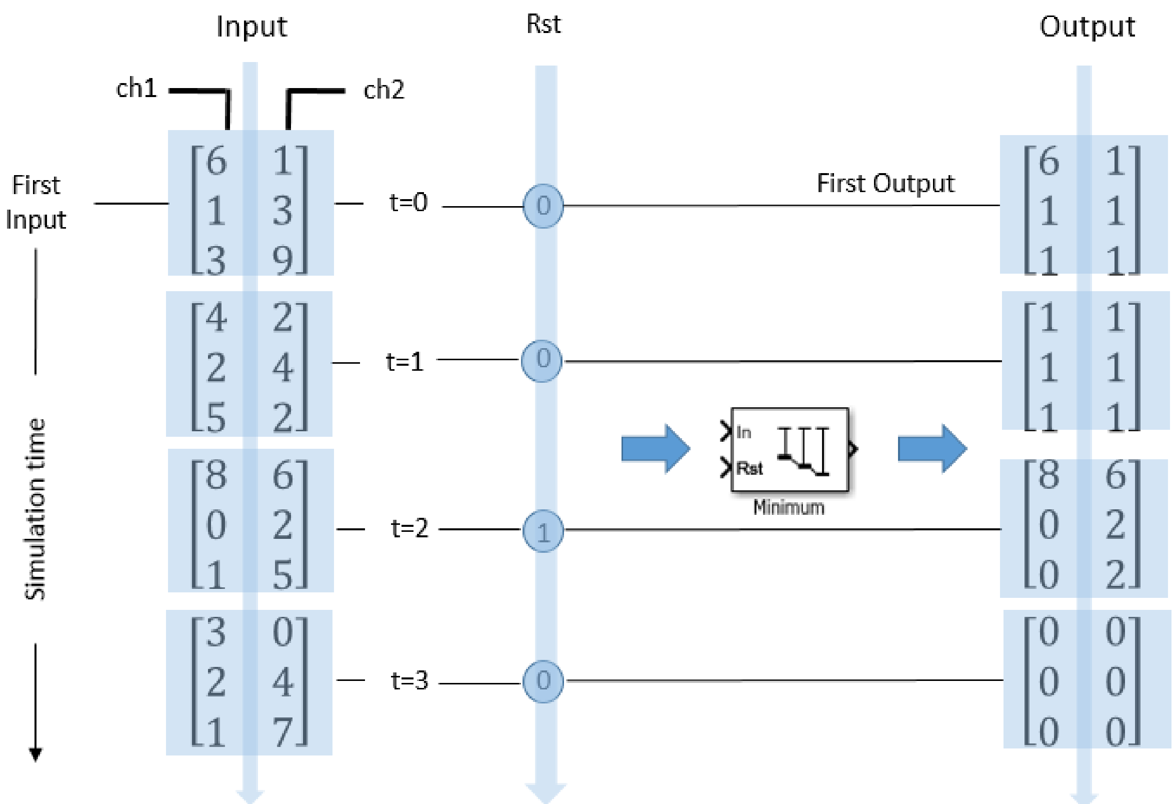
When you set **Mode** to one of `Value`, `Index`, or `Value and Index`, and specify a dimension, the block produces results identical to the MATLAB `min` function, when it is called as `[y,I] = min(u,[],D)`.

- `u` is the data input.
- `D` is the dimension.
- `y` is the minimum value.
- `I` is the index of the minimum value.

The minimum value along the entire input is identical to calling the `min` function as `[y,I] = min(u(:))`.

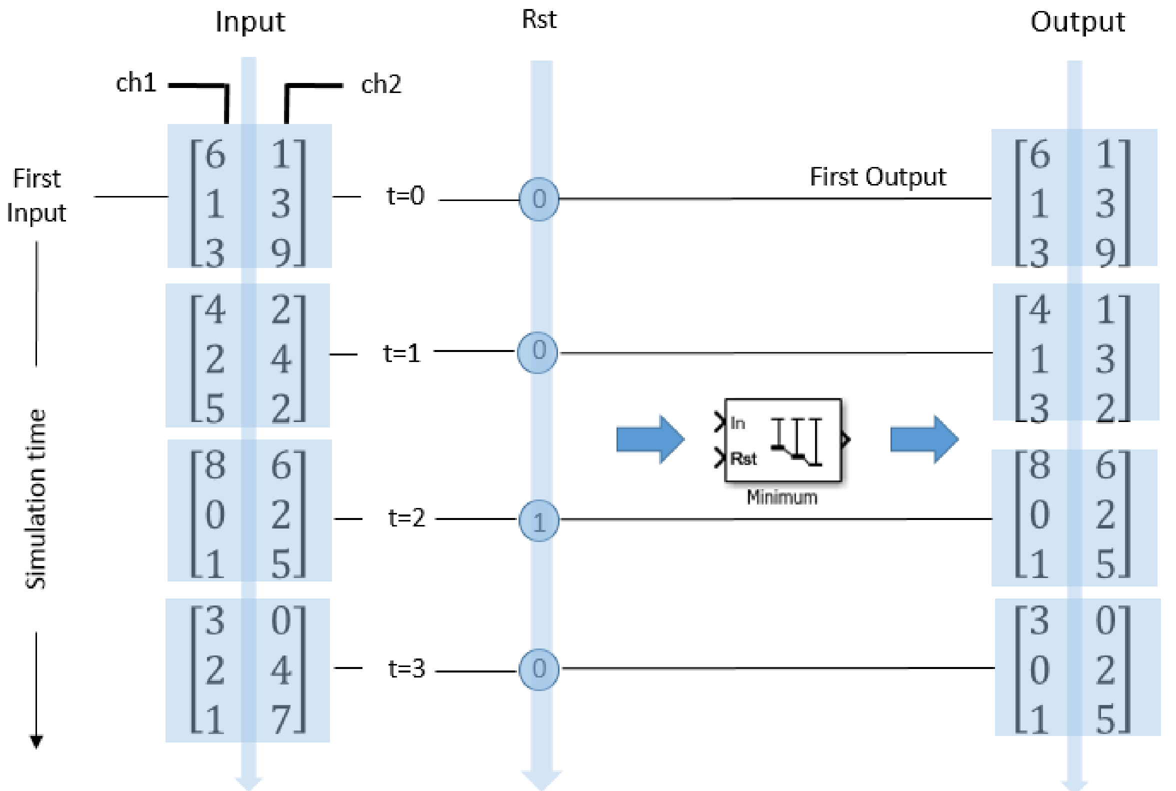
Running Minimum

When you set **Mode** to `Running`, and **Input processing** to `Columns as channels (frame based)`, the block treats each column of the input as a separate channel. In this example, the block processes a two-channel signal with a frame size of three under these settings.



The block outputs the minimum value over each channel since the last reset. At t = 2, the reset event occurs. The minimum value in the second column changes to 6, and then 2, even though these values are greater than 1, which was the minimum value since the previous reset event.

When you set **Mode** to `Running`, and **Input processing** to `Elements as channels (sample based)`, the block treats each element of the input as a separate channel. In this example, the block processes a two-channel signal with a frame size of three under these settings.



Each element y_{ij} of the output contains the minimum value observed in element u_{ij} for all inputs since the last reset. The reset event occurs at $t = 2$. When a reset event occurs, the running minimum, y_{ij}, in the current frame is reset to the element u_{ij}.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Minimum.

Fixed-Point Conversion

Design and simulate fixed-point algorithms using Fixed-Point Designer™.

The parameters on the **Data Types Tab** of the block are used only for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described under the 'Mode' parameter in “Main Tab” on page 2-1224. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

See Also

See Also

Functions

`cummin` | `min`

System Objects

dsp.Minimum | dsp.Maximum | dsp.MovingMinimum | dsp.MovingMaximum

Blocks

`Maximum` | `Mean` | `Moving Maximum` | `Moving Minimum`

Introduced before R2006a

Modified Covariance AR Estimator

Compute estimate of autoregressive (AR) model parameters using modified covariance method

Library

Estimation / Parametric Estimation

`dspparest3`

Description

The Modified Covariance AR Estimator block uses the modified covariance method to fit an autoregressive (AR) model to the input data. This method minimizes the forward and backward prediction errors in the least squares sense. The input is a frame of consecutive time samples, which is assumed to be the output of an AR system driven by white noise. The block computes the normalized estimate of the AR system parameters, *A(z)*, independently for each successive input.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

You specify the order, p, of the all-pole model in the **Estimation order** parameter. To guarantee a valid output, you must set the **Estimation order** parameter to be less than or equal to two thirds the input vector length.

The output port labeled A outputs the normalized estimate of the AR model coefficients in descending powers of z.

```
[1 a(2) ... a(p+1)]
```

The scalar gain, G, is output from the output port labeled G.

See the Burg AR Estimator block reference page for a comparison of the Burg AR Estimator, Covariance AR Estimator, Modified Covariance AR Estimator, and Yule-Walker AR Estimator blocks.

Parameters

Estimation order

Specify the order of the AR model, p.

References

Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
A	• Double-precision floating point • Single-precision floating point
G	• Double-precision floating point • Single-precision floating point

The output data type is the same as the input data type.

See Also

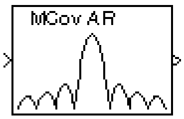
Burg AR Estimator	DSP System Toolbox

Covariance AR Estimator	DSP System Toolbox
Modified Covariance Method	DSP System Toolbox
Yule-Walker AR Estimator	DSP System Toolbox
`armcov`	Signal Processing Toolbox

Introduced before R2006a

Modified Covariance Method

Power spectral density estimate using modified covariance method



Library

Estimation / Power Spectrum Estimation

`dspspect3`

Description

The Modified Covariance Method block estimates the power spectral density (PSD) of the input using the modified covariance method. This method fits an autoregressive (AR) model to the signal. It does so by minimizing the forward and backward prediction errors in the least squares sense. The **Estimation order** parameter value specifies the order of the all-pole model. To guarantee a valid output, the **Estimation order** parameter must be less than or equal to two thirds of the input vector length. The block computes the spectrum from the FFT of the estimated AR model parameters.

The input must be a column vector. This input represents a frame of consecutive time samples from a single-channel signal. The block outputs a column vector containing the estimate of the power spectral density of the signal at N_{fft} equally spaced frequency points. The frequency points are in the range $[0,F_s)$, where F_s is the sampling frequency of the signal.

Selecting **Inherit FFT length from estimation order**, specifies that N_{fft} is one greater than the estimation order. Clearing the **Inherit FFT length from estimation order** check box allows you to use the **FFT length** parameter to specify N_{fft} as a power of 2. The block zero-pads or wraps the input to N_{fft} before computing the FFT.

When you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must hold:

- The input to the block is the original signal, with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

See the `Burg Method` block reference for a comparison of the Burg Method, Covariance Method, Modified Covariance Method, and Yule-Walker Method blocks.

Examples

The dspsacomp example compares the modified covariance method with several other spectral estimation methods.

Parameters

Estimation order

Specify the order of the AR model. To guarantee a valid output, the **Estimation order** parameter must be less than or equal to two thirds of the input vector length.

Inherit FFT length from estimation order

When you select this check box, the option specifies that the FFT length is one greater than the estimation order. To specify the number of points on which to perform the FFT, clear this check box. You can then specify a power of two FFT length using the **FFT length** parameter.

FFT length

Enter the number of data points, N_{fft}, on which to perform the FFT. When N_{fft} is larger than the input frame size, the block zero-pads each frame as needed. When N_{fft} is smaller than the input frame size, the block wraps each frame as needed. This parameter becomes visible only when you clear the **Inherit FFT length from estimation order** check box.

Inherit sample time from input

If you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must hold:

- The input to the block is the original signal, with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

Sample time of original time series

Specify the sample time of the original time-domain signal. This parameter becomes visible only when you clear the **Inherit sample time from input** check box.

References

Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L. Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

Orfanidis, S. J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

The output data type is the same as the input data type.

See Also

`Burg Method`	DSP System Toolbox
`Covariance Method`	DSP System Toolbox
`Modified Covariance AR Estimator`	DSP System Toolbox
`Short-Time FFT`	DSP System Toolbox
`Yule-Walker Method`	DSP System Toolbox

See “Spectral Analysis”for related information.

Introduced before R2006a

Moving Average

Moving average
Library: DSP System Toolbox / Statistics

Description

The Moving Average block computes the moving average of the input signal along each channel independently over time. The block uses either the sliding window method or the exponential weighting method to compute the moving average. In the sliding window method, a window of specified length moves over the data sample by sample, and the block computes the average is over the data in the window. In the exponential weighting method, the block multiplies the data samples with a set of weighting factors and then sums the weighted data to compute the average. For more details on these methods, see “Algorithms” on page 2-1246.

Ports

Input

`Port_1` — Data input

column vector | row vector | matrix

Data over which the block computes the moving average. The block accepts real-valued or complex-valued multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$ and $n \geq 1$. The block also accepts variable-size inputs. During simulation, you can change the size of each input channel. However, the number of channels cannot change.

Data Types: `single` | `double`
Complex Number Support: Yes

Output

`Port_1` — Moving average output

column vector | row vector | matrix

The size of the moving average output matches the size of the input. The block uses either the sliding window method or the exponential weighting method to compute the moving average. For more details, see “Algorithms” on page 2-1246.

Data Types: `single` | `double`
Complex Number Support: Yes

Parameters

`Method` — Averaging method

`Sliding window` (default) | `Exponential weighting`

- `Sliding window` — A window of length **Window length** moves over the input data along each channel. For every sample the window moves by, the block computes the average over the data in the window.
- `Exponential weighting` — The block multiplies the samples by a set of weighting factors. The magnitude of the weighting factors decreases exponentially as the age of the data increases, never reaching zero. To compute the average, the algorithm sums the weighted data.

`Specify window length` — Flag to specify window length

on (default) | off

When you select this check box, the length of the sliding window is equal to the value you specify in **Window length**. When you clear this check box, the length of the sliding window is infinite. In this mode, the block computes the average of the current sample and all previous samples in the channel.

`Window length` — Length of the sliding window

4 (default) | positive scalar integer

Window length specifies the length of the sliding window. This parameter appears when you select the **Specify window length** check box.

`Forgetting factor` — Exponential weighting factor

0.9 (default) | positive real scalar in the range (0,1]

This parameter applies when you set **Method** to `Exponential weighting`. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All previous samples are given an equal weight.

This parameter is tunable. You can change its value even during the simulation.

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Algorithms

Sliding Window Method

In the sliding window method, the output for each input sample is the average of the current sample and the *Len* - 1 previous samples. *Len* is the length of the window. To compute the first *Len* - 1 outputs, when the window does not have enough data yet, the algorithm fills the window with zeros. As an example, to compute the average when the second input sample comes in, the algorithm fills the window with *Len* - 2 zeros. The data vector, *x*, is then the two data samples followed by *Len* - 2 zeros.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving average of the current sample and all the previous samples in the channel.

For an example, see “Sliding Window Method and Exponential Weighting Method”.

Exponential Weighting Method

In the exponential weighting method, the moving average is computed recursively using these formulas:

$$w_{N,\lambda} = \lambda w_{N-1,\lambda} + 1,$$
$$\bar{x}_{N,\lambda} = \left(1 - \frac{1}{w_{N,\lambda}}\right)\bar{x}_{N-1,\lambda} + \left(\frac{1}{w_{N,\lambda}}\right)x_N$$

- $\bar{x}_{N,\lambda}$ — Moving average at the current sample
- x_N — Current data input sample
- $\bar{x}_{N-1,\lambda}$ — Moving average at the previous sample
- λ — Forgetting factor
- $w_{N,\lambda}$ — Weighting factor applied to the current data sample
- $\left(1 - \frac{1}{w_{N,\lambda}}\right)\bar{x}_{N-1,\lambda}$ — Effect of the previous data on the average

For the first sample, where $N = 1$, the algorithm chooses $w_{N,\lambda} = 1$. For the next sample, the weighting factor is updated and used to compute the average, as per the recursive equation. As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the current average than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the previous samples are given an equal weight.

For an example, see “Sliding Window Method and Exponential Weighting Method”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Blocks

Median Filter | Moving Maximum | Moving Minimum | Moving RMS | Moving Standard Deviation | Moving Variance

System Objects

dsp.MovingAverage | dsp.MovingRMS | dsp.MovingMaximum | dsp.MovingMinimum | dsp.MovingStandardDeviation | dsp.MovingVariance | dsp.MedianFilter

Topics

"What Are Moving Statistics?"
"Sliding Window Method and Exponential Weighting Method"

Introduced in R2016b

Moving Maximum

Moving maximum
Library: DSP System Toolbox / Statistics

Description

The Moving Maximum block determines the moving maximum of the input signal along each channel independently over time. The block uses the sliding window method to determine the moving maximum. In this method, a window of specified length moves over each channel sample by sample, and the block determines the maximum over the data in the window. For more details, see "Algorithms" on page 2-1251.

Ports

Input

`Port_1` — Data input

column vector | row vector | matrix

Data over which the moving maximum is determined using the sliding window method. The block accepts real-valued or complex-valued multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$ and $n \geq 1$. The block also accepts variable-size inputs. During simulation, you can change the size of each input channel. However, the number of channels cannot change.

Data Types: `single` | `double`
Complex Number Support: Yes

Output

`Port_1` — Moving maximum output

column vector | row vector | matrix

Moving maximum output, determined using the sliding window method. The size of the output matches the size of the input. The window slides column-wise along each channel, and the block determines the maximum of the data in the window. For more details, see "Algorithms" on page 2-1251.

Data Types: `single` | `double`
Complex Number Support: Yes

Parameters

`Specify window length` — Flag to specify window length
on (default) | off

When you select this check box, the length of the sliding window is equal to the value you specify through the **Window length** parameter. When you clear this check box, the length of the sliding window is infinite. In this mode, the block determines the maximum of the current sample and all previous samples in the channel.

`Window length` — Length of the sliding window
4 (default) | positive scalar integer

Window length specifies the length of the sliding window. This parameter appears when you select the **Specify window length** check box.

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

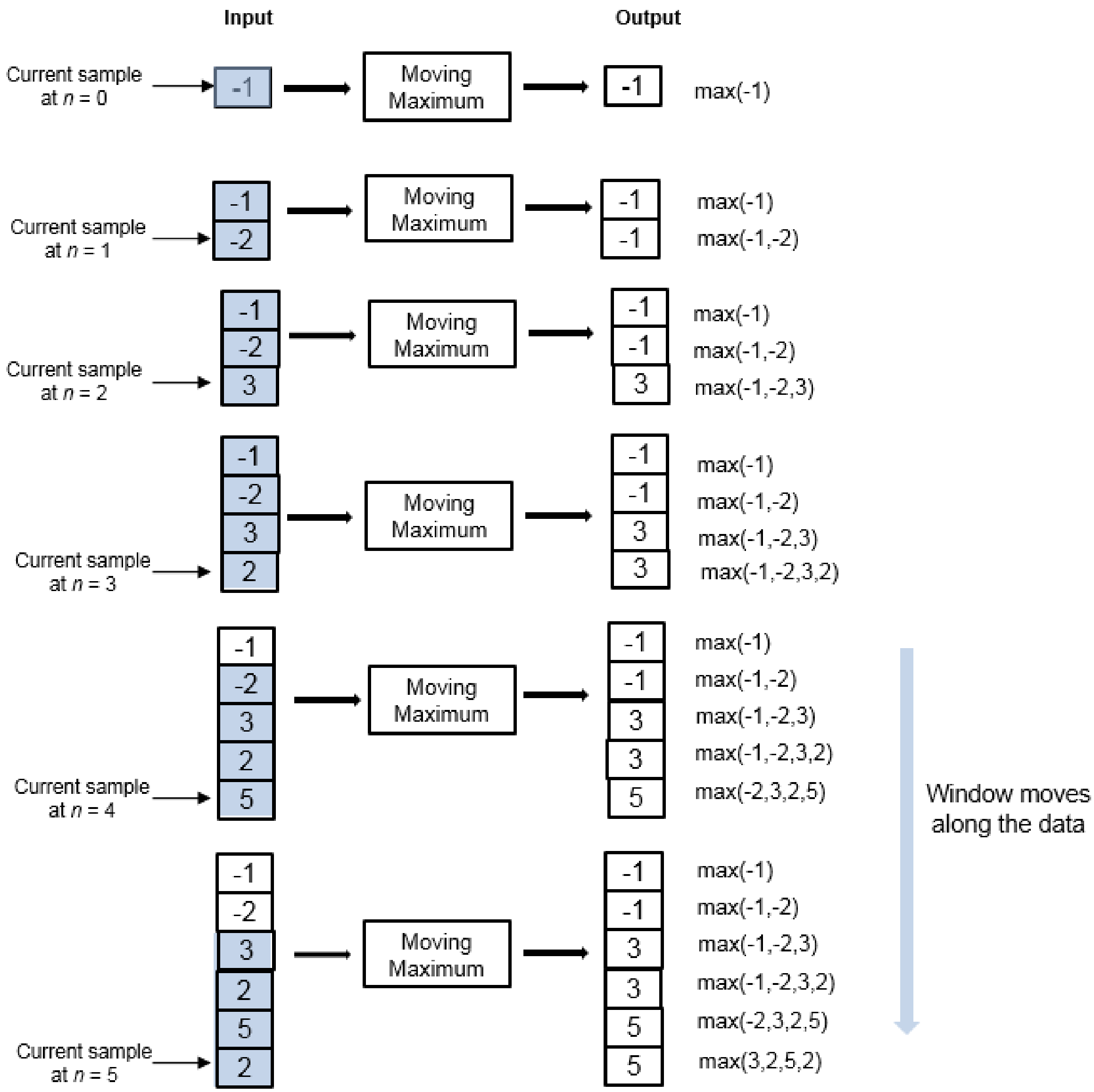
Algorithms

Sliding Window Method

In the sliding window method, the output for each input sample is the maximum of the current sample and the *Len* - 1 previous samples. *Len* is the length of the window. When the algorithm computes the first *Len* - 1 outputs, the length of the window is the length of the data that is available.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the maximum of the current sample and all the previous samples in the channel.

Consider an example of computing the moving maximum of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.


Input
Output
Current sample at n = 0
-1
Moving Maximum
-1
max(-1)
Current sample at n = 1
-1
-2
Moving Maximum
-1
-1
max(-1)
max(-1,-2)
Current sample at n = 2
-1
-2
3
Moving Maximum
-1
-1
3
max(-1)
max(-1,-2)
max(-1,-2,3)
Current sample at n = 3
-1
-2
3
2
Moving Maximum
-1
-1
3
3
max(-1)
max(-1,-2)
max(-1,-2,3)
max(-1,-2,3,2)
Current sample at n = 4
-1
-2
3
2
5
Moving Maximum
-1
-1
3
3
5
max(-1)
max(-1,-2)
max(-1,-2,3)
max(-1,-2,3,2)
max(-2,3,2,5)
Window moves along the data
Current sample at n = 5
-1
-2
3
2
5
2
Moving Maximum
-1
-1
3
3
5
5
max(-1)
max(-1,-2)
max(-1,-2,3)
max(-1,-2,3,2)
max(-2,3,2,5)
max(3,2,5,2)

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Blocks

`Maximum` | `Median Filter` | `Moving Average` | `Moving Minimum` | `Moving RMS` | `Moving Standard Deviation` | `Moving Variance`

System Objects

dsp.Maximum | dsp.MovingMaximum | dsp.MovingMinimum | dsp.MovingAverage | dsp.MovingRMS | dsp.MovingStandardDeviation | dsp.MovingVariance | dsp.MedianFilter

Topics

"Signal Statistics"
"What Are Moving Statistics?"

Introduced in R2016b

Moving Minimum

Moving minimum
Library: DSP System Toolbox / Statistics

Description

The Moving Minimum block determines the moving minimum of the input signal along each channel independently over time. The block uses the sliding window method to determine the moving minimum. In this method, a window of specified length moves over each channel sample by sample, and the block determines the minimum over the data in the window. For more details, see “Algorithms” on page 2-1256.

Ports

Input

`Port_1` — Data input
column vector | row vector | matrix

Data over which the moving minimum is determined using the sliding window method. The block accepts real-valued or complex-valued multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$ and $n \geq 1$. The block also accepts variable-size inputs. During simulation, you can change the size of each input channel. However, the number of channels cannot change.

Data Types: `single` | `double`
Complex Number Support: Yes

Output

`Port_1` — Moving minimum output
column vector | row vector | matrix

Moving minimum output, determined using the sliding window method. The size of the output matches the size of the input. The window slides column-wise along each channel, and the block determines the minimum of the data in the window. For more details, see “Algorithms” on page 2-1256.

Data Types: `single` | `double`
Complex Number Support: Yes

Parameters

`Specify window length` — Flag to specify window length

on (default) | off

When you select this check box, the length of the sliding window is equal to the value you specify through the **Window length** parameter. When you clear this check box, the length of the sliding window is infinite. In this mode, the block determines the minimum of the current sample and all the previous samples in the channel.

`Window length` — Length of the sliding window

4 (default) | positive scalar integer

Window length specifies the length of the sliding window. This parameter appears when you select the **Specify window length** check box.

`Simulate using` — Type of simulation to run

`Code generation` (default) | `Interpreted execution`

- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

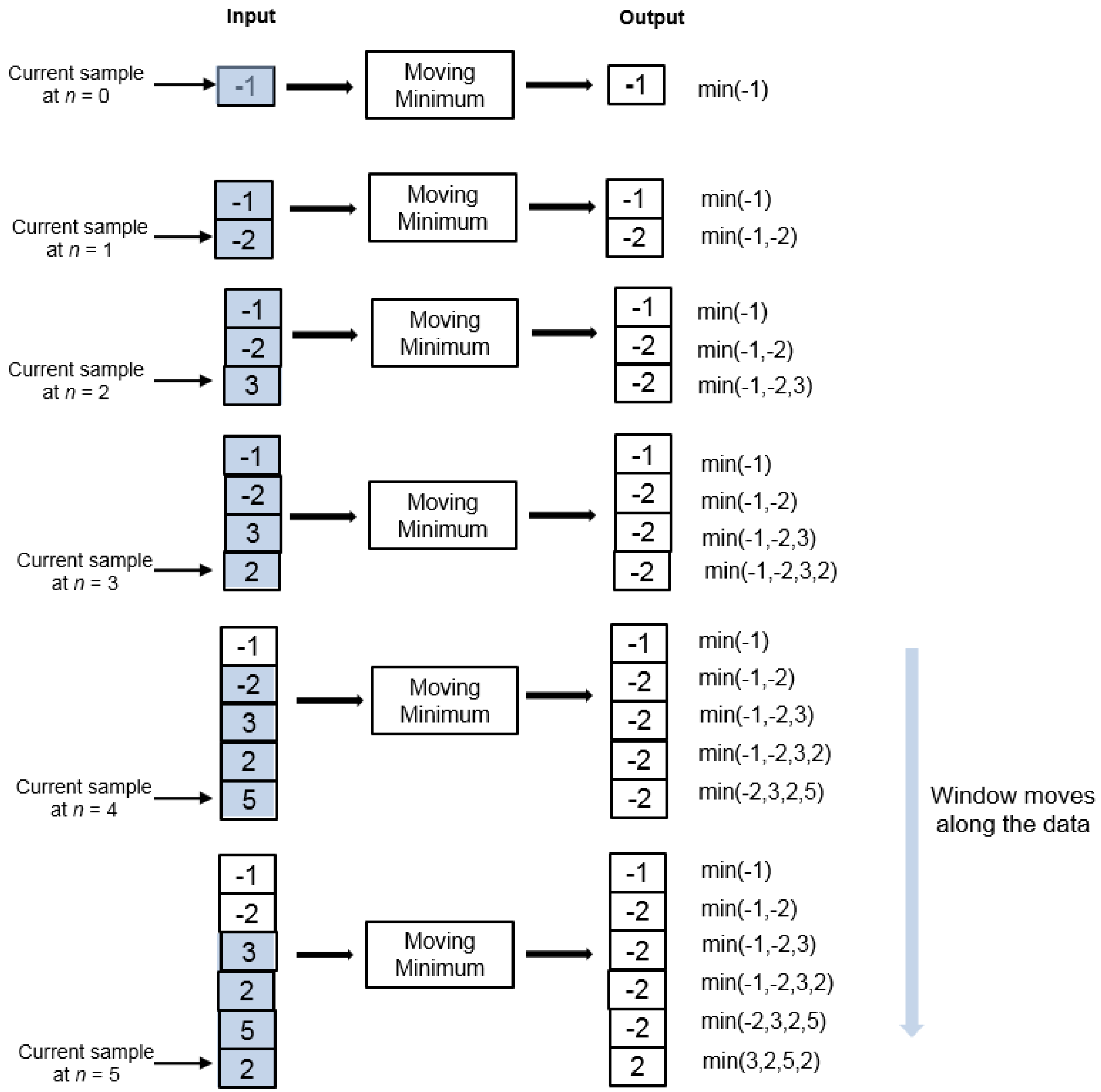
Algorithms

Sliding Window Method

In the sliding window method, the output for each input sample is the minimum of the current sample and the *Len* - 1 previous samples. *Len* is the length of the window. When the algorithm computes the first *Len* - 1 outputs, the length of the window is the length of the data that is available.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the minimum of the current sample and all the previous samples in the channel.

Consider an example of computing the moving minimum of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.


Input
Output
Current sample at n = 0
-1
Moving Minimum
-1
min(-1)
Current sample at n = 1
-1
-2
Moving Minimum
-1
-2
min(-1)
min(-1,-2)
Current sample at n = 2
-1
-2
3
Moving Minimum
-1
-2
-2
min(-1)
min(-1,-2)
min(-1,-2,3)
Current sample at n = 3
-1
-2
3
2
Moving Minimum
-1
-2
-2
-2
min(-1)
min(-1,-2)
min(-1,-2,3)
min(-1,-2,3,2)
Current sample at n = 4
-1
-2
3
2
5
Moving Minimum
-1
-2
-2
-2
-2
min(-1)
min(-1,-2)
min(-1,-2,3)
min(-1,-2,3,2)
min(-2,3,2,5)
Current sample at n = 5
-1
-2
3
2
5
2
Moving Minimum
-1
-2
-2
-2
-2
2
min(-1)
min(-1,-2)
min(-1,-2,3)
min(-1,-2,3,2)
min(-2,3,2,5)
min(3,2,5,2)
Window moves along the data

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Blocks

`Median Filter` | `Minimum` | `Moving Average` | `Moving Maximum` | `Moving RMS` | `Moving Standard Deviation` | `Moving Variance`

System Objects

dsp.MovingMinimum | dsp.Minimum | dsp.MovingMaximum | dsp.MovingAverage | dsp.MovingRMS | dsp.MovingStandardDeviation | dsp.MovingVariance | dsp.MedianFilter

Introduced in R2016b

Moving RMS

Moving RMS
Library: DSP System Toolbox / Statistics

Description

The Moving RMS block computes the moving root mean square (RMS) of the input signal along each channel independently over time. The block uses either the sliding window method or the exponential weighting method to compute the moving RMS. In the sliding window method, a window of specified length moves over the data sample by sample, and the block computes the RMS over the data in the window. In the exponential weighting method, the block squares the data samples, multiplies them with a set of weighting factors, and sums the weighed data. The block then computes the RMS by taking the square root of the sum. For more details on these methods, see "Algorithms" on page 2-1261.

Ports

Input

`Port_1` — Data input

column vector | row vector | matrix

Data over which the block computes the moving RMS. The block accepts real-valued or complex-valued multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$ and $n \geq 1$. The block also accepts variable-size inputs. During simulation, you can change the size of each input channel. However, the number of channels cannot change.

Data Types: `single` | `double`
Complex Number Support: Yes

Output

`Port_1` — Moving RMS output

column vector | row vector | matrix

The size of the moving RMS output matches the size of the input. The block uses either the sliding window method or the exponential weighting method to compute the moving RMS. For more details, see “Algorithms” on page 2-1261.

Data Types: `single` | `double`
Complex Number Support: Yes

Parameters

`Method` — Moving RMS method

`Sliding window` (default) | `Exponential weighting`

- `Sliding window` — A window of length **Window length** moves over the input data along each channel. For every sample the window moves by, the block computes the RMS over the data in the window.
- `Exponential weighting` — The block multiplies the squares of the samples by a set of weighting factors. The magnitude of the weighting factors decreases exponentially as the age of the data increases, never reaching zero. To compute the RMS, the algorithm sums the weighted data and takes a square root of the sum.

`Specify window length` — Flag to specify window length

on (default) | off

When you select this check box, the length of the sliding window is equal to the value you specify in **Window length**. When you clear this check box, the length of the sliding window is infinite. In this mode, the block computes the RMS of the current sample and all the previous samples in the channel.

`Window length` — Length of the sliding window

4 (default) | positive scalar integer

Window length specifies the length of the sliding window. This parameter appears when you select the **Specify window length** check box.

`Forgetting factor` — Exponential weighting factor

0.9 (default) | positive real scalar in the range (0,1]

This parameter applies when you set **Method** to `Exponential weighting`. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the previous samples are given an equal weight.

This parameter is tunable. You can change its value even during the simulation.

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

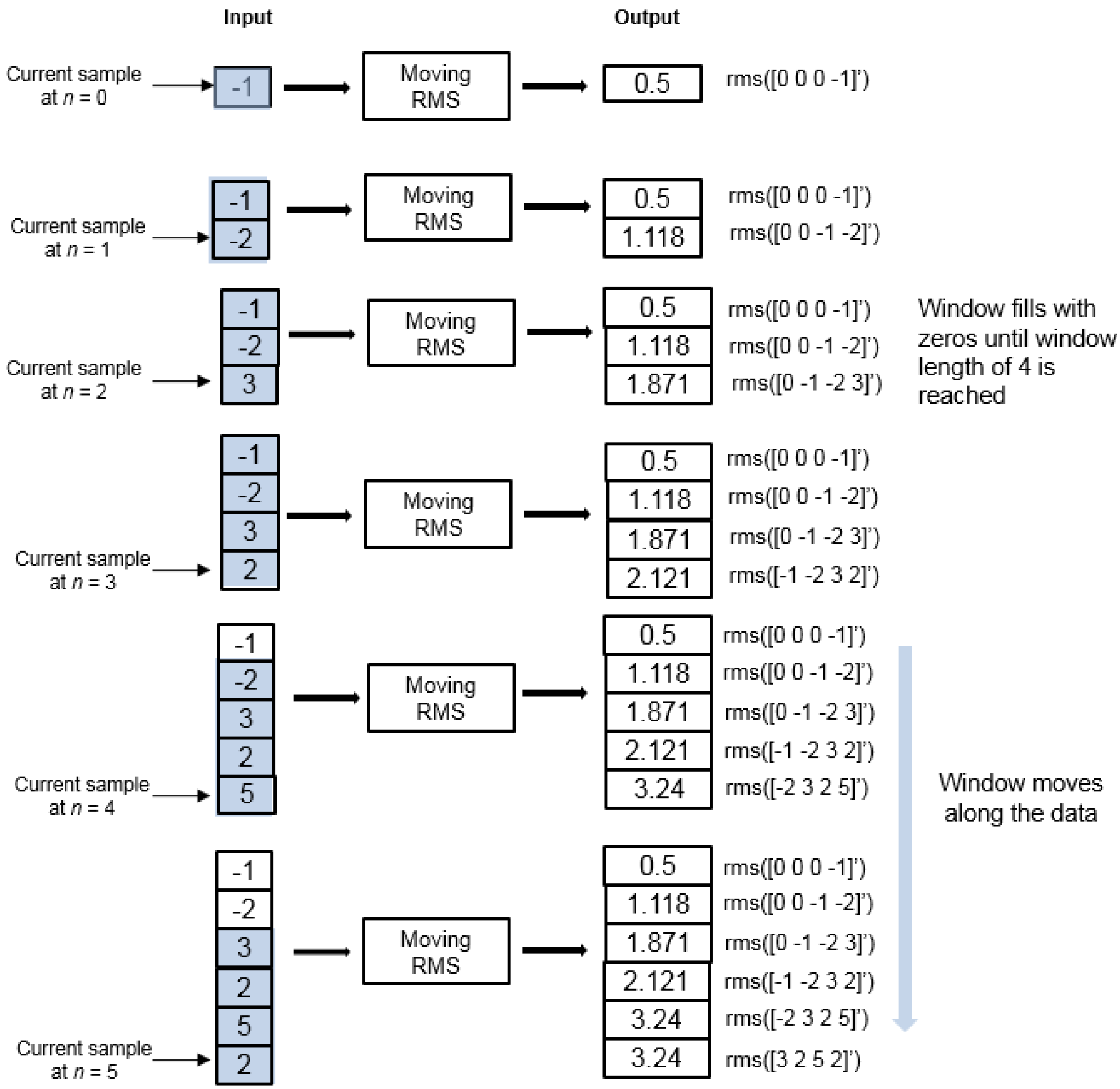
Algorithms

Sliding Window Method

In the sliding window method, the output for each input sample is the RMS of the current sample and the *Len* - 1 previous samples. *Len* is the length of the window. To compute the first *Len* - 1 outputs, when the window does not have enough data yet, the algorithm fills the window with zeros. As an example, to compute the RMS when the second input sample comes in, the algorithm fills the window with *Len* - 2 zeros. The data vector, *x*, is then the two data samples followed by *Len* - 2 zeros.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving RMS of the current sample and all the previous samples in the channel.

Consider an example of computing the moving RMS of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.


Input
Output
Current sample at n = 0
-1
Moving RMS
0.5
rms([0 0 0 -1]')
Current sample at n = 1
-1
-2
Moving RMS
0.5
1.118
rms([0 0 0 -1]')
rms([0 0 -1 -2]')
Current sample at n = 2
-1
-2
3
Moving RMS
0.5
1.118
1.871
rms([0 0 0 -1]')
rms([0 0 -1 -2]')
rms([0 -1 -2 3]')
Window fills with zeros until window length of 4 is reached
Current sample at n = 3
-1
-2
3
2
Moving RMS
0.5
1.118
1.871
2.121
rms([0 0 0 -1]')
rms([0 0 -1 -2]')
rms([0 -1 -2 3]')
rms([-1 -2 3 2]')
Current sample at n = 4
-1
-2
3
2
5
Moving RMS
0.5
1.118
1.871
2.121
3.24
rms([0 0 0 -1]')
rms([0 0 -1 -2]')
rms([0 -1 -2 3]')
rms([-1 -2 3 2]')
rms([-2 3 2 5]')
Window moves along the data
Current sample at n = 5
-1
-2
3
2
5
2
Moving RMS
0.5
1.118
1.871
2.121
3.24
3.24
rms([0 0 0 -1]')
rms([0 0 -1 -2]')
rms([0 -1 -2 3]')
rms([-1 -2 3 2]')
rms([-2 3 2 5]')
rms([3 2 5 2]')

Exponential Weighting Method

In the exponential weighting method, the moving RMS is computed recursively using these formulas:

$$w_{N,\lambda} = \lambda w_{N-1,\lambda} + 1,$$

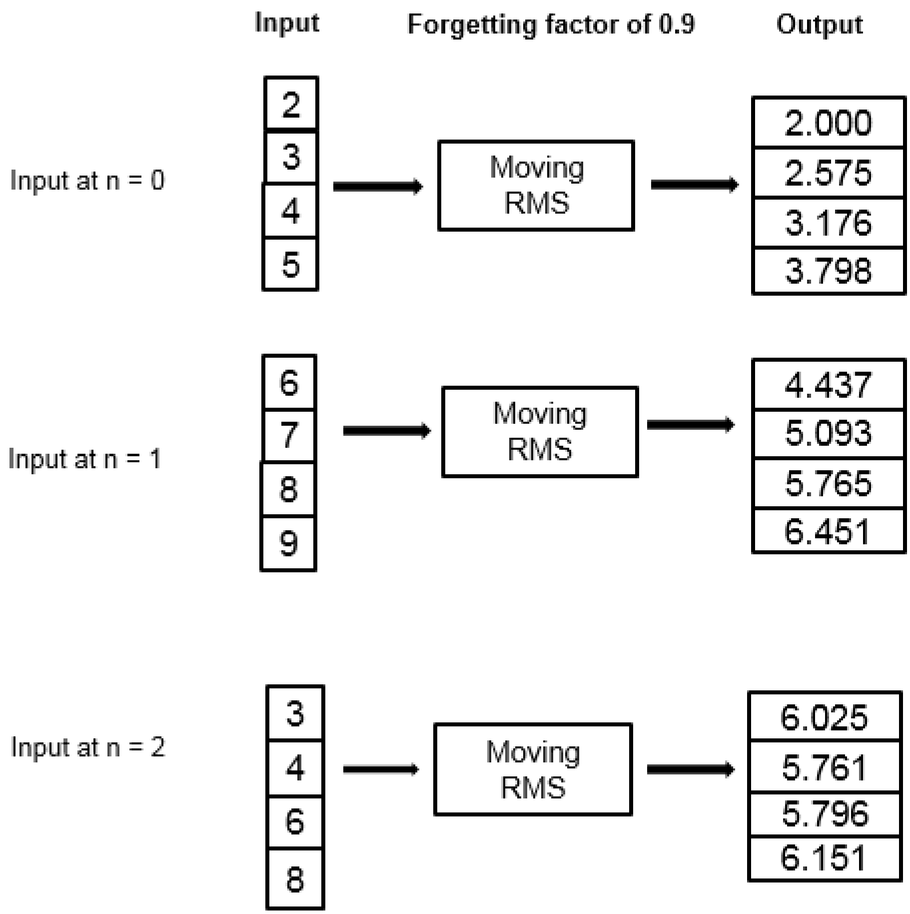
$$x\_rms_{N,\lambda} = \sqrt{\left(1 - \frac{1}{w_{N,\lambda}}\right) x\_rms_{N-1,\lambda} + \left(\frac{1}{w_{N,\lambda}}\right) x^2{}_N}$$

- $x\_rms_{N,\lambda}$ — Moving RMS at the current sample
- $x^2{}_N$ — Square of the current input data sample
- $x\_rms_{N-1,\lambda}$ — Moving RMS at the previous sample
- λ — Forgetting factor
- $w_{N,\lambda}$ — Weighting factor applied to the current data sample
- $\left(1 - \frac{1}{w_{N,\lambda}}\right) x\_rms_{N-1,\lambda}$ — Effect of the previous data on the RMS

For the first sample, where $N = 1$, the algorithm chooses $w_{N,\lambda}$ = 1. For the next sample, the weighting factor is updated and used to compute the RMS, as per the recursive equation. As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the current RMS than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the previous samples are given an equal weight.

Here is an example of computing the moving RMS using the exponential weighting method. The forgetting factor is 0.9.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Blocks
Median Filter | Moving Average | Moving Maximum | Moving Minimum | Moving Standard Deviation | Moving Variance | RMS

System Objects
dsp.MovingRMS | dsp.RMS | dsp.MovingAverage | dsp.MovingMaximum | dsp.MovingMinimum | dsp.MovingStandardDeviation | dsp.MovingVariance | dsp.MedianFilter

Topics
"What Are Moving Statistics?"
"Sliding Window Method and Exponential Weighting Method"
"Signal Statistics"
"Energy Detection in the Time Domain"

Introduced in R2016b

Moving Variance

Moving variance
Library: DSP System Toolbox / Statistics

Description

The Moving Variance block computes the moving variance of the input signal along each channel independently over time. The block uses either the sliding window method or the exponential weighting method to compute the moving variance. In the sliding window method, a window of specified length moves over the data sample by sample, and the block computes the variance over the data in the window. In the exponential weighting method, the block subtracts each sample of the data from the average, squares the difference, and multiplies the squared result by a weighting factor. The object then computes the variance by adding all the weighted data. For more details on these methods, see “Algorithms” on page 2-1268.

Ports

Input

`Port_1` — Data input
column vector | row vector | matrix

Data over which the block computes the moving variance. The block accepts real-valued or complex-valued multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$, and $n \geq 1$. The block also accepts variable-size inputs. During simulation, you can change the size of each input channel. However, the number of channels cannot change.

Data Types: `single` | `double`
Complex Number Support: Yes

Output

`Port_1` — Moving variance output

column vector | row vector | matrix

The size of the moving variance output matches the size of the input. The block uses either the sliding window method or the exponential weighting method to compute the moving variance. For more details, see "Algorithms" on page 2-1268.

Data Types: `single` | `double`
Complex Number Support: Yes

Parameters

`Method` — Moving variance method

`Sliding window` (default) | `Exponential weighting`

- `Sliding window` — A window of length **Window length** moves over the input data along each channel. For every sample the window moves by, the block computes the variance over the data in the window.
- `Exponential weighting` — The block subtracts each sample of the data from the average, squares the difference, and multiplies the squared result by a weighting factor. The block then computes the variance by adding all the weighted data. The magnitude of the weighting factors decreases exponentially as the age of the data increases, never reaching zero.

For more details on these methods, see "Algorithms" on page 2-1268.

`Specify window length` — Flag to specify window length

on (default) | off

When you select this check box, the length of the sliding window is equal to the value you specify in **Window length**. When you clear this check box, the length of the sliding window is infinite. In this mode, the block computes the variance of the current sample with respect to all the previous samples in the channel.

`Window length` — Length of the sliding window

4 (default) | positive scalar integer

Window length specifies the length of the sliding window. This parameter appears when you select the **Specify window length** check box.

`Forgetting factor` — Exponential weighting factor
0.9 (default) | positive real scalar in the range (0,1]

This parameter applies when you set **Method** to `Exponential weighting`. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the previous samples are given an equal weight.

This parameter is tunable. You can change its value even during the simulation.

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

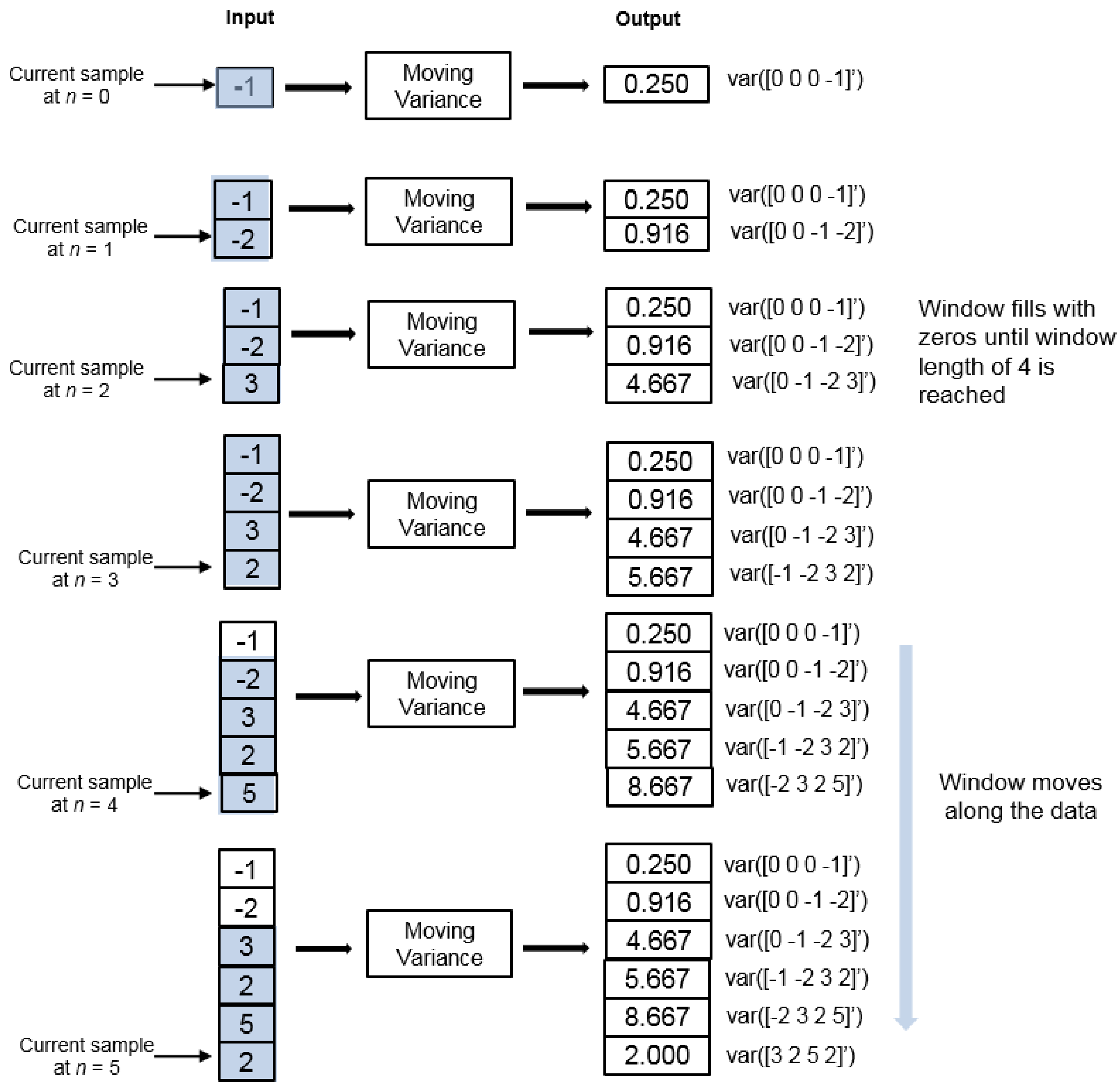
Algorithms

Sliding Window Method

In the sliding window method, the output at the current sample is the variance of the current sample with respect to the data in the window. To compute the first *Len* - 1 outputs, when the window does not have enough data yet, the algorithm fills the window with zeros. As an example, to compute the variance when the second input sample comes in, the algorithm fills the window with *Len* - 2 zeros. *Len* is the length of the window. The data vector, x, is then the two data samples followed by *Len* - 2 zeros.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving variance of the current sample with respect to all previous samples in the channel.

Consider an example of computing the moving variance of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.


Input
Output
Current sample at n = 0
-1
Moving Variance
0.250
var([0 0 0 -1]')
Current sample at n = 1
-1
-2
0.916
var([0 0 -1 -2]')
Current sample at n = 2
3
4.667
var([0 -1 -2 3]')
Window fills with zeros until window length of 4 is reached
Current sample at n = 3
2
5.667
var([-1 -2 3 2]')
Current sample at n = 4
5
8.667
var([-2 3 2 5]')
Window moves along the data
Current sample at n = 5
2.000
var([3 2 5 2]')

Exponential Weighting Method

In the exponential weighting method, the moving variance is computed recursively using these formulas:

$$s^2_{N,\lambda} = \frac{1}{v_{N,\lambda}} \sum_{k=1}^{N} \lambda^{N-k} \left[x_k - \bar{x}_{N,\lambda} \right]^2,$$

$$v_{N,\lambda} = \frac{2\lambda(1-\lambda^{N-1})}{(1-\lambda)(1+\lambda)}$$

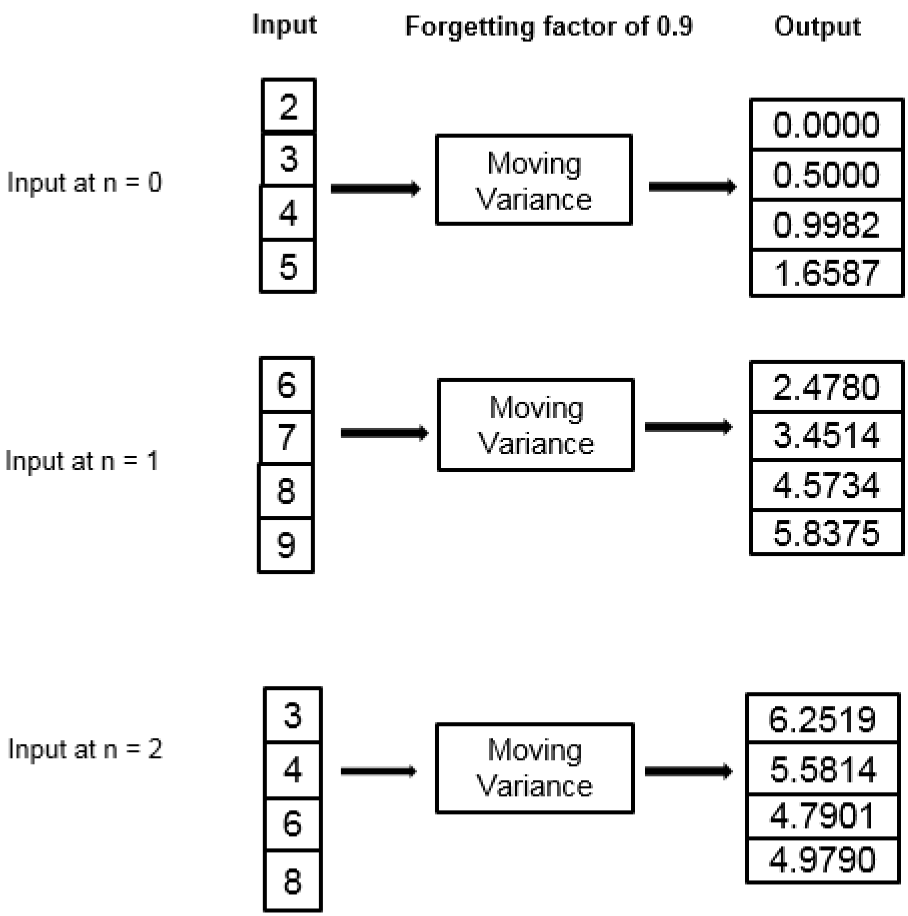
To compute the moving variance, the algorithm implements these equations recursively.

- $s^2_{N,\lambda}$ — Moving variance of the current data sample with respect to the rest of the data in the channel.
- $\bar{x}_{N,\lambda}$ — Moving average at the current sample. For details on computing the moving average, see dsp.MovingAverage.
- $\left[x_k - \bar{x}_{N,\lambda} \right]^2$ — Difference between each data sample and the average of the data, squared.
- $\sum_{k=1}^{N} \lambda^{N-k} \left[x_k - \bar{x}_{N,\lambda} \right]^2$ — Difference between each data sample and the average of the data, squared and multiplied with the forgetting factor. All the squared terms are added.
- $\frac{1}{v_{N,\lambda}}$ — Weighting factor applied to the sum.
- λ — Forgetting factor you can specify through the `ForgettingFactor` property.

As the age of the data increases, the magnitude of the weighting factor decreases exponentially, and never reaches zero. In other words, the recent data has more influence on the current variance, than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the past samples are given an equal weight.

Consider an example of computing the moving variance using the exponential weighting method. The forgetting factor is 0.9.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Blocks

Median Filter | Moving Average | Moving Maximum | Moving Minimum | Moving RMS | Moving Standard Deviation | Variance

System Objects

dsp.MovingVariance | dsp.Variance | dsp.MovingAverage | dsp.MovingMaximum | dsp.MovingMinimum | dsp.MovingStandardDeviation | dsp.MovingRMS | dsp.MedianFilter

Topics

"What Are Moving Statistics?"
"Sliding Window Method and Exponential Weighting Method"
"Signal Statistics"

Introduced in R2016b

Moving Standard Deviation

Moving standard deviation
Library: DSP System Toolbox / Statistics

Description

The Moving Standard Deviation block computes the moving standard deviation of the input signal along each channel independently over time. The block uses either the sliding window method or the exponential weighting method to compute the moving standard deviation. In the sliding window method, a window of specified length moves over the data sample by sample, and the block computes the standard deviation over the data in the window. In the exponential weighting method, the block computes the exponentially weighted moving variance and takes the square root. For more details on these methods, see “Algorithms” on page 2-1276.

Ports

Input

`Port_1` — Data input

column vector | row vector | matrix

Data over which the block computes the moving standard deviation. The block accepts real-valued or complex-valued multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$ and $n \geq 1$. The block also accepts variable-size inputs. During simulation, you can change the size of each input channel. However, the number of channels cannot change.

Data Types: `single` | `double`
Complex Number Support: Yes

Output

`Port_1` — Moving standard deviation output

column vector | row vector | matrix

The size of the moving standard deviation output matches the size of the input. The block uses either the sliding window method or the exponential weighting method to compute the moving standard deviation. For more details, see “Algorithms” on page 2-1276.

Data Types: `single` | `double`
Complex Number Support: Yes

Parameters

`Method` — Moving standard deviation method

`Sliding window` (default) | `Exponential weighting`

- `Sliding window` — A window of length **Window length** moves over the input data along each channel. For every sample the window moves by, the block computes the standard deviation over the data in the window.
- `Exponential weighting` — The block computes the exponentially weighted moving standard deviation and takes the square root.

For more details on these methods, see “Algorithms” on page 2-1276.

`Specify window length` — Flag to specify window length

on (default) | off

When you select this check box, the length of the sliding window is equal to the value you specify in **Window length**. When you clear this check box, the length of the sliding window is infinite. In this mode, the block computes the standard deviation of the current sample with respect to all the previous samples in the channel.

`Window length` — Length of the sliding window

4 (default) | positive scalar integer

Window length specifies the length of the sliding window. This parameter appears when you select the **Specify window length** check box.

`Forgetting factor` — Exponential weighting factor

0.9 (default) | positive real scalar in the range (0,1]

This parameter applies when you set **Method** to `Exponential weighting`. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the previous samples are given an equal weight.

This parameter is tunable. You can change its value even during the simulation.

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

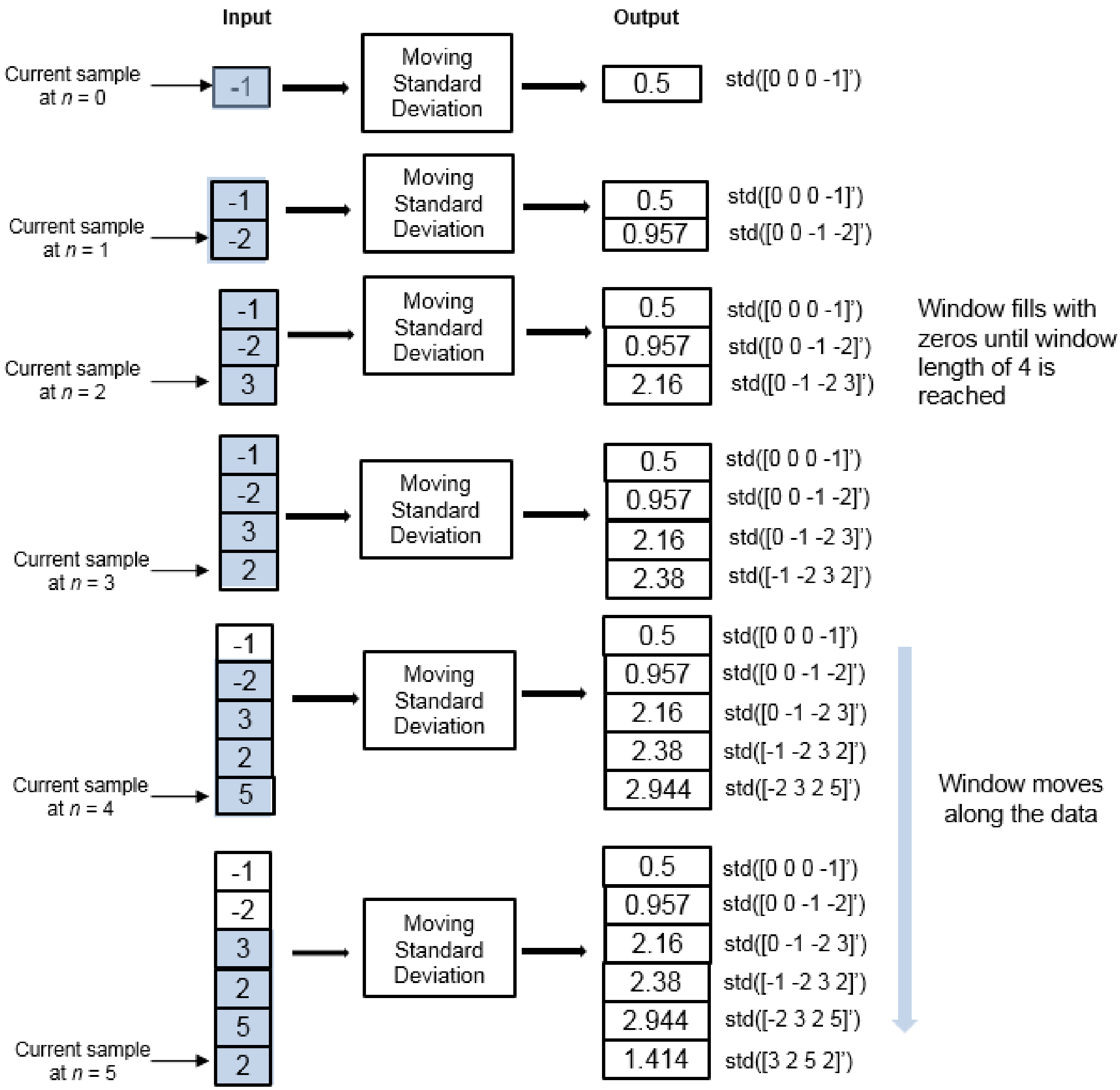
Algorithms

Sliding Window Method

In the sliding window method, the output at the current sample is the standard deviation of the current sample with respect to the data in the window. To compute the first *Len* - 1 outputs, when the window does not have enough data yet, the algorithm fills the window with zeros. As an example, to compute the standard deviation when the second input sample comes in, the algorithm fills the window with *Len* - 2 zeros. *Len* is the length of the window. The data vector, *x*, is then the two data samples followed by *Len* - 2 zeros.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving standard deviation of the current sample with respect to all the previous samples in the channel.

Consider an example of computing the moving standard deviation of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.


Input
Output
Current sample at $n = 0$
-1
Moving Standard Deviation
0.5
std([0 0 0 -1]')
Current sample at $n = 1$
-1
-2
0.5
0.957
std([0 0 0 -1]')
std([0 0 -1 -2]')
Current sample at $n = 2$
-1
-2
3
0.5
0.957
2.16
std([0 0 0 -1]')
std([0 0 -1 -2]')
std([0 -1 -2 3]')
Window fills with zeros until window length of 4 is reached
Current sample at $n = 3$
-1
-2
3
2
0.5
0.957
2.16
2.38
std([0 0 0 -1]')
std([0 0 -1 -2]')
std([0 -1 -2 3]')
std([-1 -2 3 2]')
Current sample at $n = 4$
-1
-2
3
2
5
0.5
0.957
2.16
2.38
2.944
std([0 0 0 -1]')
std([0 0 -1 -2]')
std([0 -1 -2 3]')
std([-1 -2 3 2]')
std([-2 3 2 5]')
Window moves along the data
Current sample at $n = 5$
-1
-2
3
2
5
2
0.5
0.957
2.16
2.38
2.944
1.414
std([0 0 0 -1]')
std([0 0 -1 -2]')
std([0 -1 -2 3]')
std([-1 -2 3 2]')
std([-2 3 2 5]')
std([3 2 5 2]')

Exponential Weighting Method

In the exponential weighting method, the moving standard deviation is computed recursively using these formulas:

$$s_{N,\lambda} = \sqrt{\frac{1}{v_{N,\lambda}} \sum_{k=1}^{N} \lambda^{N-k} \left[x_k - \bar{x}_{N,\lambda} \right]^2},$$

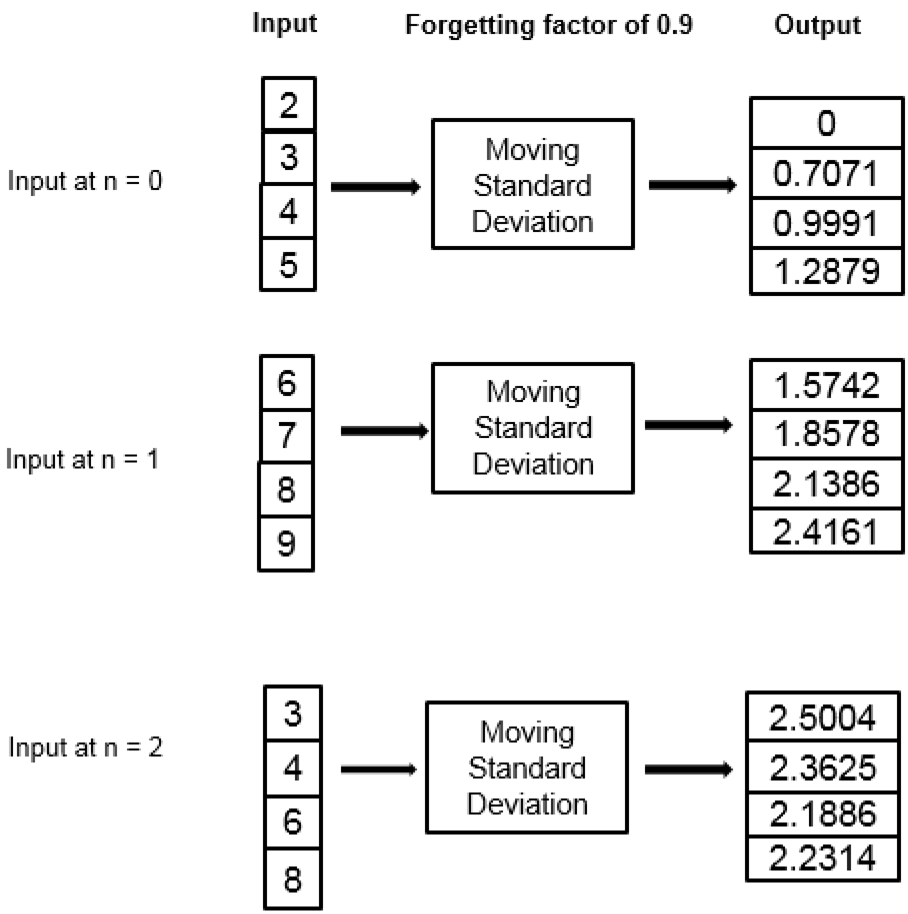
$$v_{N,\lambda} = \frac{2\lambda(1-\lambda^{N-1})}{(1-\lambda)(1+\lambda)}$$

- $s_{N,\lambda}$ — Moving standard deviation of the current data sample with respect to the rest of the data.
- $\left[x_k - \bar{x}_{N,\lambda} \right]^2$ — Difference between each data sample and the average of the data, squared.
- $\sum_{k=1}^{N} \lambda^{N-k} \left[x_k - \bar{x}_{N,\lambda} \right]^2$ — Difference between each data sample and the average of the data, squared and multiplied with the forgetting factor. All the squared terms are added.
- $\frac{1}{v_{N,\lambda}}$ — Weighting factor applied to the sum.
- λ — Forgetting factor.

As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the current standard deviation than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All previous samples are given an equal weight.

Consider an example of computing the moving standard deviation using the exponential weighting method. The forgetting factor is 0.9.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Blocks

`Median Filter` | `Moving Average` | `Moving Maximum` | `Moving Minimum` | `Moving RMS` | `Moving Variance` | `Standard Deviation`

System Objects

dsp.MovingStandardDeviation | dsp.StandardDeviation | dsp.MovingAverage | dsp.MovingMaximum | dsp.MovingMinimum | dsp.MovingVariance | dsp.MovingRMS | dsp.MedianFilter

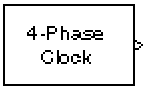
Topics

"What Are Moving Statistics?"
"Sliding Window Method and Exponential Weighting Method"
"Signal Statistics"

Introduced in R2016b

Multiphase Clock

Generate multiple binary clock signals



Library

- Sources

  `dspsrcs4`
- Signal Management / Switches and Counters

  `dspswit3`

Description

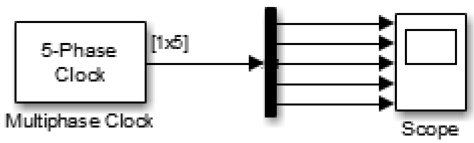
The Multiphase Clock block generates a 1-by-*N* vector of clock signals, where you specify the integer *N* in the **Number of phases** parameter. Each of the *N* phases has the same frequency, *f*, specified in hertz by the **Clock frequency** parameter.

The clock signal indexed by the **Starting phase** parameter is the first to become active, at *t*=0. The other signals in the output vector become active in turn, each one lagging the preceding signal's activation by 1/(*N**f*) seconds, the phase interval. The period of the output is therefore 1/(*N**f*) seconds.

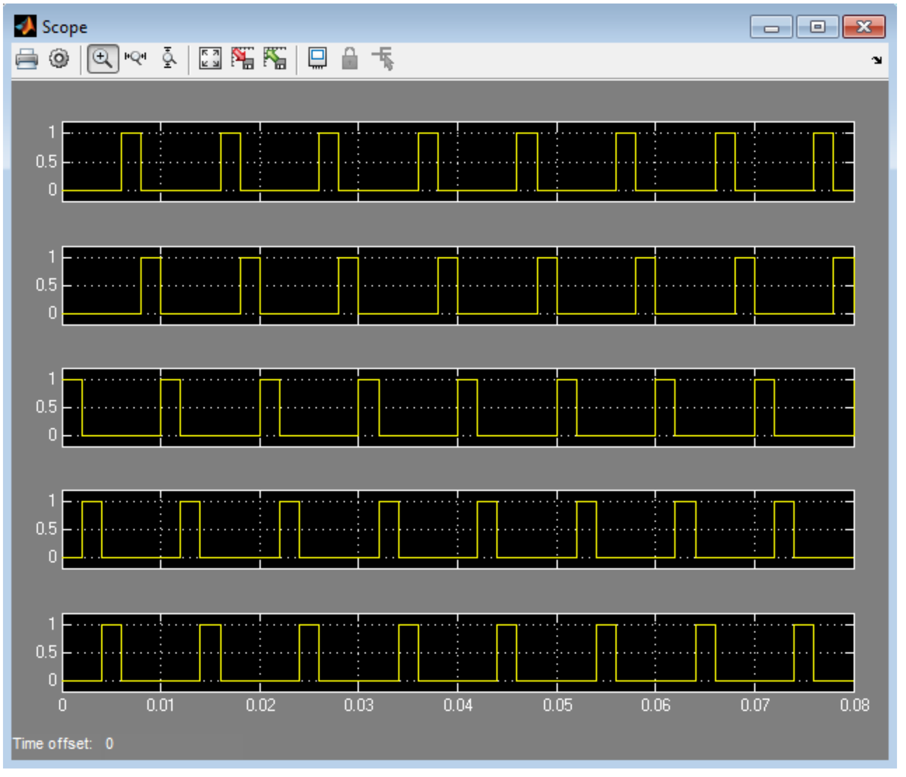
The active level can be either high (`1`) or low (`0`), as specified by the **Active level (polarity)** parameter. The duration of the active level, *D*, is set by the **Number of phase intervals over which the clock is active**. This value, which can be an integer value between 1 and *N*-1, specifies the number of phase intervals that each signal should remain in the active state after becoming active. The active duty cycle of the signal is *D*/*N*.

Examples

In the following ex_multiphaseclock_ref model, the Multiphase Clock block generates a 100 Hz five-phase output in which the third signal is first to become active. The block uses a high active level with a duration of one interval.



The Scope window below shows the Multiphase Clock block's output. Note that the first active level appears at *t*=0 on `y(3)`, the second active level appears at *t*=0.002 on `y(4)`, the third active level appears at *t*=0.004 on `y(5)`, the fourth active level appears at *t*=0.006 on `y(1)`, and the fifth active level appears at *t*=0.008 on `y(2)`. Each signal becomes active 1/(5*100) seconds after the previous signal.



To experiment further, try changing the **Number of phase intervals over which clock is active** setting to **3** so that the active-level duration is three phase intervals (60% duty cycle).

Parameters

Clock frequency

The frequency of all output clock signals.

Number of phases

The number of different phases, *N*, in the output vector.

Starting phase

The vector index of the output signal to first become active.

Number of phase intervals over which clock is active

The duration of the active level for every output signal.

Active level

The active level, `High (1)` or `Low (0)`.

Output data type

The output data type.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean

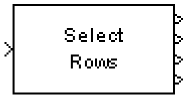
See Also

Clock	Simulink
Counter	DSP System Toolbox
Pulse Generator	Simulink
Event-Count Comparator	DSP System Toolbox

Introduced before R2006a

Multiport Selector

Distribute arbitrary subsets of input rows or columns to multiple output ports



Library

Signal Management / Indexing

`dspindex`

Description

The Multiport Selector block extracts multiple subsets of rows or columns from M-by-N input matrix u, and propagates each new submatrix to a distinct output port. The block treats an unoriented length-M vector input as an M-by-1 matrix.

The **Indices to output** parameter is a cell array whose kth cell contains a one-dimensional indexing expression specifying the subset of input rows or columns to be propagated to the kth output port. The total number of cells in the array determines the number of output ports on the block.

When you set the **Select** parameter to `Rows`, the block uses the one-dimensional indices you specify to select matrix rows, and all elements on the chosen rows are included. When you set the **Select** parameter to `Columns`, the block uses the one-dimensional indices you specify to select matrix columns, and all elements on the chosen columns are included. A given input row or column can appear any number of times in any of the outputs, or not at all.

When an index references a nonexistent row or column of the input, the block reacts with the action you specify using the **Invalid index** parameter.

Examples

Example 1

Consider the following **Indices to output** cell array:

```
{4,[1:2 5],[7;8],10:-1:6}
```

This is a four-cell array, which requires the block to generate four independent outputs (each at a distinct port). The table below shows the dimensions of these outputs when **Select** = `Rows` and the input dimension is M-by-N.

Cell	Expression	Description	Output Size
1	`4`	Row 4 of input	1-by-N
2	`[1:2 5]`	Rows 1, 2, and 5 of input	3-by-N
3	`[7;8]`	Rows 7 and 8 of input	2-by-N
4	`10:-1:6`	Rows 10, 9, 8, 7, and 6 of input	5-by-N

Parameters

Select

Specify the dimension of the input to select, `Rows` or `Columns`.

Indices to output

A cell array specifying the row- or column-subsets to propagate to each of the output ports. The number of cells in the array determines the number of output ports on the block.

Invalid index

Specify how the block handles an invalid index value. You can select one of the following options:

- `Clip index` — Clip the index to the nearest valid value, and do not issue an alert.

  For example, if the block receives a `64`-by-`4` input and the **Select** parameter is set to `Rows`, the block clips an index of `72` to `64`. For the same input, if the **Select** parameter is set to `Columns`, the block clips an index of `72` to `4`. In both cases, the block clips an index of `-2` to `1`.

- `Clip and warn` — Clip the index to the nearest valid value and display a warning message at the MATLAB command line.
- `Generate error` — Display an error dialog box and terminate the simulation.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Multiport Selector.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated
Outputs	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated

See Also

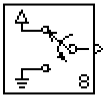
Permute Matrix	DSP System Toolbox

Selector	Simulink
Submatrix	DSP System Toolbox
Variable Selector	DSP System Toolbox

Introduced before R2006a

N-Sample Enable

Output ones or zeros for specified number of sample times



Library

- Sources

  `dspsrcs4`
- Signal Management / Switches and Counters

  `dspswit3`

Description

The N-Sample Enable block outputs the inactive value (`0` or `1`, whichever is not selected in the **Active level** parameter) during the first *N* sample times, where *N* is the **Trigger count** value. Beginning with output sample *N*+1, the block outputs the active value (`1` or `0`, whichever you select in the **Active level** parameter) until a reset event occurs or the simulation terminates.

The output of the block is always a scalar.

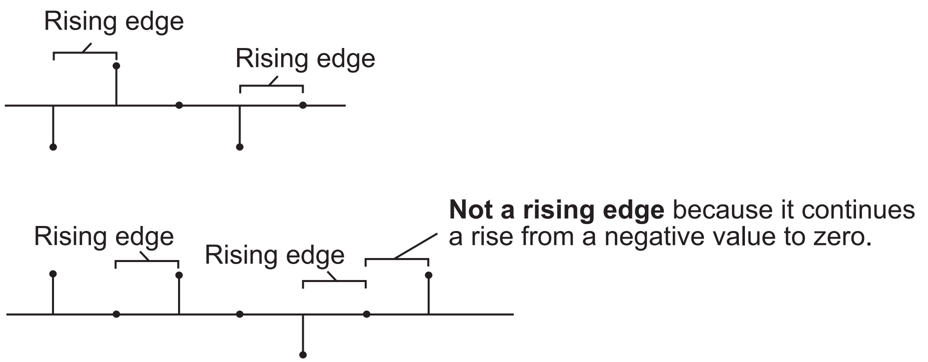
The **Reset input** check box enables the Rst input port. At any time during the count, a trigger event at the input port resets the counter to its initial state. This block supports triggered subsystems when you select the **Reset input** check box.

You specify the triggering event in the **Trigger type** pop-up menu:

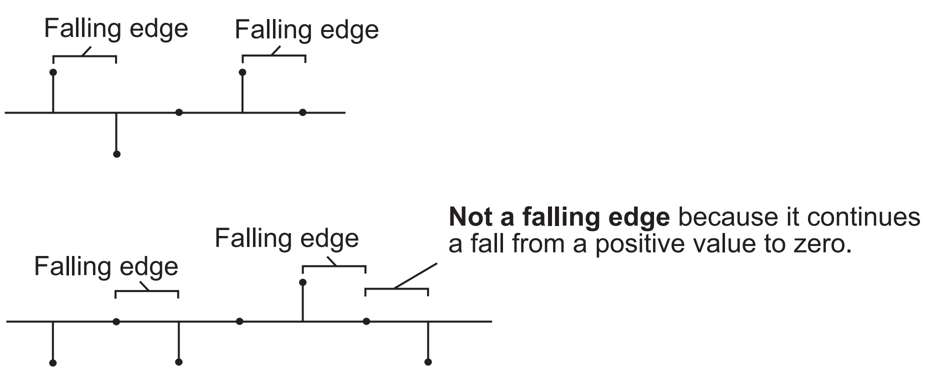
- `Rising edge` — Triggers a reset operation when the `Rst` input does one of the following:

  - Rises from a negative value to a positive value or zero

- Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- `Falling edge` — Triggers a reset operation when the `Rst` input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero.

Parameters

Trigger count

Specify the number of samples for which the block outputs the active value. Tunable (Simulink).

Active level

Specify the value to output after the first *N* sample times, 0 or 1. Tunable (Simulink).

Reset input

Select to enable the Rst input port.

Trigger type

Select type of event that triggers a reset when the Rst port is enabled.

Sample time

Specify the sample period, T_s, for the block's counter. The block switches from the active value to the inactive value at $t=T_s*(N+1)$.

Output data type

Select the output data type.

Supported Data Types

- Double-precision floating point
- Boolean — The block accepts Boolean inputs to the Rst port, which is enabled when you select the **Reset input** parameter.

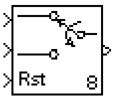
See Also

Counter	DSP System Toolbox
N-Sample Switch	DSP System Toolbox

Introduced before R2006a

N-Sample Switch

Switch between two inputs after specified number of sample periods



Library

Signal Management / Switches and Counters

`dspswit3`

Description

The N-Sample Switch block outputs the signal connected to the top input port during the first *N* sample times after the simulation begins or the block is reset, where you specify *N* in the **Switch count** parameter. Beginning with output sample *N*+1, the block outputs the signal connected to the bottom input until the next reset event or the end of the simulation.

You specify the sample period of the output in the **Sample time** parameter (that is, the output sample period is not inherited from the sample period of either input). The block applies a zero-order hold at the input ports, so the value the block reads from a given port between input sample times is the value of the most recent input to that port.

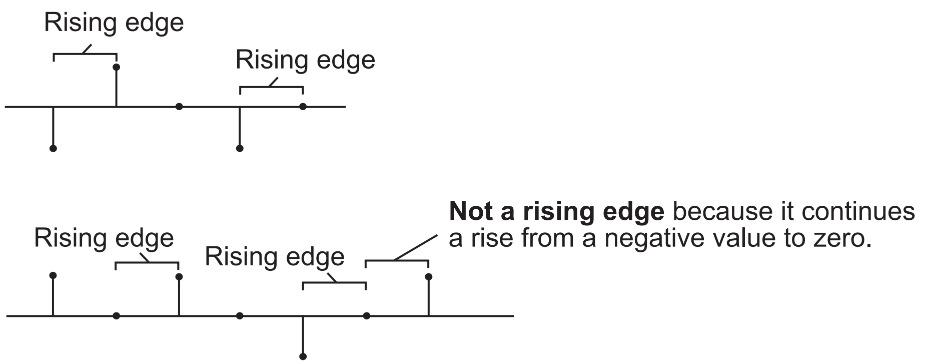
Both inputs must have the same dimension, except in the following two cases:

- When one input is a scalar, the block expands the scalar input to match the size of the other input.
- When one input is an unoriented vector and the other input is a row or column vector with the same number of elements, the block reshapes the unoriented vector to match the dimension of the other input.

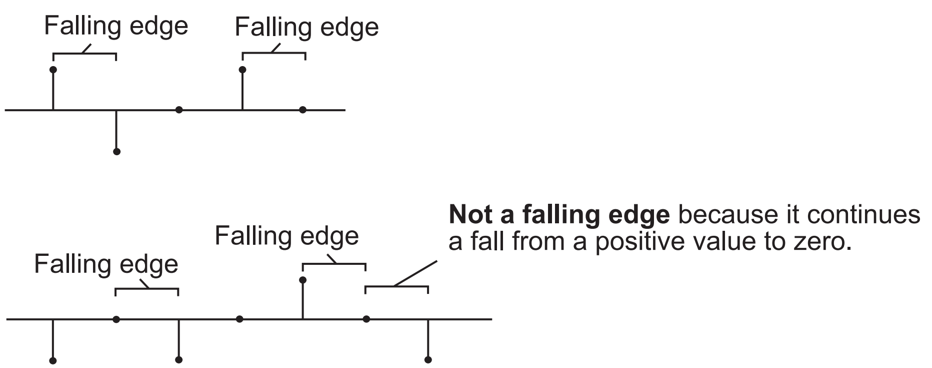
The **Reset input** check box enables the Rst input port. At any time during the count, a trigger event at the Rst port resets the counter to zero. The reset sample time must be a positive integer multiple of the input sample time. This block supports triggered subsystems when you select the **Reset input** check box.

You specify the triggering event in the **Trigger type** pop-up menu, and can be one of the following:

- `Rising edge` — Triggers a reset operation when the `Rst` input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- `Falling edge` — Triggers a reset operation when the `Rst` input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero.

Parameters

Switch count

The number of sample periods, N, for which the output is connected to the top input before switching to the bottom input. Tunable (Simulink).

Reset input

Enables the Rst input port when selected. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input.

Trigger type

The type of event at the Rst port that resets the block's counter. This parameter is enabled when you select **Reset input**. Tunable (Simulink).

Sample time

The sample period, T_s, for the block's counter. The block switches inputs at $t=T_s*(N+1)$.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- Boolean — The block accepts Boolean inputs to the Rst port, which is enabled when you set the **Reset input** parameter.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

See Also

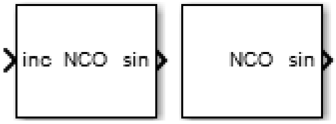
Counter	DSP System Toolbox

N-Sample Enable	DSP System Toolbox

Introduced before R2006a

NCO

Generate real or complex sinusoidal signals



Library

Signal Operations

`dspsigops`

Sources

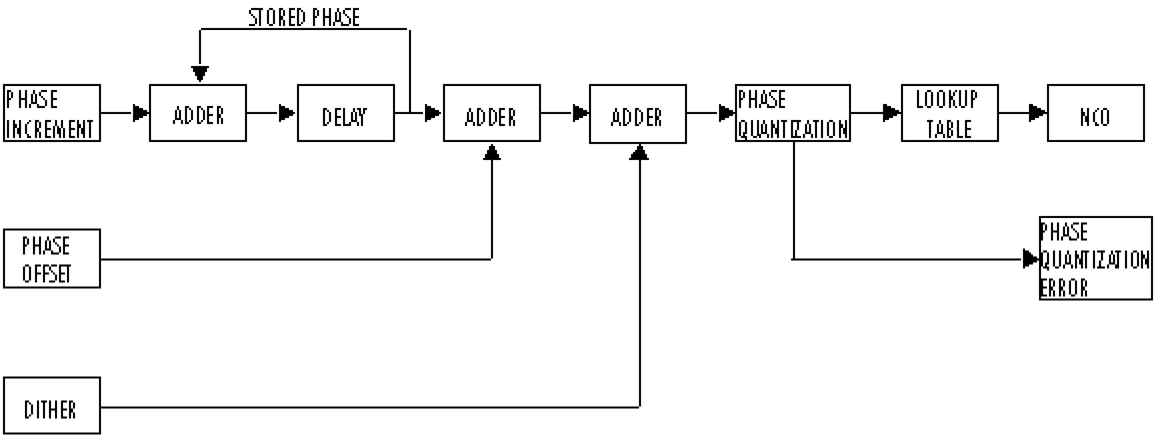
`dspsrcs4`

Description

The block appears in `dspsigops` with **Phase increment Source** parameter set to `Input port`.

In addition, the block also appears in `dspsrsc4` library with **Phase increment Source** parameter set to `Specify via dialog`.

The NCO block generates a multichannel real or complex sinusoidal signal, with independent frequency and phase in each output channel. The amplitude of the created signal is always 1. The block implements the algorithm as shown in the following diagram:



The implementation of a numerically controlled oscillator (NCO) has two distinct parts. First, a phase accumulator accumulates the phase increment and adds in the phase offset. In this stage, an optional internal dither signal can also be added. The NCO output is then calculated by quantizing the results of the phase accumulator section and using them to select values from a lookup table. Since the lookup table contains a finite set of entries, in its normal mode of operation, the NCO block allows the adder's numeric values to overflow and wrap around. The Fixed-Point infrastructure then causes overflow warnings to appear on the command line. This overflow is of no consequence.

Given a desired output frequency F_0, calculate the value of the **Phase increment** block parameter with

$$phase\ increment = \left(\frac{F_0 \cdot 2^N}{F_s}\right)$$

where N is the accumulator word length and

$$F_s = \frac{1}{T_s} = \frac{1}{sample\ time}$$

The frequency resolution of an NCO is defined by

$$\Delta f = \frac{1}{T_s \cdot 2^N} \text{Hz}$$

Given a desired phase offset (in radians), calculate the **Phase offset** block parameter with

$$phase\ offset = \frac{2^N \cdot desired\ phase\ offset}{2\pi}$$

The spurious free dynamic range (SFDR) is estimated as follows for a lookup table with 2^P entries, where P is the number of quantized accumulator bits:

$$\begin{aligned} SFDR &= (6P) \text{ dB} && \text{without dither} \\ SFDR &= (6P+12) \text{ dB} && \text{with dither} \end{aligned}$$

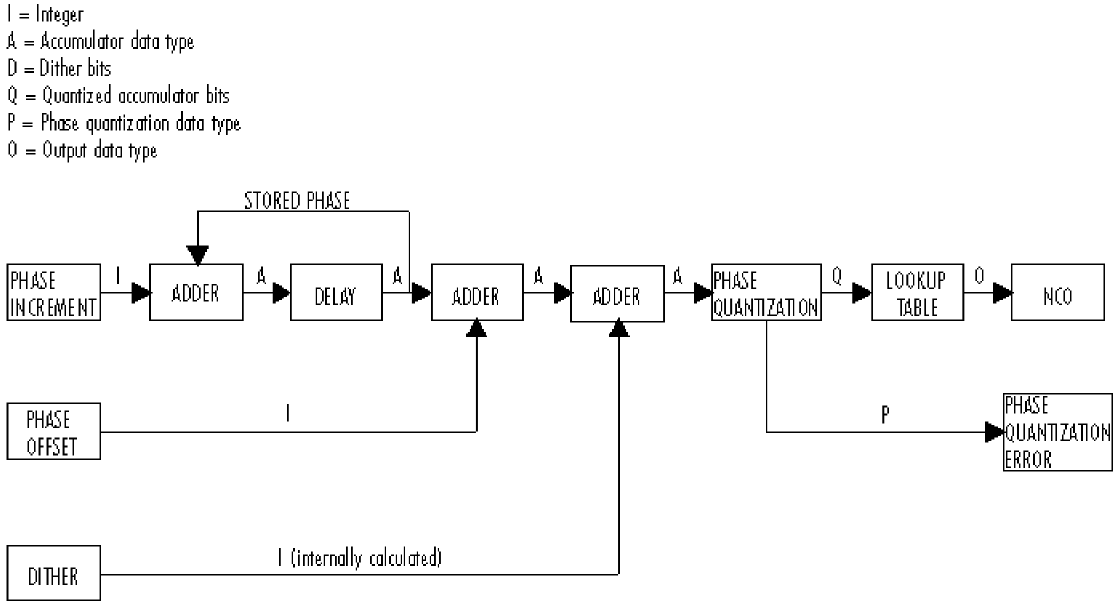
This block uses a quarter-wave lookup table technique that stores table values from 0 to π/2. The block calculates other values on demand using the accumulator data type, then casts them into the output data type. This can lead to quantization effects at the range limits of a given data type. For example, consider a case where you would expect the value of the sine wave to be –1 at π. Because the lookup table value at that point must be calculated, the block might not yield exactly –1, depending on the precision of the accumulator and output data types.

The NCO block supports real inputs only. All outputs are real except for the output signal in `Complex exponential` mode.

To produce a multichannel output, specify a vector quantity for the **Phase increment** and **Phase offset** parameters. Both parameters must have the same length, which defines the number of output channels. Each element of each vector is applied to a different output channel.

Fixed-Point Data Types

The following diagram shows the data types used within the NCO block.



- You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 2-1303 below.

Note: The lookup table for this block is constructed from double-precision floating-point values. Thus, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length of the **Output** data type to values greater than 53 bits does not improve the precision of your output.

- The phase increment and phase offset inputs must be integers or fixed-point data types with zero fraction length.
- You specify the number of quantized accumulator bits in the **Number of quantized accumulator bits** parameter.
- The phase quantization error word length is equal to the accumulator word length minus the number of quantized accumulator bits, and the fraction length is zero.

Examples

The NCO block is used in the GSM Digital Down Converter product example. Open this example by typing `dspddc` at the MATLAB command line.

You can also try the following example. Design an NCO source with the following specifications:

- Desired output frequency $F_0 = 510$ Hz
- Frequency resolution $\Delta f = 0.05$ Hz
- Spurious free dynamic range $SFDR \geq 90$ dB
- Sample period $T_s = 1/8000$ s
- Desired phase offset $\pi / 2$

1 Calculate the number of required accumulator bits from the equation for frequency resolution:

$$\Delta f = \frac{1}{T_s \cdot 2^N} \text{Hz}$$

$$0.05 = \frac{1}{\frac{1}{8000} \cdot 2^N} \text{Hz}$$

$$N = 18$$

Note that N must be an integer value. The value of N is rounded up to the nearest integer; 18 accumulator bits are needed to accommodate the value of the frequency resolution.

2 Using this best value of N, calculate the frequency resolution that will be achieved by the NCO block:

$$\Delta f = \frac{1}{T_s \cdot 2^N}\ \mathrm{Hz}$$

$$\Delta f = \frac{1}{\frac{1}{8000} \cdot 2^{18}}\ \mathrm{Hz}$$

$$\Delta f = 0.0305$$

3 Calculate the number of quantized accumulator bits from the equation for spurious free dynamic range and the fact that for a lookup table with 2^P entries, P is the number of quantized accumulator bits:

$$SFDR = (6P + 12)\,\mathrm{dB}$$

$$96\ \mathrm{dB} = (6P + 12)\ \mathrm{dB}$$

$$P = 14$$

4 Select the number of dither bits. In general, a good choice for the number of dither bits is the accumulator word length minus the number of quantized accumulator bits; in this case 4.

5 Calculate the phase increment:

$$phase\ increment = \mathrm{round}(\frac{F_0 \cdot 2^N}{F_s})$$

$$phase\ increment = \mathrm{round}(\frac{510 \cdot 2^{18}}{8000})$$

$$phase\ increment = 16712$$

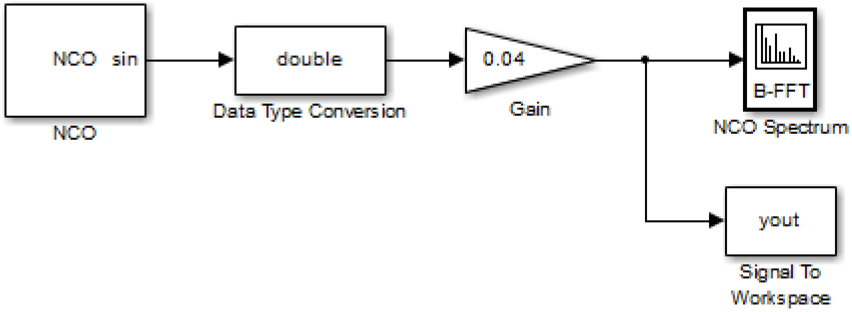
6 Calculate the phase offset:

$$phase\ offset = \frac{2^{accumulator\ word\ length} \cdot desired\ phase\ offset}{2\pi}$$

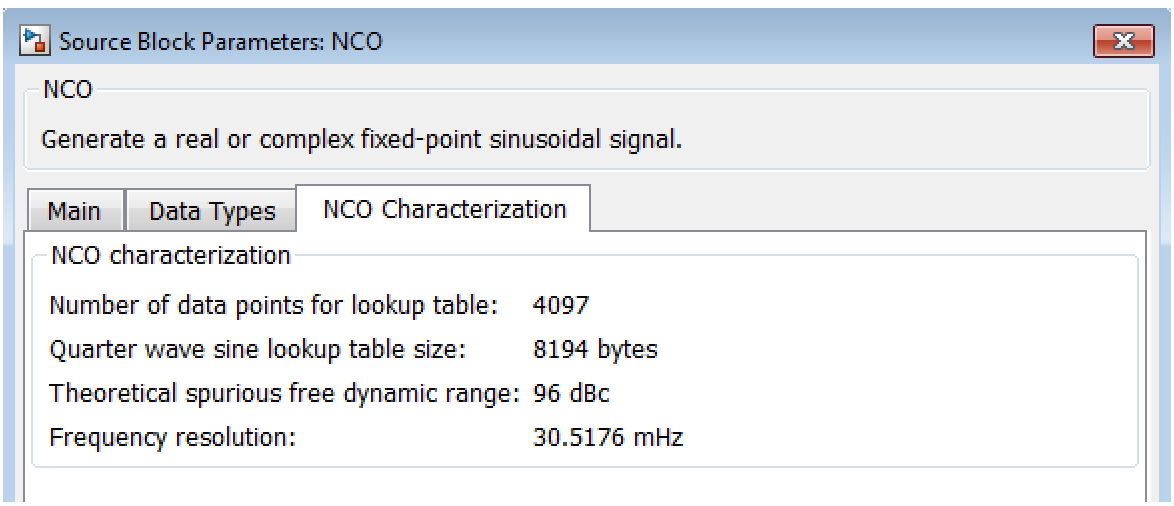
$$phase\ offset = \frac{2^{18} \cdot \frac{\pi}{2}}{2\pi}$$

$$phase\ offset = 65536$$

7 Type `ex_nco_example` at the MATLAB command line to open the following model:



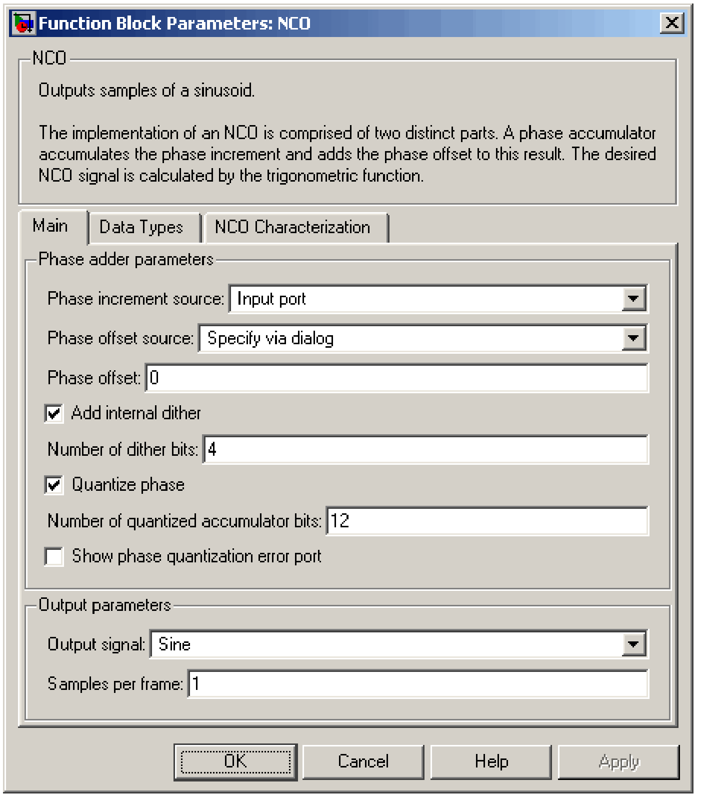
The NCO block in the model is populated with the specifications and quantities you just calculated. The output word length and fraction length depend on the constraints of your hardware; this example uses a word length of 16 and a fraction length of 14. You can verify that the specifications of this problem have been met by looking at the **NCO Characterization** pane of the NCO block.



8 Experiment with the model to observe the effects on the output shown on the Spectrum Analyzer. For example, try turning dithering on and off, and try changing the number of dither bits.

Dialog Box

The **Main** pane of the NCO dialog appears as follows.



Phase increment source

Choose how you specify the phase increment. The phase increment can come from an input port or from the dialog.

- If you select `Input port`, the inc port appears on the block icon.
- If you select `Specify via dialog`, the **Phase increment** parameter appears.

Phase increment

Specify the phase increment. Only integer data types, including fixed-point data types with zero fraction length, are allowed. The dimensions of the phase increment are dictated by those of the phase offset:

- When you specify the phase offset on the block dialog box, the phase increment must be a scalar or a vector with the same length as the phase offset. The block applies each element of the vector to a different channel, and therefore the vector length defines the number of output channels.
- When you specify the phase offset via an input port, the offset port treats each column of the input as an independent channel. The phase increment length must equal the number of columns in the input to the offset port.

This parameter is visible only if you set the **Phase increment source** parameter to `Specify via dialog`.

Phase offset source

Choose how you specify the phase offset. The phase offset can come from an input port or from the dialog.

- If you select `Input port`, the offset port appears on the block icon.
- If you select `Specify via dialog`, the **Phase offset** parameter appears.

When you specify the phase offset via an input port, it can be a scalar, vector, or a full matrix. The block treats each column of the input to the offset port as an independent channel. The number of channels in the phase offset must match the number of channels in the data input. For each frame of the input, the block can apply different phase offsets to each sample and channel. Only integer data types, including fixed-point data types with zero fraction length, are allowed.

Phase offset

Specify the phase offset. When you specify the phase offset using this parameter rather than via an input port, it must be a scalar or vector with the same length as the phase increment. Scalars are expanded to a vector with the same length as the phase increment. Each element of the phase offset vector is applied to a different channel of the input, and therefore the vector length defines the number of output channels. Only integer data types, including fixed-point data types with zero fraction length, are allowed.

This parameter is visible only if `Specify via dialog` is selected for the **Phase offset source** parameter.

Add internal dither

Select to add internal dithering to the NCO algorithm. Dithering is added using the PN Sequence Generator from the Communications System Toolbox™ product.

Number of dither bits

Specify the number of dither bits.

This parameter is visible only if **Add internal dither** is selected.

Quantize phase

Select to enable quantization of the accumulated phase.

Number of quantized accumulator bits

Specify the number of quantized accumulator bits. This determines the number of entries in the lookup table. The number of quantized accumulator bits must be less than the accumulator word length.

This parameter is visible only if **Quantize phase** is selected.

Show phase quantization error port

Select to output the phase quantization error. When you select this, the Qerr port appears on the block icon.

This parameter is visible only if **Quantize phase** is selected.

Output signal

Choose whether the block should output a `Sine`, `Cosine`, `Complex exponential`, or `Sine and cosine` signals. If you select `Sine and cosine`, the two signals output on different ports.

Sample time

Specify the sample time in seconds when the block is acting as a source. When either the phase increment or phase offset come in via block input ports, the sample time is inherited and this parameter is not visible.

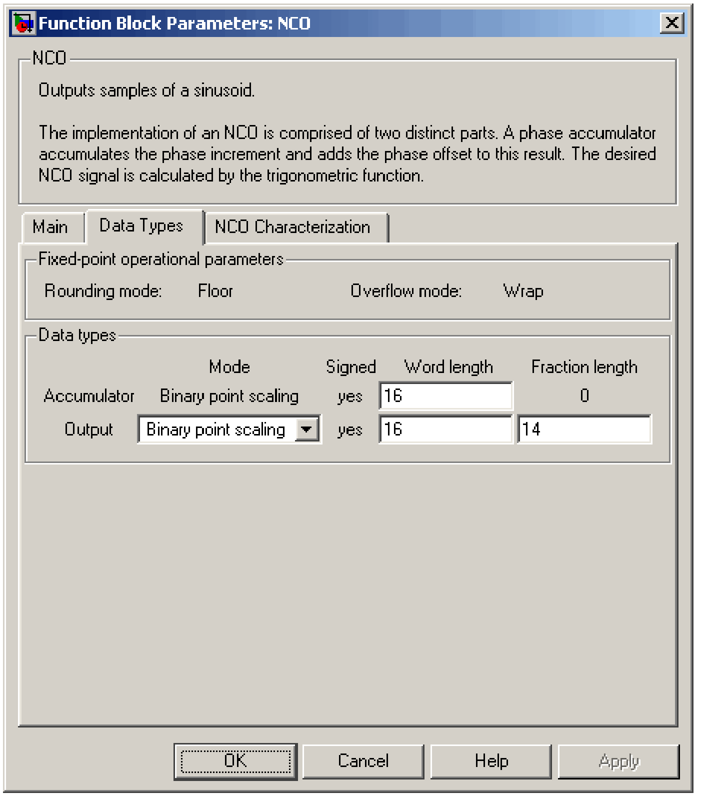
Samples per frame

Specify the number of samples per frame. When the value is greater than 1, the phase increment and phase offset can vary from channel to channel and from frame to frame, but they are constant along each channel in a given frame.

When the phase offset input port exists, it has the same frame status as any output port present. When the phase increment input port exists, it does not support frames.

This parameter is only visible if either **Phase increment source** or **Phase offset source** is set to `Specify via dialog`.

The **Data Types** pane of the NCO dialog appears as follows.



Rounding mode

The rounding mode used for this block when inputs are fixed point is always `Floor`.

Overflow mode

The overflow mode used for this block when inputs are fixed point is always `Wrap`.

Accumulator

Specify the word length of the accumulator data type. The fraction length is always zero; this is an integer data type.

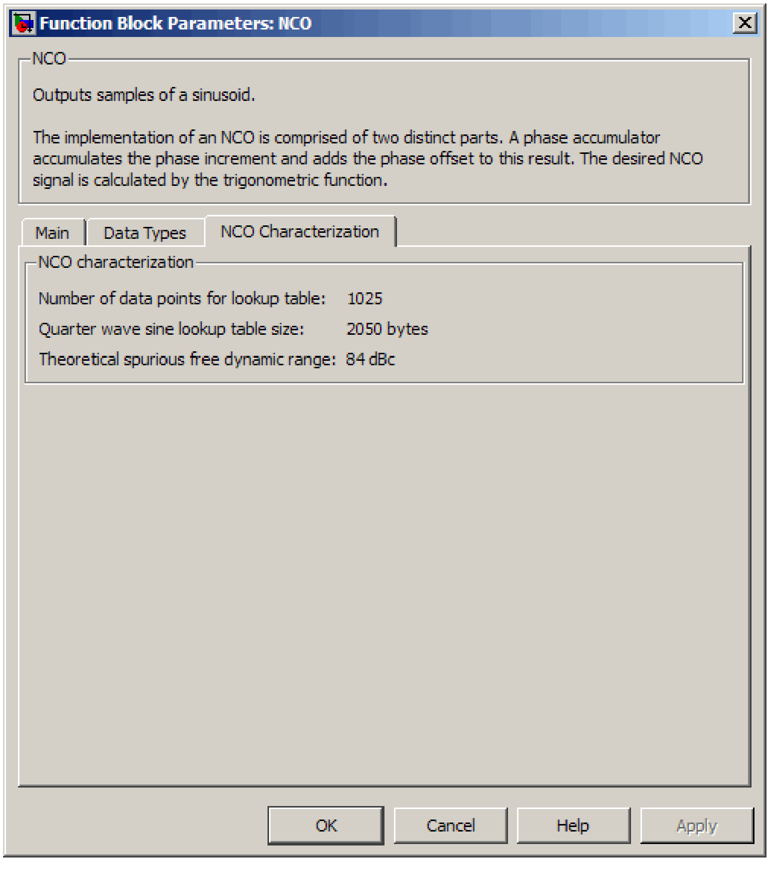
Output

Specify the output data type.

- Choose `double` or `single` for a floating-point implementation.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the output, in bits.

Note: The lookup table for this block is constructed from double-precision floating-point values. Thus, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length of the **Output** data type to values greater than 53 bits does not improve the precision of your output.

The **NCO Characterization** pane of the NCO dialog appears as follows.



The **NCO Characterization** pane does not have any parameters. Instead, it provides you with details on the NCO signal currently being implemented by the block:

- `Number of data points for lookup table` — The lookup table is implemented as a quarter-wave sine table. The number of lookup table data points is defined by

  $$2^{\text{number of quantized accumulator bits}-2}+1$$

- `Quarter wave sine lookup table size` — The quarter wave sine lookup table size is defined by

  $$\frac{(\text{number of data points for lookup table})\cdot(\text{output word length})}{8}\ \text{bytes}$$

- `Theoretical spurious free dynamic range` — The spurious free dynamic range (SFDR) is calculated as follows for a lookup table with 2^P entries:

  $$SFDR=(6P)\ \text{dB} \qquad \text{without dither}$$
  $$SFDR=(6P+12)\ \text{dB} \qquad \text{with dither}$$

Supported Data Types

Port	Supported Data Types
inc	• Fixed point with zero fraction length • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
offset	• Fixed point with zero fraction length • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
sin	• Double-precision floating point • Single-precision floating point • Fixed point
Qerr	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

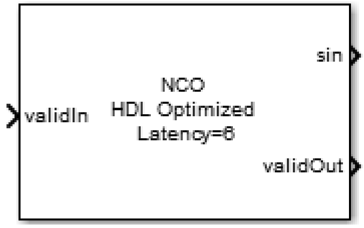
See Also

PN Sequence Generator	Communications System Toolbox
`Sine Wave`	DSP System Toolbox
`Digital Down-Converter`	DSP System Toolbox
`Digital Up-Converter`	DSP System Toolbox

Introduced before R2006a

NCO HDL Optimized

Generate real or complex sinusoidal signals—optimized for HDL code generation



Library

Signal Operations

`dspsigops`

Sources

`dspsrcs4`

Description

Note:

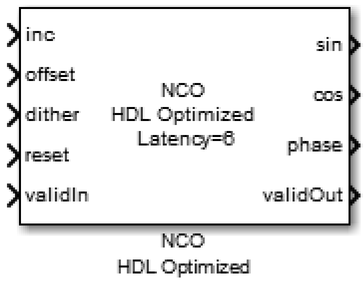
- This block appears in `dspsrsc4` library with **Phase increment source** parameter set to `Property`. The only input port is `validIn`.
- This block appears in `dspsigops` with **Phase increment source** parameter set to `Input port`. This configuration shows the optional input port `inc`.

The NCO HDL Optimized block generates real or complex sinusoidal signals, while providing hardware-friendly control signals. It uses the same phase accumulation and lookup table technology as implemented in the NCO block. It provides the following features:

- A lookup table compression option to reduce the lookup table size with less than one LSB loss in precision. See “Lookup Table Algorithm” on page 2-1315 for more detail.
- An option to synthesize the lookup table to a ROM when using HDL Coder with an FPGA target. To enable this feature, right-click the block, select **HDL Code > HDL Block Properties** and set **LUTRegisterResetType** to `none`.
- An optional input port for external dither.
- An optional reset port that triggers a reset of the phase to its initial value during the sinusoid output generation.
- An optional output port for the current NCO phase.

Signal Attributes

This icon shows all the possible ports of the NCO HDL Optimized block.



Port	Direction	Description	Data Type
`inc`	Input	Optional. Phase increment, specified as a scalar integer.	• `fixdt([],N,0)` • `int`, `uint` `double` and `single` are allowed for simulation but not for HDL code generation.
`offset`	Input	Optional. Phase offset, specified as a scalar integer.	• `fixdt([],N,0)` • `int`, `uint` `double` and `single` are allowed for simulation but not for HDL code generation.

Port	Direction	Description	Data Type
dither	Input	Optional. Dither, specified as a scalar integer.	• fixdt([],N,0) • int, uint double and single are allowed for simulation but not for HDL code generation.
reset	Input	Optional. Reset the accumulator to zero when reset is high.	boolean
validIn	Input	Optional. Increment the phase when validIn is true. When validIn is false, the phase is held.	boolean
sin, cos, exp	Output	Generated waveform, returned as a scalar value. By default, the output is a sine wave, sin. The port labels change based on your selection for **Type of output signal**.	• fixdt() double and single are allowed for simulation but not for HDL code generation.
phase	Output	Optional. Current phase of NCO, returned as a scalar value.	fixdt(1,M,-Z), where M is the number of quantized accumulator bits, and Z is the accumulator word length.
validOut	Output	Whether the data output is valid or not. When validOut is true, the data output is valid. When validOut is false, the data output is not valid.	boolean

Parameters

Main

Note: This block supports double input for simulation but not for HDL code generation. When a data input is fixed point, or when no data input ports are enabled, the block computes the output waveform based on the fixed-point mask settings. When a data input is floating-point, the block ignores **Number of dither bits**, **Quantize phase**, **Number of quantizer accumulator bits**, **Enable look up table compression method**, and the **Data Types** tab, and computes a double-precision output waveform.

Phase increment source

Defines how you specify the phase increment. You can set the phase increment with an input port or you can enter a value for the parameter. The default value is `Input port`. If you select `Property`, the **Phase increment** parameter appears.

Phase increment

Specify the phase increment. The default value is 100. This value is scalar.

This parameter is visible when you set **Phase increment source** to `Property`.

Phase offset source

Defines how you specify the phase offset. You can set the phase offset from an input port or from a parameter. The default value is `Property`. If you select `Input port`, the offset port appears on the block icon.

Phase offset

Specify the phase offset. The default value is 0. This value is scalar. You can use integer data types, including fixed-point data types with zero fraction length.

This parameter is visible when you set **Phase offset source** to `Property`.

Dither source

Defines how you specify the dither. The default value is `Property`. You can set the dither from an input port or from a parameter. If you select `Property`, the **Number of dither bits** parameter appears. If you select `Input port`, a port appears on the block. If you select `None`, the block does not add dither.

Number of dither bits

Specify the dither bits. The default value is 4. This value must be a positive integer.

This option is visible when you set **Dither source** to `Property`.

Quantize phase

Select to enable quantization of the accumulated phase. The default value is selected.

When you select **Quantize phase**, the **Number of quantizer accumulator bits** parameter appears.

Number of quantizer accumulator bits

Specify the number of quantized accumulator bits. The default value is 12. This parameter determines the number of entries needed in the lookup table of sine values. The number of quantized accumulator bits must be less than the accumulator word length.

This parameter is visible only if you select **Quantize phase**.

Enable look up table compression method

Compress the lookup table when selected. The default value is not selected.

Enable reset input port

When selected, the `reset` port is present on the block icon. When `reset` is high, the block resets the accumulator to zero. The default value is not selected.

Enable validIn input port

When selected, the `validIn` port is present on the block icon. When `validIn` is `true`, the block increments the phase. When `validIn` is `false`, the phase is held. The default value is selected.

Type of output signal

Choose whether the block output signal is `Sine`, `Cosine`, `Complex exponential`, or `Sine and cosine`. If you select `Complex exponential`, the output is of the form `sine + j*cosine` and the port is labeled `exp`. If you select `Sine and cosine`, the block shows two ports, `sin` and `cos`. The default is `Sine`.

Show phase port

Output the current phase when selected. The default is not selected.

Data Types

Rounding Mode

The rounding mode when inputs are fixed point is `Floor`.

Overflow Mode

The overflow mode when inputs are fixed point is `Wrap`.

(Accumulator) **Data Type**

The output data type is `Binary point scaling`.

(Accumulator) **Signed**

The accumulator data type is signed.

(Accumulator) **Word length**

Accumulator word length. Default value is 16.

(Accumulator) **Fraction length**

Accumulator fraction length. Value is 0.

(Output) **Data Type**

Select `double`, `single`, or `Binary point scaling`. The default is `Binary point scaling`.

If you select `Binary point scaling`, the parameters for output word length and fraction length appear. All output data types are signed.

(Output) **Signed**

All output data types are signed.

(Output) **Word length**

Output word length. Default value is 16.

(Output) **Fraction length**

Output fraction length. Default value is 14.

Algorithms

Lookup Table Algorithm

When you select lookup table (LUT) compression, the NCO HDL Optimized block applies the Sunderland compression method. Sunderland techniques use trigonometric identities to divide each phase of the quarter sine wave into three components and express it as:

$$\sin(A+B+C) = \sin(A+B)\cos C + \cos A \cos B \cos C - \sin A \sin B \sin C$$

If the phase has 12 bits, the components are defined as:

- A, the four most significant bits

  $(0 \leq A \leq \frac{\pi}{2})$

- B, the following four bits

$$(0 \le B \le \frac{\pi}{2} \times 2^{-4})$$

- C, the four least significant bits

$$(0 \le C \le \frac{\pi}{2} \times 2^{-8})$$

Because C is small enough that sin(C)≅1 and cos(C)≅0, the equation is approximated by:

$$\sin(A+B+C) \approx \sin(A+B) + \cos A \sin C$$

The NCO HDL Optimized block implements this equation with one LUT for sin(A+B) and one LUT for cos(A)sin(C). The second term is a fine correction factor that you can truncate to fewer bits without losing precision. With the default accumulator size of 16 bits, and the example phase width of 12 bits, the LUTs use only $2^8 \times 16$ plus $2^8 \times 4$ bits (5kb). A quarter sine lookup table would use $2^{12} \times 16$ bits (65kb). This approximation is accurate within 1 LSB which gives an SNR of at least 60 dB on the output. See L. Cordesses, "Direct Digital Synthesis: A Tool for Periodic Wave Generation (Part 1)", IEEE Signal Processing Magazine, DSP Tips & Tricks column, pp. 50–54, Vol. 21, No. 4 July 2004.

Control Signals

There are two input control signals, `reset` and `validIn`, and one output control signal, `validOut`. When `reset` is high, the block sets the phase accumulator to zero. When `validIn` is high, the block increments the phase. When `validIn` is low, the block stops the phase accumulator and holds its state. When `validOut` is high, the output is valid.

Latency

The latency of the NCO HDL Optimized block is 6 cycles.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

For more information on implementations, properties, and restrictions for HDL code generation, see NCO HDL Optimized in the HDL Coder documentation.

See Also

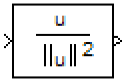
See Also
dsp.HDLNCO | NCO

Topics
"IEEE 802.11 WLAN - HDL Optimized Beacon Frame Receiver with Captured Data" (HDL Coder)

Introduced in R2013a

Normalization

Perform vector normalization along rows, columns, or specified dimension



Library

Math Functions / Math Operations

`dspmathops`

Description

The Normalization block independently normalizes each row, column, or vector of the specified dimension of the input. The block accepts both fixed- and floating-point signals in the squared 2-norm mode, but only floating-point signals in the 2-norm mode. The output always has the same dimensions as the input.

This block treats an arbitrarily dimensioned input *U* as a collection of vectors oriented along the specified dimension. The block normalizes these vectors by either their norm or the square of their norm.

For example, consider a 3-dimensional input *U(i,j,k)* and assume that you want to normalize along the second dimension. First, define the 2-dimensional intermediate quantity *V(i,k)* such that each element of *V* is the norm of one of the vectors in *U*:

$$V(i,k) = \left(\sum_{j=1}^{J} U^2(i,j,k) \right)^{1/2}$$

Given *V*, the output of the block *Y(i, j,k)* in 2-norm mode is

$$Y(i,j,k) = \frac{U(i,j,k)}{V(i,k)+b}$$

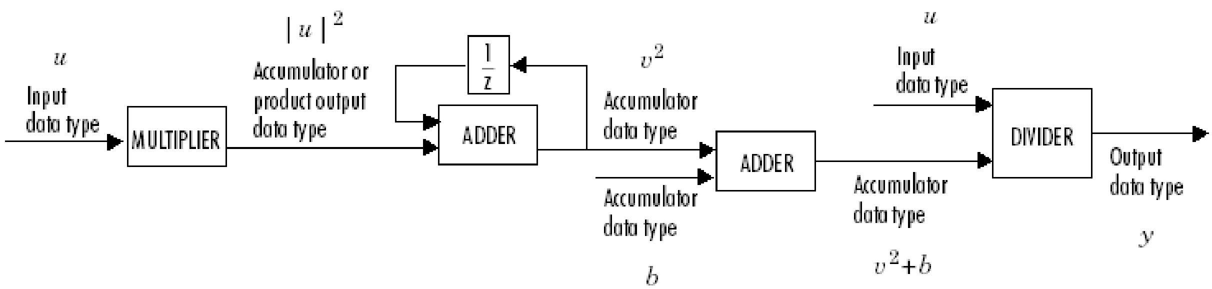
In squared 2-norm mode, the block output is

$$Y(i,j,k) = \frac{U(i,j,k)}{V(i,k)^2 + b}$$

The normalization bias, b, is typically chosen to be a small positive constant (for example, 1e-10) that prevents potential division by zero.

Fixed-Point Data Types

The following diagram shows the data types used within the Normalization block for fixed-point signals (squared 2-norm mode only).



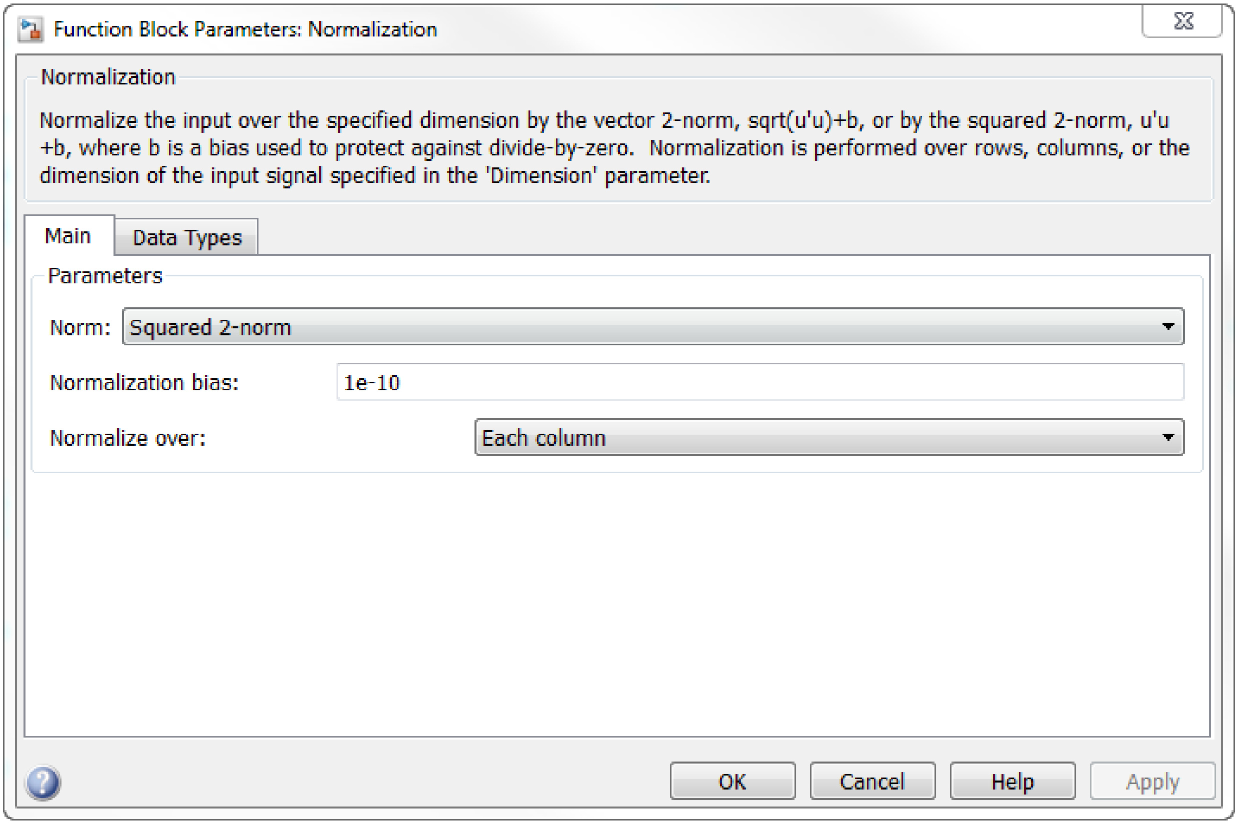
The output of the multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types". You can set the accumulator, product output, and output data types in the block dialog as discussed in "Dialog Box" on page 2-1319.

Examples

See "Zero Algorithmic Delay" in the *DSP System Toolbox User's Guide* for an example.

Dialog Box

The **Main** pane of the Normalization dialog appears as follows.



Norm

Specify the type of normalization to perform, `2-norm` or `Squared 2-norm`. `2-norm` mode supports floating-point signals only. `Squared 2-norm` supports both fixed-point and floating-point signals.

Normalization bias

Specify the real value *b* to be added in the denominator to avoid division by zero. Tunable (Simulink).

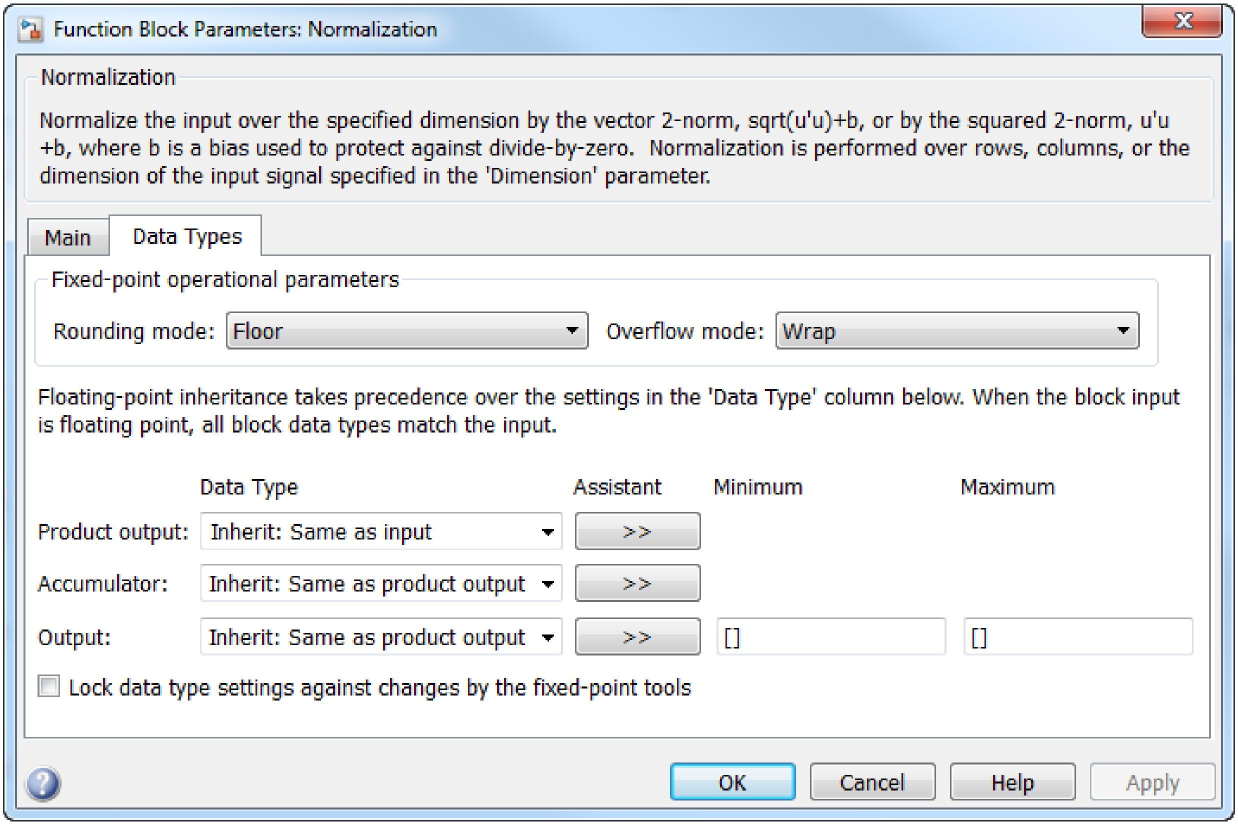
Normalize over

Specify whether to normalize along rows, columns, or the dimension specified in the **Dimension** parameter.

Dimension

Specify the one-based value of the dimension over which to normalize. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter is only visible if `Specified dimension` is selected for the **Normalize over** parameter.

The **Data Types** pane of the Normalization dialog appears as follows.



Note The parameters on this pane are only applicable to fixed-point signals when the block is in squared 2-norm mode. See "Fixed-Point Data Types" on page 2-1319 for a diagram of how the product output, accumulator, and output data types are used in this case.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-1319 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-1319 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`
- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-1319 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Port	Supported Data Types
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

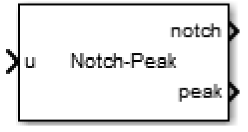
See Also

`Array-Vector Multiply`	DSP System Toolbox
`Reciprocal Condition`	DSP System Toolbox
`norm`	MATLAB

Introduced before R2006a

Notch-Peak Filter

Design second-order tunable notching and peaking IIR filter



Library

Filtering / Filter Designs

`dspfdesign`

Description

The Notch-Peak Filter block filters each channel of the input signal over time using a specified center frequency and 3 dB bandwidth. This block offers tunable filter design parameters, which enable you to tune the filter characteristics while the simulation is running.

The block designs the filter according to the filter parameters set in the block dialog box. The output port properties such as datatype, complexity, and dimension, are identical to the input port properties.

Each column of the input signal is treated as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then it is interpreted as a single channel.

This block supports variable-size input, enabling you to change the channel length during simulation. To enable variable-size input, clear the **Inherit sample rate from input** check box. The number of channels must remain constant.

Algorithms

This block brings the capabilities of the dsp.NotchPeakFilter System object to the Simulink environment.

The filter uses a coupled allpass structure to optimize joint computation of the peak and notch response. For information on the algorithms used by the Notch-Peak filter block, see the “Algorithms” on page 4-1385 section of `dsp.NotchPeakFilter`.

Parameters

Filter specification

Parameters or coefficients used to design the filter, specified as one of the following:

- `Bandwidth and center frequency` (default) — Design the filter using **3 dB bandwidth (Hz)** and **Notch/Peak center frequency (Hz)**.
- `Coefficients` — Design the filter using **Bandwidth coefficient** and **Center frequency coefficient**.
- `Quality factor and center frequency` — Design the filter using **Quality factor** and **Notch/Peak center frequency (Hz)**.

This parameter is nontunable.

3 dB bandwidth (Hz)

3 dB bandwidth of the filter, specified as a finite positive numeric scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter specification** to `Bandwidth and center frequency`. The default is `2205`. This parameter is tunable.

Notch/Peak center frequency (Hz)

Center frequency of the notch and peak of the filter, specified as a finite positive numeric scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter specification** to `Bandwidth and center frequency` or `Quality factor and center frequency`. The default is `11025`. This parameter is tunable.

Bandwidth coefficient

Coefficient that determines the 3 dB bandwidth of the filter, specified as a finite numeric scalar in the range `[-1 1]`.

- `-1` corresponds to the maximum 3 dB bandwidth (one-fourth the sample rate of the input signal).
- `1` corresponds to the minimum bandwidth (0 Hz, that is, an allpass filter).

This parameter applies when you set **Filter specification** to `Coefficients`. The default is `0.72654`. This parameter is tunable.

Center frequency coefficient

Coefficient that determines the center frequency of the filter, specified as a finite numeric scalar in the range `[-1 1]`.

- `-1` corresponds to the minimum center frequency (`0` Hz).
- `1` corresponds to the maximum center frequency (half the sample rate of the input signal).

This parameter applies when you set **Filter specification** to `Coefficients`. The default is `0`, which corresponds to quarter the sample rate of the input signal. This parameter is tunable.

Quality factor

Quality factor of the notch and peak filter, specified as a real positive scalar. The quality factor is defined as **Notch/Peak center frequency (Hz) / 3 dB bandwidth (Hz)**. A higher quality factor corresponds to a narrower peak or dip. This parameter applies when you set **Filter specification** to `Quality factor and center frequency`. The default is `5`. This parameter is tunable.

Output

Output of the filter block, specified as one of the following:

- `Notch and Peak` (default) — The block outputs the notch and peak responses of the filter.
- `Notch` — The block outputs the notch response of the filter.
- `Peak` — The block outputs the peak response of the filter.

This parameter is nontunable.

Inherit sample rate from input

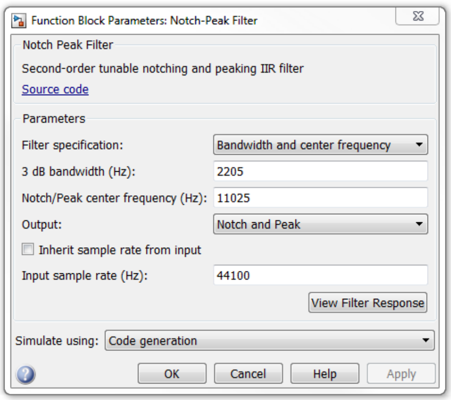
When you select this check box, the block sample rate is computed as *N*/*Ts*, where *N* is the frame size of the input signal, and *Ts* is the sample time of the input signal. When you clear this check box, the block's sample rate is the value specified in **Input sample rate (Hz)**. By default, this check box is selected.

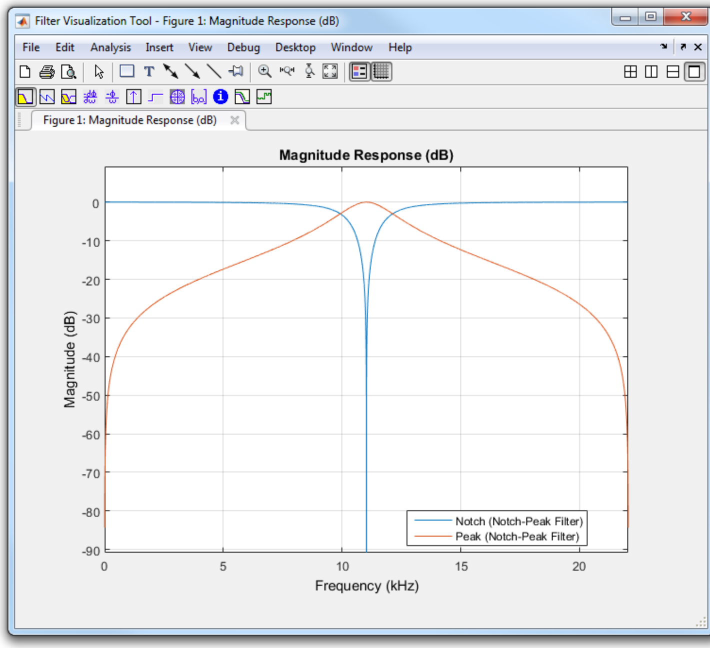
Input sample rate (Hz)

Sample rate of the input signal, specified as a positive scalar value. The default is `44100`. This parameter applies when you clear the **Inherit sample rate from input** check box. This parameter is nontunable.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the Notch-Peak Filter. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.





To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

See Also

dsp.NotchPeakFilter   DSP System Toolbox

Introduced in R2015a

Nyquist Filter

Design Nyquist filter

Library

Filtering / Filter Designs

`dspfdesign`

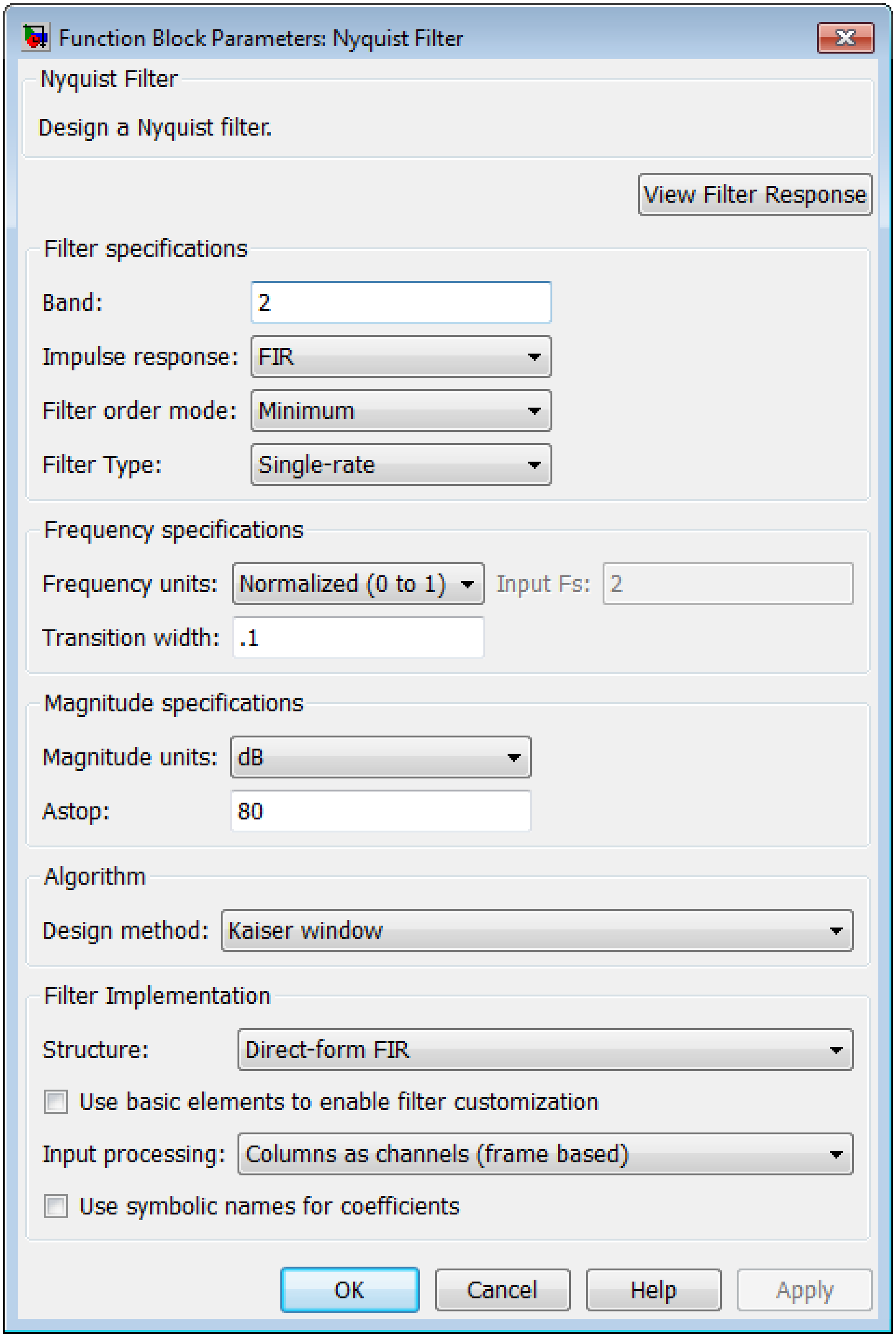
Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See "Nyquist Filter Design — Main Pane" on page 5-779 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Parameters of this block that do not change filter order or structure are tunable.


Function Block Parameters: Nyquist Filter
Nyquist Filter
Design a Nyquist filter.
View Filter Response
Filter specifications
Band:
2
Impulse response:
FIR
Filter order mode:
Minimum
Filter Type:
Single-rate
Frequency specifications
Frequency units:
Normalized (0 to 1)
Input Fs:
2
Transition width:
.1
Magnitude specifications
Magnitude units:
dB
Astop:
80
Algorithm
Design method:
Kaiser window
Filter Implementation
Structure:
Direct-form FIR
Use basic elements to enable filter customization
Input processing:
Columns as channels (frame based)
Use symbolic names for coefficients
OK
Cancel
Help
Apply

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Band

Specifies the location of the center of the transition region between the passband and the stopband. The center of the transition region, F_c, is calculated using the value for **Band**:

$F_c = F_s/(2 \cdot \mathbf{Band})$.

The default value, `2`, corresponds to a halfband filter.

Impulse response

Select either `FIR` or `IIR` from the drop-down list. `FIR` is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly. These options are both available only when **Band** is `2`. For values of **Band** greater than `2`, only FIR designs are supported.

Note: The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

Frequency constraints

Select the filter features that the block uses to define the frequency response characteristics.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Transition width

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default)
- `Squared` — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is `Butterworth`, and the default FIR method is `Kaiser window`.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent *n* in the relation $(1/f)^n$ to define the stopband decay. The block applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2006b

Octave Filter

Design octave filter

Note: The Octave Filter block will be removed from DSP System Toolbox in a future release. Existing instances of the block continue to run. For new code, use the Octave Filter block from Audio System Toolbox instead.

Library

Filtering / Filter Designs

`dspfdesign`

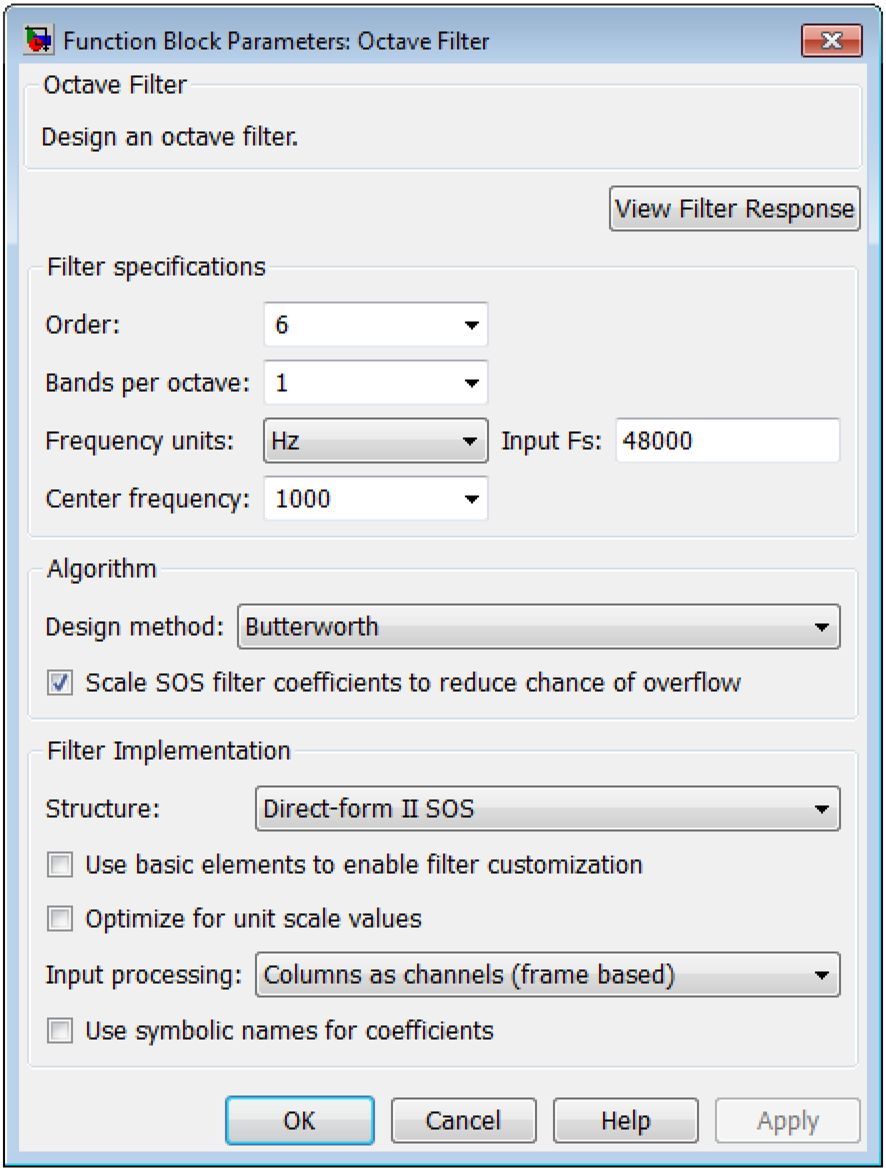
Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See "Octave Filter Design — Main Pane" on page 5-786 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Parameters of this block that do not change filter order or structure are tunable.



View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

Order

Specify filter order. Possible values are: `4, 6, 8, 10`.

Bands per octave

Specify the number of bands per octave. Possible values are: `1, 3, 6, 12, 24`.

Frequency units

Specify frequency units as `Hz` or `kHz`.

Input Fs

Specify the input sampling frequency in the frequency units specified previously.

Center Frequency

Select from the drop-down list of available center frequency values.

Algorithm

Design Method

Butterworth is the design method used for this type of filter.

Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

Filter Implementation

Structure

Specify filter structure. Choose from:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

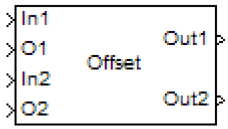
Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2007a

Offset

Truncate vectors by removing or keeping beginning or ending values



Library

Signal Operations

`dspsigops`

Description

The Offset block removes or keeps values from the beginning or end of the input vectors. You specify the length of the output vectors using the **Output port length** parameter. The inputs to the In ports (In1, In2, ...) can be scalars or vectors, but they must be the same size and data type. The offset values are the inputs to the O ports (O1, O2, ...); they must be scalar values with the same data type. These offset values should be integer values because they determine the number of values the block discards or retains from each input vector. The block rounds any offset value that is a noninteger value to the nearest integer value. There is one output port for each pair of In and O ports.

Use the **Mode** parameter to determine which values the block discards or retains from the input vector. To discard the initial values of the vector, select `Remove beginning samples`. To discard the final values of the vector, select `Remove ending samples`. To retain the initial values of the vector, select `Keep beginning samples`. To retain the final values of a vector, select `Keep ending` samples.

Use the **Number of input data-offset pairs** parameter to specify the number of inputs to the block. The number of input ports is twice the scalar value you enter. For example, if you enter `3`, ports In1, O1, In2, O2, In3, and O3 appear on the block.

The block uses the **Output port length** parameter to determine the length of the output vectors. If you select `Same as input`, the block outputs vectors that are the same length as the input to the In ports. If you select `User-defined`, the **Output length** parameter appears. Enter a scalar that represents the desired length of the output vectors. If your desired output length is greater than the number of values you extracted from your input vector, the block zero-pads the end of the vector to reach the length you specified.

Use the **Action for out of range offset value** parameter to determine how the block behaves when an offset value is not in the range 0 ≤ offset value ≤ N, where N is the input vector length. Select `Clip` if you want any offset values less than 0 to be set to 0 and any offset values greater than N to be set to N. Select `Clip and warn` if you want to be warned when any offset values less than 0 are set to 0 and any offset values greater than N are set to N. Select `Error` if you want the simulation to stop and display an error when the offset values are out of range.

Parameters

Mode

Use this parameter to determine which values the block discards or retains from the input vector. Your choices are `Remove beginning samples`, `Remove ending samples`, `Keep beginning samples`, and `Keep ending samples`.

Number of input data-offset pairs

Specify the number of inputs to the block. The number of input ports is twice the scalar value you enter.

Output port length

Use this parameter to specify the length of the output vectors. If you select `Same as input`, the output vectors are the same length as the input vectors. If you select `User-defined`, you can enter the desired length of the output vectors.

Output length

Enter a scalar that represents the desired length of the output vectors. This parameter is visible if, for the **Output port length** parameter, you select `User-defined`.

Action for out of range offset value

Use this parameter to determine how the block behaves when an offset value is not in the range such that 0 ≤ offset value ≤ N, where N is the input vector length. When you want any offset values less than 0 to be set to 0 and any offset values greater than N to be set to N, select `Clip`. When you want to be warned when any offset values less than 0 are set to 0 and any offset values greater than N are set to N, select `Clip and warn`. When you want the simulation to stop and display an error when the offset values are out of range, select `Error`.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
O	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

Introduced before R2006a

Overlap-Add FFT Filter

Implement overlap-add method of frequency-domain filtering

Library

Filtering / Filter Implementations

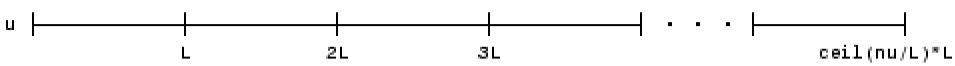
`dsparch4`

Description

The Overlap-Add FFT Filter block uses an FFT to implement the *overlap-add method*, a technique that combines successive frequency-domain filtered sections of an input sequence.

The block accepts vector or matrix inputs, and treats each column of the input as an individual channel. The block unbuffers the input data into row vectors such that the length of the output vector is equal to the number of channels in the input. The data output rate of the block is M times faster than its data input rate, where M is the length of the columns in the input (frame-size).

The block breaks the scalar input sequence `u`, of length `nu`, into length-`L` nonoverlapping data sections,



which it linearly convolves with the filter's FIR coefficients,

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_{n+1} z^{-n}$$

The numerator coefficients for H(z) are specified as a vector by the **FIR coefficients** parameter. The coefficient vector, `b = [b(1) b(2) ... b(n+1)]`, can be generated by one of the filter design functions in the Signal Processing Toolbox product, such as `fir1`. All filter states are internally initialized to zero.

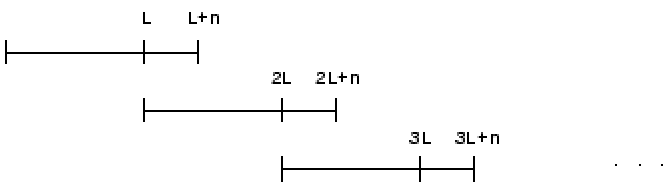
When either the filter coefficients or the inputs to the block are complex, the **Output** parameter should be set to `Complex`. Otherwise, the default **Output** setting, `Real`, instructs the block to take only the real part of the solution.

The block's overlap-add operation is equivalent to

```
y = ifft(fft(u(i:i+L-1),nfft) .* fft(b,nfft))
```

where you specify `nfft` in the **FFT size** parameter as a power-of-two value greater (typically *much* greater) than `n+1`. Values for **FFT size** that are not powers of two are rounded upwards to the nearest power-of-two value to obtain `nfft`.

The block overlaps successive output sections by `n` points and sums them.



The first `L` samples of each summation are output in sequence. The block chooses the parameter `L` based on the filter order and the FFT size.

```
L = nfft - n
```

Latency

In *single-tasking* operation, the Overlap-Add FFT Filter block has a latency of `nfft-n+1` samples. The first `nfft-n+1` consecutive outputs from the block are zero; the first filtered input value appears at the output as sample `nfft-n+2`.

In *multitasking* operation, the Overlap-Add FFT Filter block has a latency of `2*(nfft-n)+1` samples. The first `2*(nfft-n)+1` consecutive outputs from the block are zero; the first filtered input value appears at the output as sample `2*(nfft-n)+3`.

Note: For more information on latency and the Simulink software tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

Parameters

FFT size

The size of the FFT, which should be a power-of-two value greater than the length of the specified FIR filter.

FIR coefficients

The filter numerator coefficients.

Output

The complexity of the output; `Real` or `Complex`. When the input signal or the filter coefficients are complex, this should be set to `Complex`.

Treat Mx1 and unoriented sample-based signals as

Specify how the block treats sample-based *M*-by-1 column vectors and unoriented sample-based vectors of length *M*. You can select one of the following options:

- `One channel` — When you select this option, the block treats *M*-by-1 and unoriented sample-based inputs as a column vector (one channel).
- `M channels (this choice will be removed – see release notes)` — When you select this option, the block treats *M*-by-1 and unoriented sample-based inputs as a 1-by-*M* row vector.

Note: This parameter will be removed in a future release. At that time, the block will always treat *M*-by-1 and unoriented vectors as a single channel.

References

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

See Also

`Overlap-Save FFT Filter`	DSP System Toolbox product

Introduced before R2006a

Overlap-Save FFT Filter

Implement overlap-save method of frequency-domain filtering

Library

Filtering / Filter Implementations

`dsparch4`

Description

The Overlap-Save FFT Filter block uses an FFT to implement the *overlap-save method*, a technique that combines successive frequency-domain filtered sections of an input sequence.

The block accepts vector or matrix inputs, and treats each column of the input as an individual channel. The block unbuffers the input data into row vectors such that the length of the output vector is equal to the number of channels in the input. The data output rate of the block is M times faster than its data input rate, where M is the length of the columns in the input (frame-size).

Overlapping sections of input `u` are circularly convolved with the FIR filter coefficients

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_{n+1} z^{-n}$$

The numerator coefficients for $H(z)$ are specified as a vector by the **FIR coefficients** parameter. The coefficient vector, `b = [b(1) b(2) ... b(n+1)]`, can be generated by one of the filter design functions in the Signal Processing Toolbox product, such as `fir1`. All filter states are internally initialized to zero.

When either the filter coefficients or the inputs to the block are complex, the **Output** parameter should be set to `Complex`. Otherwise, the default **Output** setting, `Real`, instructs the block to take only the real part of the solution.

The circular convolution of each section is computed by multiplying the FFTs of the input section and filter coefficients, and computing the inverse FFT of the product.

```
y = ifft(fft(u(i:i+(L-1)),nfft) .* fft(b,nfft))
```

where you specify `nfft` in the **FFT size** parameter as a power of two value greater (typically *much* greater) than `n+1`. Values for **FFT size** that are not powers of two are rounded upwards to the nearest power-of-two value to obtain `nfft`.

The first `n` points of the circular convolution are invalid and are discarded. The Overlap-Save FFT Filter block outputs the remaining `nfft-n` points, which are equivalent to the linear convolution.

Latency

In *single-tasking* operation, the Overlap-Save FFT Filter block has a latency of `nfft-n+1` samples. The first `nfft-n+1` consecutive outputs from the block are zero; the first filtered input value appears at the output as sample `nfft-n+2`.

In *multitasking* operation, the Overlap-Save FFT Filter block has a latency of `2*(nfft-n+1)` samples. The first `2*(nfft-n+1)` consecutive outputs from the block are zero; the first filtered input value appears at the output as sample `2*(nfft-n)+3`.

Note: For more information on latency and the Simulink environment tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

Parameters

FFT size

The size of the FFT, which should be a power of two value greater than the length of the specified FIR filter.

FIR coefficients

The filter numerator coefficients.

Output

The complexity of the output; `Real` or `Complex`. When the input signal or the filter coefficients are complex, this should be set to `Complex`.

Treat Mx1 and unoriented sample-based signals as

Specify how the block treats sample-based M-by-1 column vectors and unoriented sample-based vectors of length M. You can select one of the following options:

- `One channel` — When you select this option, the block treats M-by-1 and unoriented sample-based inputs as a column vector (one channel).
- `M channels (this choice will be removed — see release notes)` — When you select this option, the block treats M-by-1 and unoriented sample-based inputs as a 1-by-M row vector.

Note: This parameter will be removed in a future release. At that time, the block will always treat M-by-1 and unoriented vectors as a single channel.

References

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

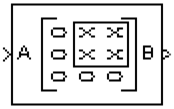
See Also

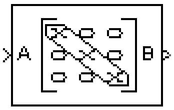
`Overlap-Add FFT Filter` DSP System Toolbox

Introduced before R2006a

Overwrite Values

Overwrite submatrix or subdiagonal of input





Library

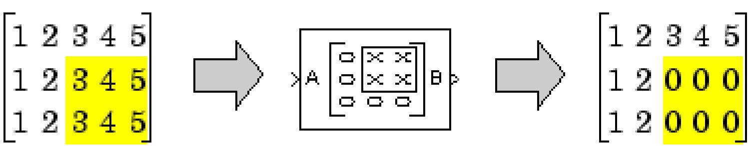
- Math Functions / Matrices and Linear Algebra / Matrix Operations

  `dspmtrx3`
- Signal Management / Indexing

  `dspindex`

Description

The Overwrite Values block overwrites a contiguous submatrix or subdiagonal of an input matrix. You can provide the overwriting values by typing them in a block parameter, or through an additional input port, which is useful for providing overwriting values that change at each time step.



The block accepts scalars, vectors and matrices. The output always has the same size as the original input signal, not necessarily the same size as the signal containing the overwriting values. The input(s) and output of this block must have the same data type.

Specifying the Overwriting Values

The **Source of overwriting value(s)** parameter determines how you must provide the overwriting values, and has the following settings.

- `Specify via dialog` — You must provide the overwriting value(s) in the **Overwrite with** parameter. The block uses the same overwriting values to overwrite the specified portion of the input at each time step. To learn how to specify valid overwriting values, see “Valid Overwriting Values” on page 2-1355.
- `Second input port` — You must provide overwriting values through a second block input port, `V`. Use this setting to provide different overwriting values at each time step. The output inherits its size and rate from the input signal, *not* the overwriting values.

  The rate at which you provide the overwriting values through input port `V` must match the rate at which the block receives each input matrix at input port `A`. In other words, the input signals must have the same Simulink sample time.

Valid Overwriting Values

The overwriting values can be a single constant, vector, or matrix, depending on the portion of the input you are overwriting, regardless of whether you provide the overwriting values through an input port or by providing them in the **Overwrite with** parameter.

Valid Overwriting Values

Portion of Input to Overwrite	Valid Overwriting Values	Example
A single element in the input $\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$	Any constant value, v	v `= 9` $\begin{bmatrix} x & x & x & x & x \\ x & x & x & 9 & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$

Portion of Input to Overwrite	Valid Overwriting Values	Example
A length-k portion of the diagonal $\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$	Any length-k column or row vector, v	$k = 3 \quad v = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix}$ or $\begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$ $\begin{bmatrix} 2 & x & x & x & x \\ x & 4 & x & x & x \\ x & x & 6 & x & x \\ x & x & x & x & x \end{bmatrix}$
A length-k portion of a row $\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$	Any length-k row vector, v	$k = 3 \quad v = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix}$ $\begin{bmatrix} x & x & x & x & x \\ x & 2 & 4 & 6 & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$
A length-k portion of a column $\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$	Any length-k column vector, v	$k = 2 \quad v = \begin{bmatrix} 4 \\ 6 \end{bmatrix}$ $\begin{bmatrix} x & x & x & x & x \\ x & x & x & 4 & x \\ x & x & x & 6 & x \\ x & x & x & x & x \end{bmatrix}$
An m-by-n submatrix $\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$	Any m-by-n matrix, v	$m = 2$ $n = 3$ $v = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ $\begin{bmatrix} x & x & x & x & x \\ x & x & 4 & 5 & 6 \\ x & x & 7 & 8 & 9 \\ x & x & x & x & x \end{bmatrix}$

This block supports Simulink virtual buses.

Parameters

Note Only some of the following parameters are visible in the dialog box at any one time.

Overwrite

Determines whether to overwrite a specified submatrix or a specified portion of the diagonal.

Source of overwriting value(s)

Determines where you must provide the overwriting values: either through an input port, or by providing them in the **Overwrite with** parameter. For more information, see "Specifying the Overwriting Values" on page 2-1355.

Overwrite with

The value(s) with which to overwrite the specified portion of the input matrix. Enabled only when **Source of overwriting value(s)** is set to `Specify via dialog`. To learn how to specify valid overwriting values, see "Valid Overwriting Values" on page 2-1355.

Row span

The range of input rows to be overwritten. Options are `All rows`, `One row`, or `Range of rows`. For descriptions of these options, see "Parameters" on page 2-1357.

Row/Starting row

The input row that is the first row of the submatrix that the block overwrites. For a description of the options for the **Row** and **Starting row** parameters, see Settings for Row, Column, Starting Row, and Starting Column Parameters. **Row** is enabled when **Row span** is set to `One row`, and **Starting row** when **Row span** is set to `Range of rows`.

Row index/Starting row index

Index of the input row that is the first row of the submatrix that the block overwrites. See how to use these parameters in Settings for Row, Column, Starting Row, and Starting Column Parameters. **Row index** is enabled when **Row** is set to `Index`, and **Starting row index** when **Starting row** is set to `Index`.

Row offset/Starting row offset

The offset of the input row that is the first row of the submatrix that the block overwrites. See how to use these parameters in Settings for Row, Column, Starting Row, and Starting Column Parameters. **Row offset** is enabled when **Row** is set to `Offset from middle` or `Offset from last`, and **Starting row offset** is enabled when **Starting row** is set to `Offset from middle` or `Offset from last`.

Ending row

The input row that is the last row of the submatrix that the block overwrites. For a description of this parameter's options, see Settings for Ending Row and Ending Column Parameters. This parameter is enabled when **Row span** is set to `Range of rows`, and **Starting row** is set to any option but `Last`.

Ending row index

Index of the input row that is the last row of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters. Enabled when **Ending row** is set to `Index`.

Ending row offset

The offset of the input row that is the last row of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters. Enabled when **Ending row** is set to `Offset from middle` or `Offset from last`.

Column span

The range of input columns to be overwritten. Options are `All columns`, `One column`, or `Range of columns`. For descriptions of the analogous row options, see "Parameters" on page 2-1357.

Column/Starting column

The input column that is the first column of the submatrix that the block overwrites. For a description of the options for the **Column** and **Starting column** parameters, see Settings for Row, Column, Starting Row, and Starting Column Parameters. **Column** is enabled when **Column span** is set to `One column`, and **Starting column** when **Column span** is set to `Range of columns`.

Column index/Starting column index

Index of the input column that is the first column of the submatrix that the block overwrites. See how to use these parameters in Settings for Row, Column, Starting Row, and Starting Column Parameters. **Column index** is enabled when **Column** is set to `Index`, and **Starting column index** when **Starting column** is set to `Index`.

Column offset/Starting column offset

The offset of the input column that is the first column of the submatrix that the block overwrites. See how to use these parameters in Settings for Row, Column, Starting Row, and Starting Column Parameters. **Column offset** is enabled when **Column** is set to `Offset from middle` or `Offset from last`, and **Starting column offset** is enabled when **Starting column** is set to `Offset from middle` or `Offset from last`.

Ending column

The input column that is the last column of the submatrix that the block overwrites. For a description of this parameter's options, see Settings for Ending Row and Ending Column Parameters. This parameter is enabled when **Column span** is set to `Range of columns`, and **Starting column** is set to any option but `Last`.

Ending column index

Index of the input column that is the last column of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters. This parameter is enabled when **Ending column** is set to `Index`.

Ending column offset

The offset of the input column that is the last column of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters. This parameter is enabled when **Ending column** is set to `Offset from middle` or `Offset from last`.

Diagonal span

The range of diagonal elements to be overwritten. Options are `All elements`, `One element`, or `Range of elements`. For descriptions of these options, see "Overwriting a Subdiagonal" on page 2-1364.

Element/Starting element

The input diagonal element that is the first element in the subdiagonal that the block overwrites. For a description of the options for the **Element** and **Starting element** parameters, see Element and Starting Element Parameters. **Element** is enabled when **Element span** is set to `One element`, and **Starting element** when **Element span** is set to `Range of elements`.

Element index/Starting element index

Index of the input diagonal element that is the first element of the subdiagonal that the block overwrites. See how to use these parameters in Element and Starting

Element Parameters. **Element index** is enabled when **Element** is set to `Index`, and **Starting element index** when **Starting element** is set to `Index`.

Element offset/Starting element offset

The offset of the input diagonal element that is the first element of the subdiagonal that the block overwrites. See how to use these parameters in Element and Starting Element Parameters. **Element offset** is enabled when **Element** is set to `Offset from middle` or `Offset from last`, and **Starting element offset** is enabled when **Starting element** is set to `Offset from middle` or `Offset from last`.

Ending element

The input diagonal element that is the last element of the subdiagonal that the block overwrites. For a description of this parameter's options, see Ending Element Parameters. This parameter is enabled when **Element span** is set to `Range of elements`, and **Starting element** is set to any option but `Last`.

Ending element index

Index of the input diagonal element that is the last element of the subdiagonal that the block overwrites. See how to use this parameter in Ending Element Parameters. This parameter is enabled when **Ending element** is set to `Index`.

Ending element offset

The offset of the input diagonal element that is the last element of the subdiagonal that the block overwrites. See how to use this parameter in Ending Element Parameters. This parameter is enabled when **Ending element** is set to `Offset from middle` or `Offset from last`.

Examples

Overwriting a Submatrix

To overwrite a submatrix, follow these steps:

1. Set the **Overwrite** parameter to `Submatrix`.
2. Specify the overwriting values as described in "Specifying the Overwriting Values" on page 2-1355.
3. Specify which rows and columns of the input matrix are contained in the submatrix that you want to overwrite by setting the **Row span** parameter to one of the following options and the **Column span** to the analogous column-related options:

- `All rows` — The submatrix contains all rows of the input matrix.
- `One row` — The submatrix contains only one row of the input matrix, which you must specify in the **Row** parameter, as described in the following table.
- `Range of rows` — The submatrix contains one or more rows of the input, which you must specify in the **Starting Row** and **Ending row** parameters, as described in the following tables.

4 When you set **Row span** to `One row` or `Range of rows`, you need to further specify the row(s) contained in the submatrix by setting the **Row** or **Starting row** and **Ending row** parameters. Likewise, when you set **Column span** to `One column` or `Range of columns`, you must further specify the column(s) contained in the submatrix by setting the **Column** or **Starting column** and **Ending column** parameters. For descriptions of the settings for these parameters, see the following tables.

Settings for Row, Column, Starting Row, and Starting Column Parameters

Settings for Specifying the Submatrix's First Row or Column	First Row of Submatrix (Only row for Row span = One row)	First Column of Submatrix (Only row for Row span = One row)
`First`	First row of the input	First column of the input
`Index`	Input row specified in the **Row index** parameter	Input column specified in the **Column index** parameter
`Offset from last`	Input row with the index `M - rowOffset` where `M` is the number of input rows, and `rowOffset` is the value of the **Row offset** or **Starting row offset** parameter	Input column with the index `N - colOffset` where `N` is the number of input columns, and `colOffset` is the value of the **Column offset** or **Starting column offset** parameter
`Last`	Last row of the input	Last column of the input
`Offset from middle`	Input row with the index `floor(M/2 + 1 - rowOffset)` where `M` is the number of input rows, and `rowOffset` is the value of the **Row offset** or **Starting row offset** parameter	Input column with the index `floor(N/2 + 1 - rowOffset)` where `N` is the number of input columns, and `colOffset` is the value of the or **Column offset** or **Starting column offset** parameter

Settings for Specifying the Submatrix's First Row or Column	First Row of Submatrix (Only row for Row span = One row)	First Column of Submatrix (Only row for Row span = One row)
`Middle`	Input row with the index `floor(M/2 + 1)` where `M` is the number of input rows	Input columns with the index `floor(N/2 + 1)` where `N` is the number of input columns

Settings for Ending Row and Ending Column Parameters

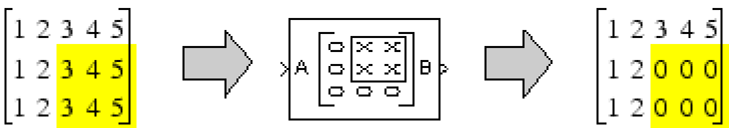
Settings for Specifying the Submatrix's Last Row or Column	Last Row of Submatrix	Last Column of Submatrix
`Index`	Input row specified in the **Ending row index** parameter	Input column specified in the **Ending column index** parameter
`Offset from last`	Input row with the index `M - rowOffset` where `M` is the number of input rows, and `rowOffset` is the value of the **Ending row offset** parameter	Input column with the index `N - colOffset` where `N` is the number of input columns, and `colOffset` is the value of the **Ending column offset** parameter
`Last`	Last row of the input	Last column of the input
`Offset from middle`	Input row with the index `floor(M/2 + 1 - rowOffset)` where `M` is the number of input rows, and `rowOffset` is the value of the **Ending row offset** parameter	Input column with the index `floor(N/2 + 1 - rowOffset)` where `N` is the number of input columns, and `colOffset` is the value of the **Ending column offset** parameter
`Middle`	Input row with the index `floor(M/2 + 1)` where `M` is the number of input rows	Input columns with the index `floor(N/2 + 1)` where `N` is the number of input columns

For example, to overwrite the lower-right 2-by-3 submatrix of a 3-by-5 input matrix with all zeros, enter the following set of parameters:

- **Overwrite** = `Submatrix`
- **Source of overwriting value(s)** = `Specify via dialog`

- **Overwrite with** = `0`
- **Row span** = `Range of rows`
- **Starting row** = `Index`
- **Starting row index** = `2`
- **Ending row** = `Last`
- **Column span** = `Range of columns`
- **Starting column** = `Offset from last`
- **Starting column offset** = `2`
- **Ending column** = `Last`

The following figure shows the block with the above settings overwriting a portion of a 3-by-5 input matrix.



There are often several possible parameter combinations that select the *same* submatrix from the input. For example, instead of specifying `Last` for **Ending column**, you could select the same submatrix by specifying

- **Ending column** = `Index`
- **Ending column index** = `5`

Overwriting a Subdiagonal

To overwrite a subdiagonal, follow these steps:

1. Set the **Overwrite** parameter to `Diagonal`.
2. Specify the overwriting values as described in "Specifying the Overwriting Values" on page 2-1355.
3. Specify the subdiagonal that you want to overwrite by setting the **Diagonal span** parameter to one of the following options:
   - `All elements` — Overwrite the entire input diagonal.
   - `One element` — Overwrite one element in the diagonal, which you must specify in the **Element** parameter (described below).
   - `Range of elements` — Overwrite a portion of the input diagonal, which you must specify in the **Starting element** and **Ending element** parameters, as described in the following table.
4. When you set **Diagonal span** to `One element` or `Range of elements`, you need to further specify which diagonal element(s) to overwrite by setting the **Element** or **Starting element** and **Ending element** parameters. See the following tables.

Element and Starting Element Parameters

Settings for Element and Starting Element Parameters	First Element in Subdiagonal (Only element when Diagonal span = One element)
`First`	Diagonal element in first row of the input
`Index`	kth diagonal element, where k is the value of the **Element index** or **Starting element index** parameter
`Offset from last`	Diagonal element in the row with the index `M - offset` where `M` is the number of input rows, and `offset` is the value of the **Element offset** or **Starting element offset** parameter
`Last`	Diagonal element in the last row of the input
`Offset from middle`	Diagonal element in the input row with the index `floor(M/2 + 1 - offset)` where `M` is the number of input rows, and `offset` is the value of the **Element offset** or **Starting element offset** parameter

Settings for Element and Starting Element Parameters	First Element in Subdiagonal (Only element when Diagonal span = One element)
`Middle`	Diagonal element in the input row with the index `floor(M/2 + 1)` where `M` is the number of input rows

Ending Element Parameters

Settings for Ending Element Parameter	Last Element in Subdiagonal
`Index`	*k*th diagonal element, where *k* is the value of the **Ending element index** parameter
`Offset from last`	Diagonal element in the row with the index `M - offset` where `M` is the number of input rows, and `offset` is the value of the **Ending element offset** parameter
`Last`	Diagonal element in the last row of the input
`Offset from middle`	Diagonal element in the input row with the index `floor(M/2 + 1 - offset)` where `M` is the number of input rows, and `offset` is the value of the **Ending element offset** parameter
`Middle`	Diagonal element in the input row with the index `floor(M/2 + 1)` where `M` is the number of input rows

Supported Data Types

The input(s) and output of this block must have the same data type.

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean

Port	Supported Data Types
	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
V	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
B	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

Reshape	Simulink
Selector	Simulink
Submatrix	DSP System Toolbox
`Variable Selector`	DSP System Toolbox
`reshape`	MATLAB

Introduced before R2006a

Pad

Pad or truncate specified dimension(s)

Library

Signal Operations

`dspsigops`

Description

The Pad block extends or crops the dimensions of the input by padding or truncating along its columns, rows, columns and rows, or any dimension(s) you specify. Truncation occurs when you specify output dimensions that are shorter than the corresponding input dimensions. If the input and output lengths are the same, the block is a pass-through.

You can enter the pad value in the block mask or via an input port. You can enter output sizes in the block mask, or have the block pad the specified dimensions until their length is the next highest power of two. The **Pad signal at** parameter controls whether the specified input dimensions are padded or truncated at their beginning, end, or both. For odd pad or truncation lengths, the extra pad value or truncation is applied to the end of the signal. When the block is in `Specified dimensions` mode, you can specify either the output size or the pad size.

You can have the block warn or error when an input signal is truncated using the **Action when truncation occurs** parameter.

Parameters

Pad over

Specify the dimensions over which to pad or truncate: `Columns`, `Rows`, `Columns and rows`, `None`, or `Specified dimensions`.

Dimensions to pad

Specify the one-based dimension(s) over which to pad or truncate. The value for this parameter can be a scalar or a vector. For example, specify `1` to pad columns. Specify `[1 2]` to pad columns and rows. Specify `[1 3 5]` to pad the first, third, and fifth dimensions.

This parameter is only visible when `Specified dimensions` is selected for the **Pad over** parameter.

Pad value source

Choose how you specify the pad value. The pad value can come from an input port or from the dialog:

- If you select `Input port`, the `PVal` port appears on the block icon.
- If you select `Specify via dialog`, the **Pad value** parameter appears.

Pad value

Specify the constant scalar value with which to pad the input. Tunable (Simulink).

This parameter is only visible when `Specify via dialog` is selected for the **Pad value source** parameter.

Output column mode

Choose how you specify the column length of the output:

- If you select `User-specified`, the **Column size** parameter appears.
- If you select `Next power of two`, the block pads the output columns until their length is the next highest power of two. If the column length is already a power of two, the columns are not padded.

This parameter is only visible when `Columns` or `Columns and rows` is selected for the **Pad over** parameter.

Column size

Specify the column length of the output. If the specified column length is longer than the input column length, the columns are padded. If the specified column length is shorter than the input column length, the columns are truncated. This parameter is only visible when `User-specified` is selected for the **Output column mode** parameter.

Output row mode

Choose how you specify the output row length of the output:

- If you select `User-specified`, the **Row size** parameter appears.
- If you select `Next power of two`, the block pads the output rows until their length is the next highest power of two. If the row length is already a power of two, the rows are not padded.

This parameter is only visible when `Rows` or `Columns and rows` is selected for the **Pad over** parameter.

Row size

Specify the row length of the output. If the specified row length is longer than the input row length, the rows are padded. If the specified row length is shorter than the input row length, the rows are truncated. This parameter is only visible when `User-specified` is selected for the **Output row mode** parameter.

Specify

Choose whether you want to control the output length of the specified dimensions by specifying the pad size or the output size.

This parameter is only visible when `Specified dimensions` is selected for the **Pad over** parameter.

Pad size at beginning

Specify how many values to add to the beginning of the input signal along the specified dimension(s). This parameter must be a scalar or a vector with the same number of elements as the **Dimensions to pad** parameter. Each element in the **Pad size at beginning** parameter gives the pad length for the beginning of the corresponding dimension in the **Dimensions to pad** parameter. Values of this parameter must be zero or a positive integer.

This parameter is only visible if `Pad size` is selected for the **Specify** parameter.

Pad size at end

Specify how many values to add to the end of the input signal along the specified dimension(s). This parameter must be a scalar or a vector with the same number of elements as the **Dimensions to pad** parameter. Each element in the **Pad size at end** parameter gives the pad length for the end of the corresponding dimension in the **Dimensions to pad** parameter. Values of this parameter must be zero or a positive integer.

This parameter is only visible if `Pad size` is selected for the **Specify** parameter.

Output size mode

Choose how you specify the output length of the specified dimensions:

- If you select `User-specified`, the **Output size** parameter appears.
- If you select `Next power of two`, the block pads the specified dimensions until their length is the next highest power of two. If the dimension length is already a power of two, no padding occurs in that dimension.

This parameter is only visible if `Output size` is selected for the **Specify** parameter.

Output size

Specify the output length of the specified dimension(s). This parameter must be a scalar or a vector with the same number of elements as the **Dimensions to pad** parameter. Each element in the **Output size** vector gives the output length for the corresponding dimension in the **Dimensions to pad** vector. If the specified length is longer than the input length for a given dimensions, that dimension is padded. If the specified length is shorter than the input length for a given dimension, that dimension is truncated.

This parameter is only visible if `Output size` is selected for the **Specify** parameter.

Pad signal at

Specify whether to pad or truncate the signal at the `Beginning`, `End`, or `Beginning and end` of the specified dimension(s). When you select `Beginning and end`, half the pad length is added to the beginning of the signal, and half is added to the end of the signal. For an odd pad length, the extra value is added to the end of the signal. This also applies to truncation. In this mode, an equal number of values are truncated from the beginning and the end of the signal. In the case of an odd truncation length, the extra value is removed from the end of the signal.

Action when truncation occurs

Choose `None` when you do not want to be notified that the input is truncated. Select `Warning` to display a warning when the input is truncated. Choose `Error` when to display an error and terminate the simulation when the input is truncated.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating-point

Port	Supported Data Types
	• Single-precision floating-point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating-point • Single-precision floating-point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

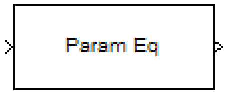
See Also

Concatenate	Simulink
Repeat	DSP System Toolbox
Submatrix	DSP System Toolbox
Upsample	DSP System Toolbox
`Variable Selector`	DSP System Toolbox

Introduced before R2006a

Parametric Equalizer

Design parametric equalizer



Note: The Parametric Equalizer block has been replaced by the `Parametric EQ Filter` block. Existing instances of the Parametric Equalizer block will continue to operate. For new models, use the Parametric EQ Filter block.

Library

Filtering / Filter Designs

`dspfdesign`

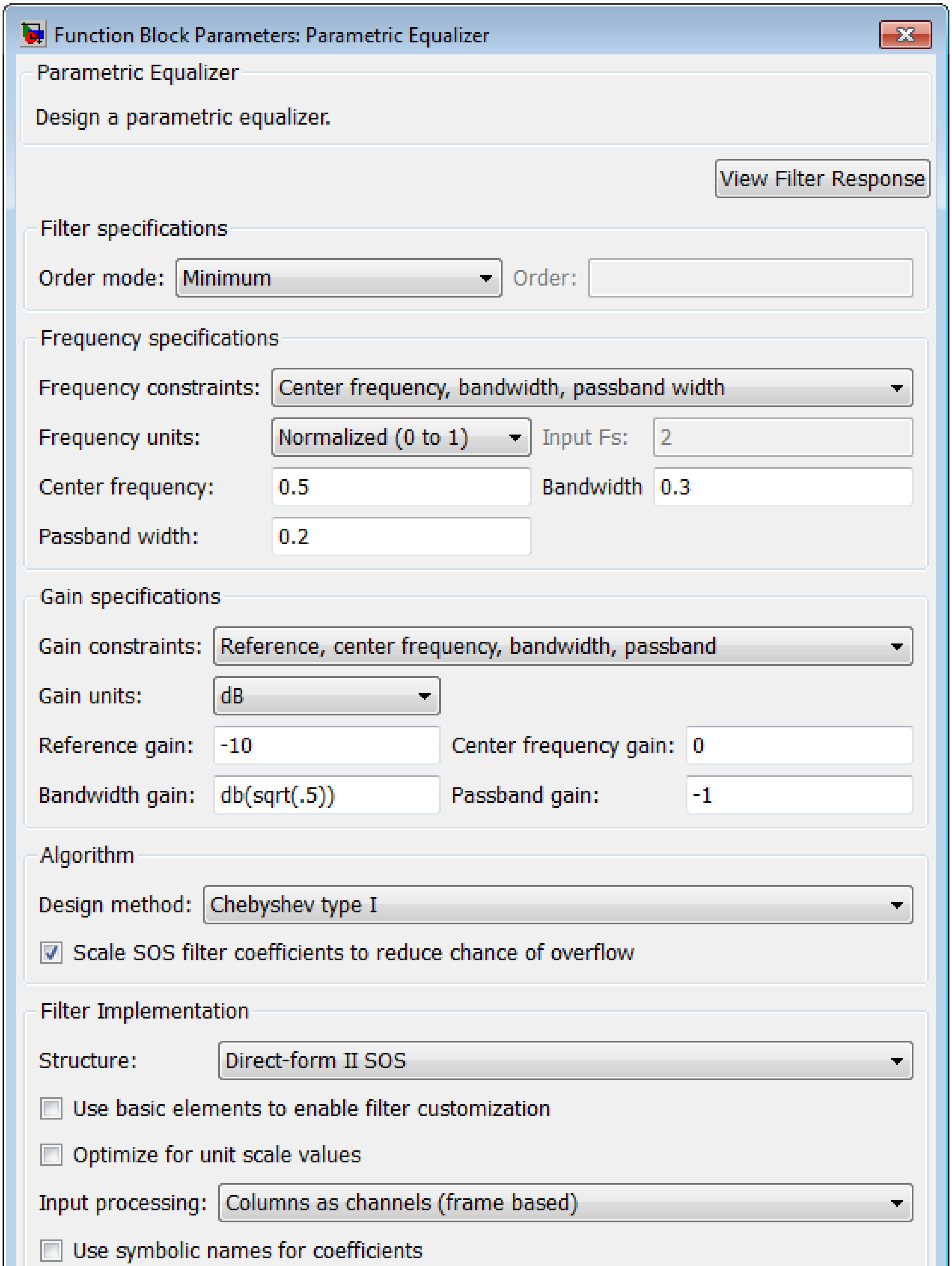
Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See "Parametric Equalizer Filter Design — Main Pane" on page 5-788 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Parameters of this block that do not change filter order or structure are tunable.


Function Block Parameters: Parametric Equalizer
Parametric Equalizer
Design a parametric equalizer.
View Filter Response
Filter specifications
Order mode: Minimum
Order:
Frequency specifications
Frequency constraints: Center frequency, bandwidth, passband width
Frequency units: Normalized (0 to 1)
Input Fs: 2
Center frequency: 0.5
Bandwidth 0.3
Passband width: 0.2
Gain specifications
Gain constraints: Reference, center frequency, bandwidth, passband
Gain units: dB
Reference gain: -10
Center frequency gain: 0
Bandwidth gain: db(sqrt(.5))
Passband gain: -1
Algorithm
Design method: Chebyshev type I
Scale SOS filter coefficients to reduce chance of overflow
Filter Implementation
Structure: Direct-form II SOS
Use basic elements to enable filter customization
Optimize for unit scale values
Input processing: Columns as channels (frame based)
Use symbolic names for coefficients

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

Order mode

Select `Minimum` to design a minimum order filter that meets the design specifications, or `Specify` to enter a specific filter order. The order mode also affects the possible frequency constraints, which in turn limit the gain specifications. For example, if you specify a `Minimum` order filter, the available frequency constraints are:

- `Center frequency, bandwidth, passband width`
- `Center frequency, bandwidth, stopband width`

If you select `Specify`, the available frequency constraints are:

- `Center frequency, bandwidth`
- `Center frequency, quality factor`
- `Shelf type, cutoff frequency, quality factor`
- `Shelf type, cutoff frequency, shelf slope parameter`
- `Low frequency, high frequency`

Order

Specify the filter order. This parameter is enabled only if the **Order mode** is set to `Specify`.

Frequency specifications

Depending on the filter order, the possible frequency constraints change. Once you choose the frequency constraints, the input boxes in this area change to reflect the selection.

Frequency constraints

Select the specification to represent the frequency constraints. The following options are available:

- `Center frequency, bandwidth, passband width` (available for minimum order only)
- `Center frequency, bandwidth, stopband width` (available for minimum order only)
- `Center frequency, bandwidth` (available for a specified order only)
- `Center frequency, quality factor` (available for a specified order only)
- `Shelf type, cutoff frequency, quality factor` (available for a specified order only)
- `Shelf type, cutoff frequency, shelf slope parameter` (available for a specified order only)
- `Low frequency, high frequency` (available for a specified order only)

Frequency units

Select the frequency units from the available drop down list (`Normalized`, `Hz`, `kHz`, `MHz`, `GHz`). If `Normalized` is selected, then the **Input Fs** box is disabled for input.

Input Fs

Enter the input sampling frequency. This input box is disabled for input if `Normalized` is selected in the **Frequency units** input box.

Center frequency

Enter the center frequency in the units specified by the value in **Frequency units**.

Bandwidth

The bandwidth determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Bandwidth gain** in the **Gain specifications** section. By default, the **Bandwidth gain** defaults to `db(sqrt(.5))`, or –3 dB relative to the center frequency. The **Bandwidth** property only applies when the **Frequency constraints** are: `Center frequency, bandwidth, passband width`, `Center frequency, bandwidth, stopband width`, or `Center frequency, bandwidth`.

Passband width

The passband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Passband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is `Center frequency, bandwidth, passband width`.

Stopband width

The stopband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Stopband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is `Center frequency, bandwidth, stopband width`.

Low frequency

Enter the low frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is `Low frequency, high frequency`. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

High frequency

Enter the high frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is `Low frequency, high frequency`. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

Gain Specifications

Depending on the filter order and frequency constraints, the possible gain constraints change. Also, once you choose the gain constraints the input boxes in this area change to reflect the selection.

Gain constraints

Select the specification array to represent gain constraints, and remember that not all of these options are available for all configurations. The following is a list of all available options:

- `Reference, center frequency, bandwidth, passband`
- `Reference, center frequency, bandwidth, stopband`
- `Reference, center frequency, bandwidth, passband, stopband`

- `Reference, center frequency, bandwidth`

Gain units

Specify the gain units either `dB` or `squared`. These units are used for all gain specifications in the dialog box.

Reference gain

The reference gain determines the level to which the filter magnitude attenuates in **Gain units**. The reference gain is a *floor* gain for the filter magnitude response. For example, you may use the reference gain together with the **Center frequency gain** to leave certain frequencies unattenuated (reference gain of 0 dB) while boosting other frequencies.

Bandwidth gain

Specifies the gain in **Gain units** at which the bandwidth is defined. This property applies only when the **Frequency constraints** specification contains a `bandwidth` parameter, or is `Low frequency, high frequency`.

Center frequency gain

Specify the center frequency in **Gain units**

Passband gain

The passband gain determines the level in **Gain units** at which the passband is defined. The passband is determined either by the **Passband width** value, or the **Low frequency** and **High frequency** values in the **Frequency specifications** section.

Stopband gain

The stopband gain is the level in **Gain units** at which the stopband is defined. This property applies only when the **Order mode** is minimum and the **Frequency constraints** are `Center frequency, bandwidth, stopband width`.

Boost/cut gain

The boost/cut gain applies only when the designing a shelving filter. Shelving filters include the `Shelf type` parameter in the **Frequency constraints** specification. The gain in the passband of the shelving filter is increased by **Boost/cut gain** dB from a *floor* gain of 0 dB.

Algorithm

Design method

Select the design method from the drop-down list. Different methods are available depending on the chosen filter constraints.

Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

Filter Implementation

Structure

Specify filter structure. Choose from:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note: The `Inherited (this choice will be removed — see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2007a

Parametric EQ Filter

Model second-order parametric equalizer filter

Note: The Parametric EQ Filter block will be removed from DSP System Toolbox in a future release. Existing instances of the block continue to run. For new code, use the Parametric EQ Filter block from Audio System Toolbox instead.

Library

Filtering / Filter Designs

`dspfdesign`

Description

The Parametric EQ Filter block filters each channel of the input signal over time using a specified center frequency, bandwidth, and peak (dip) gain. This block offers tunable filter design parameters, which enable you to tune the filter characteristics while the simulation is running.

The block designs the filter according to the filter parameters set in the block dialog box. The output port properties, such as datatype, complexity, and dimension, are identical to the input port properties.

Each column of the input signal is treated as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then it is interpreted as a single channel.

This block supports variable-size input, enabling you to change the channel length during simulation. To enable variable-size input, clear the **Inherit sample rate from input** check box. The number of channels must remain constant.

Algorithms

This block brings the capabilities of the dsp.ParametricEQFilter System object to the Simulink environment.

The filter uses a coupled allpass structure to optimize joint computation of the peak and notch response. For information on the algorithms used by the Parametric EQ Filter block, see the "Algorithm" on page 4-1401 section of `dsp.ParametricEQFilter`.

Parameters

Filter specification

Parameters or coefficients used to design the filter, specified as one of the following:

- `Bandwidth and center frequency` (default) — Design the filter using **Filter bandwidth (Hz)**, **Equalizer center frequency (Hz)**, and **Gain (dB)**.
- `Coefficients` — Design the filter using **Bandwidth coefficient**, **Center frequency coefficient**, and **Gain (Linear Units)**.
- `Quality factor and center frequency` — Design the filter using **Equalizer center frequency (Hz)**, **Gain (dB)**, and **Quality factor**.

This parameter is nontunable.

Filter bandwidth (Hz)

Bandwidth of the filter, specified as a finite positive numeric scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter specification** to `Bandwidth and center frequency`. The default is `2205`. This parameter is tunable.

Equalizer center frequency (Hz)

Center frequency of the filter, specified as a finite positive scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter specification** to `Bandwidth and center frequency` or `Quality factor and center frequency`. The default is `11025`. This parameter is tunable.

Gain (dB)

Peak or dip gain of the filter, specified as a real scalar in dB. A value greater than zero corresponds to a peak. A value less than zero corresponds to a dip. This parameter applies when you set **Filter specification** to `Bandwidth and center frequency` or `Quality factor and center frequency`. The default is `6.0206`. This parameter is tunable.

Bandwidth coefficient

Coefficient that determines the filter bandwidth, specified as a finite numeric scalar in the range `[-1 1]`.

- `-1` corresponds to the maximum bandwidth (one-fourth the sample rate of the input signal).
- `1` corresponds to the minimum bandwidth (0 Hz, that is, an allpass filter).

This parameter applies when you set **Filter specification** to `Coefficients`. The default is `0.72654`. This parameter is tunable.

Center frequency coefficient

Coefficient that determines the center frequency of the filter, specified as a finite numeric scalar in the range `[-1 1]`.

- `-1` corresponds to the minimum center frequency (`0` Hz).
- `1` corresponds to the maximum center frequency (half the sample rate of the input signal).

This parameter applies when you set **Filter specification** to `Coefficients`. The default is `0`, which corresponds to one-fourth the sample rate of the input signal. This parameter is tunable.

Gain (Linear Units)

Peak or dip gain of the filter, specified as a real positive scalar in linear units. A value greater than one boosts the input signal. A value less than one attenuates the input signal. This parameter applies when you set **Filter specification** to `Coefficients`. The default is `2`. This parameter is tunable.

Quality factor

Quality factor of the filter, specified as a real positive scalar. The quality factor is defined as **Equalizer center frequency (Hz) / Filter bandwidth (Hz)**. A higher quality factor corresponds to a narrower peak or dip. This parameter applies when you set **Filter specification** to `Quality factor and center frequency`. The default is `5`. This parameter is tunable.

Inherit sample rate from input

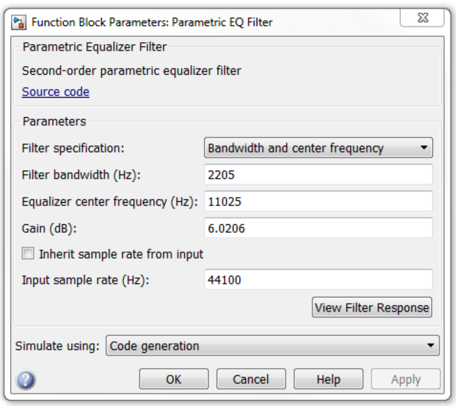
When you select this check box, the block's sample rate is computed as *N*/*Ts*, where *N* is the frame size of the input signal, and *Ts* is the sample time of the input signal. When you clear this check box, the block sample rate is the value specified in **Input sample rate (Hz)**. By default, this check box is selected.

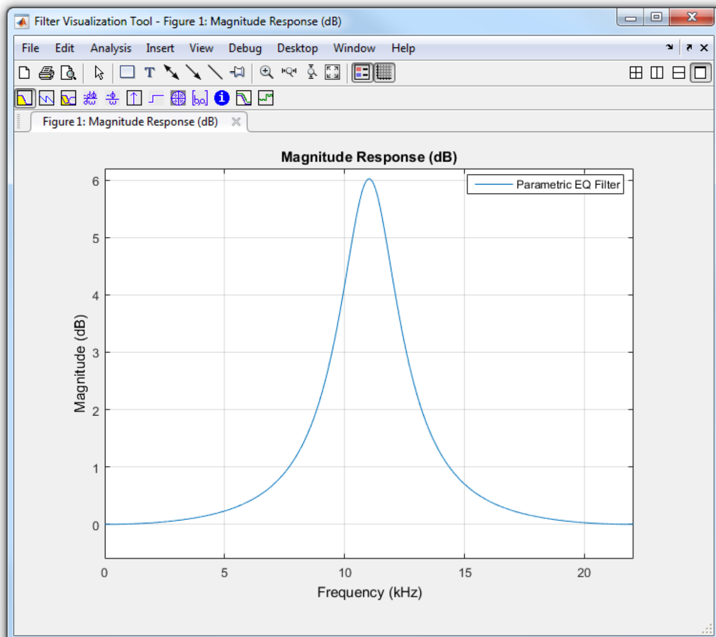
Input sample rate (Hz)

Sample rate of the input signal, specified as a positive scalar value. The default is `44100`. This parameter applies when you clear the **Inherit sample rate from input** check box. This parameter is nontunable.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the Parametric EQ Filter. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.





To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing* . Upper Saddle River, NJ: Prentice-Hall, 1996.

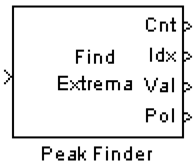
See Also

dsp.ParametricEQFilter DSP System Toolbox

Introduced in R2015a

Peak Finder

Determine whether each value of input signal is local minimum or maximum



Library

Signal Operations

`dspsigops`

Description

The Peak Finder block counts the number of local extrema in each column of the real-valued input signal. The block outputs the number of local extrema at the Cnt port. You can also configure the block to output the extrema indices, the extrema values, and a binary indicator of whether or not the extrema are maxima or minima.

To qualify as an extrema, a point has to be larger (or smaller) than both of its neighboring points. Thus, end points are never considered extrema.

If you select the **Output peak indices** check box, the Idx port appears on the block. The block outputs the zero-based extrema indices at the Idx port. If you select the **Output peak values** check box, the Val port appears on the block. The block outputs the extrema values at the Val port. If you select either of these check boxes and set the **Peak type(s)** to `Maxima and Minima`, the Pol port also appears on the block. If the signal value is a maximum, the block outputs a 1 at the Pol ("Polarity") port. If the signal value is a minimum, the block outputs a 0 at the Pol port.

Use the **Maximum number of peaks to find** parameter to specify how many extrema to look for in each input signal. The block stops searching the input signal once this maximum number of extrema has been found.

If you select the **Ignore peaks within threshold of neighboring values** check box, the block no longer detects low-amplitude peaks. This feature allows the block to ignore noise within a threshold value that you define. Enter a threshold value for the **Threshold** parameter. Now, the current value is a maximum if (current – previous) > threshold and (current – next) > threshold. The current value is a minimum if (current – previous) < –threshold and (current – next) < –threshold.

Examples

Example 1

Consider the input vector

`[9 6 10 3 4 5 0 12]`

The table below shows the analysis made by the Peak Finder block. Note that the first and last input signal values are not considered:

Previous, current, and next values	9 6 10	6 10 3	10 3 4	3 4 5	4 5 0	5 0 12
Current value if it is an extremum	6	10	3	—	5	0
Index of current value if it is an extremum	1	2	3	—	5	6
Polarity of current value if it is an extremum	0	1	0	—	1	0

Therefore, for this example the outputs at the block ports are

Cnt: `5`

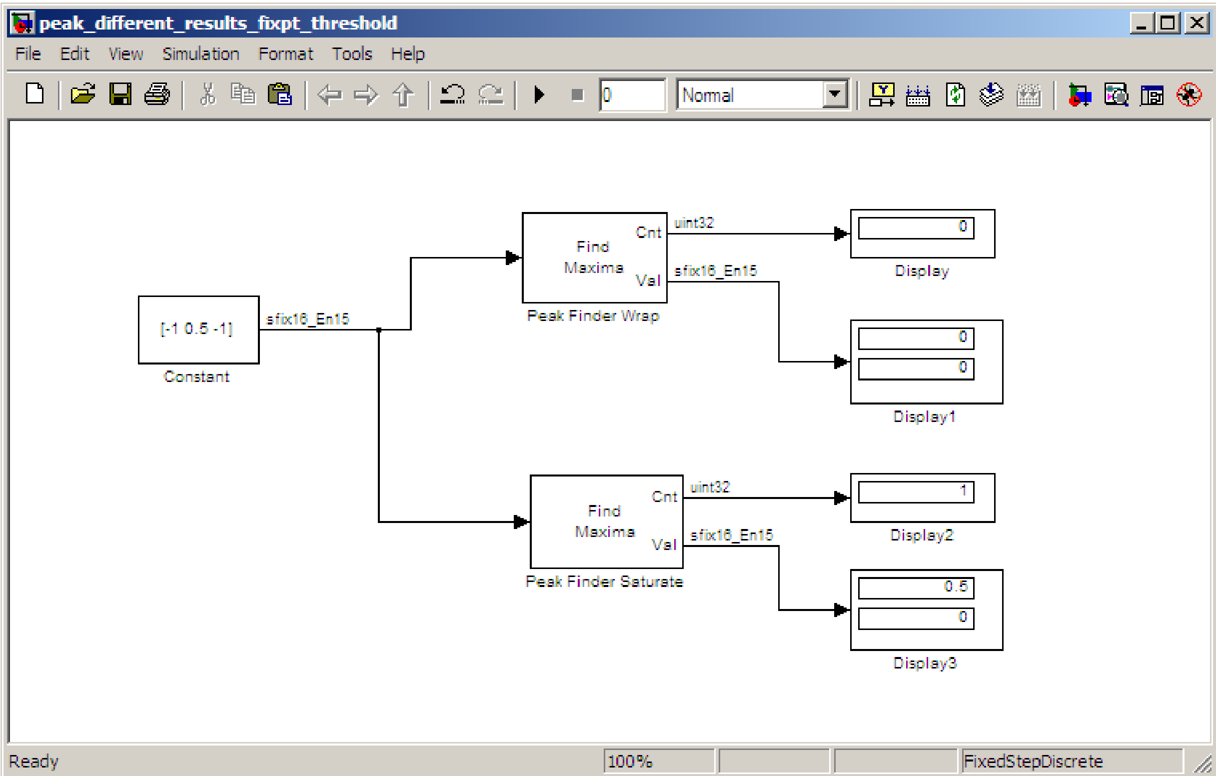
Idx: `[1 2 3 5 6]`

Val: `[6 10 3 5 0]`

Pol: `[0 1 0 1 0]`

Example 2

Note that the **Overflow mode** parameter can affect the output of the block when the input is fixed point. Consider the following model:



In this model, the settings in the Constant block are

- **Constant value** — `[-1 0.5 -1]`
- **Interpret vector parameters as 1-D** — not selected
- **Sampling mode** — `Sample based`
- **Sample time** — `1`
- **Output data type** — `<data type expression>`
- **Mode** — `Fixed point`
- **Sign** — `Signed`
- **Scaling** — `Binary point`
- **Word length** — `16`
- **Fraction length** — `15`

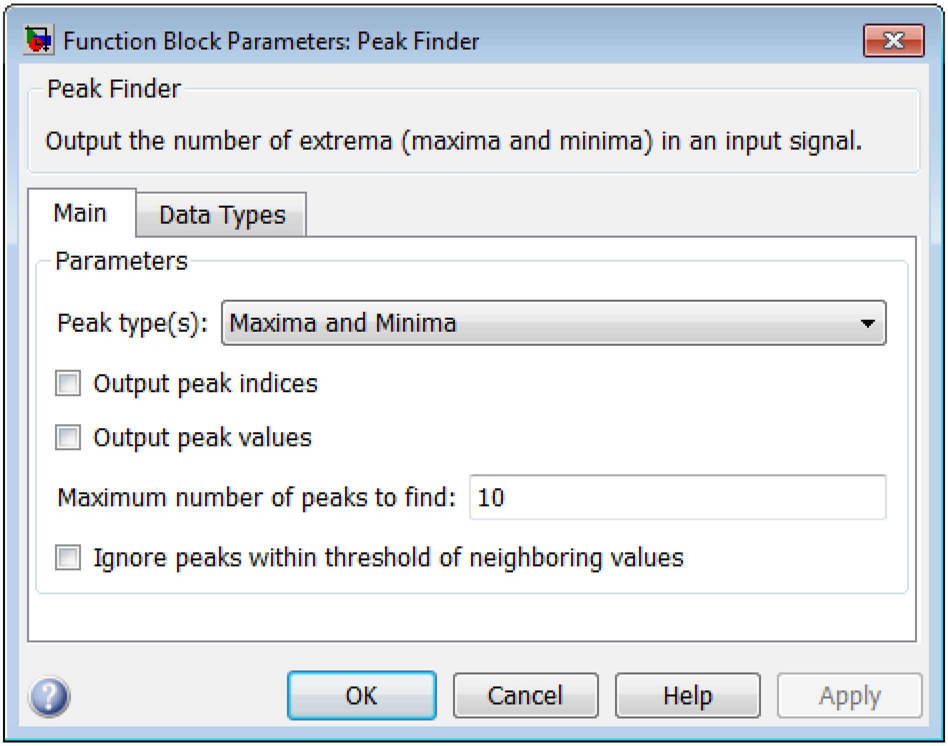
The settings in the Peak Finder blocks are

- **Peak type(s)** — `Maxima`
- **Output peak indices** — not selected
- **Output peak values** — selected
- **Maximum number of peaks to find** — `2`
- **Ignore peaks within threshold of neighboring values** — selected
- **Threshold** — `0.25`
- **Overflow mode** — `Wrap` for Peak Finder Wrap, `Saturate` for Peak Finder Saturate

Setting the **Overflow mode** parameter of the Peak Finder Wrap block to `Wrap` causes the calculations (current – previous) > threshold and (current – next) > threshold to wrap on overflow, thereby causing the maximum to be missed.

Dialog Box

The **Main** pane of the Peak Finder block dialog appears as follows.



Peak type(s)

Specify whether you are looking for maxima, minima, or both.

Output peak indices

Select this check box if you want the block to output the extrema indices at the Idx port.

Output peak values

Select this check box if you want the block to output the extrema values at the Val port.

Maximum number of peaks to find

Enter the number of extrema to look for in each input signal. The block stops searching the input signal for extrema once the maximum number of extrema has been found. The value of this parameter must be an integer greater than or equal to one.

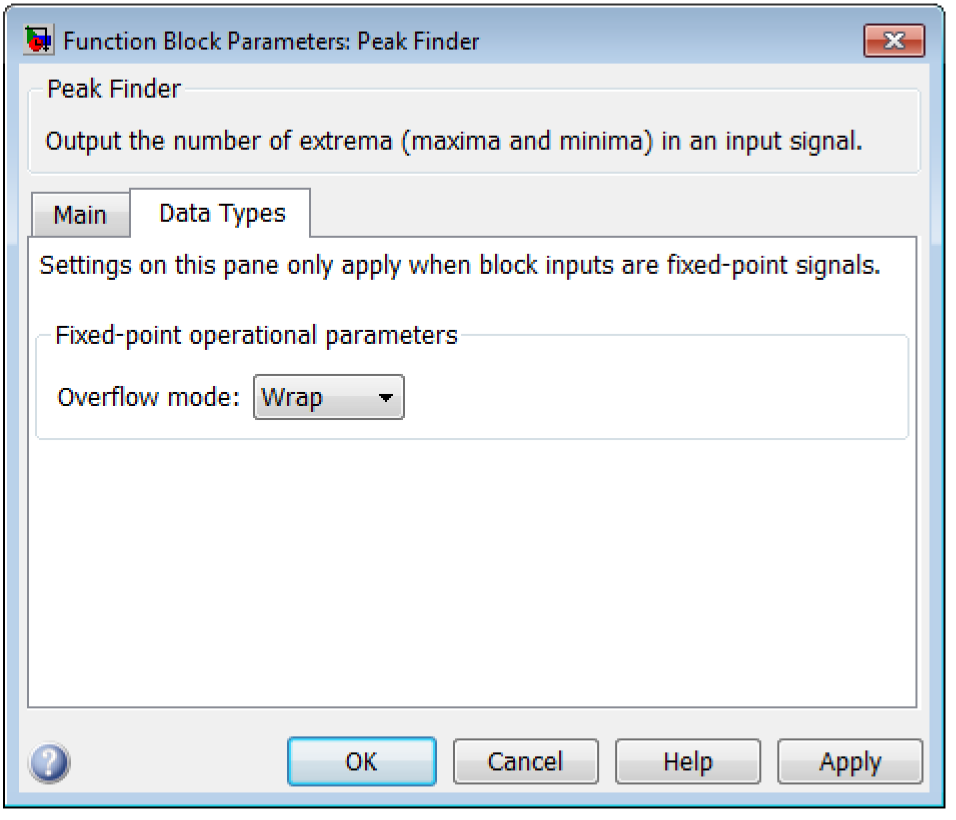
Ignore peaks within threshold of neighboring values

Select this check box if you want to eliminate the detection of peaks whose amplitudes are within a specified threshold of neighboring values.

Threshold

Enter your threshold value. This parameter appears if you select the **Ignore peaks within threshold of neighboring values** check box.

When you select the **Ignore peaks within threshold of neighboring values** check box, the **Data Types** pane of the Peak Finder block appears as follows.



Overflow mode

Select the overflow mode to be used when block inputs are fixed point.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Cnt	• 32-bit unsigned integers
Idx	• 32-bit unsigned integers
Val	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Pol	• Boolean

See Also

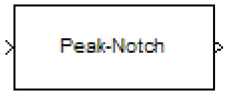
See Also

Blocks
`Maximum` | `Minimum`

Introduced before R2006a

Peak-Notch Filter

Design peak or notch filter



Note: The Peak-Notch Filter block has been replaced by the `Notch-Peak Filter` block. Existing instances of the Peak-Notch Filter block will continue to operate. For new models, use the Notch-Peak Filter block.

Library

Filtering / Filter Designs

`dspfdesign`

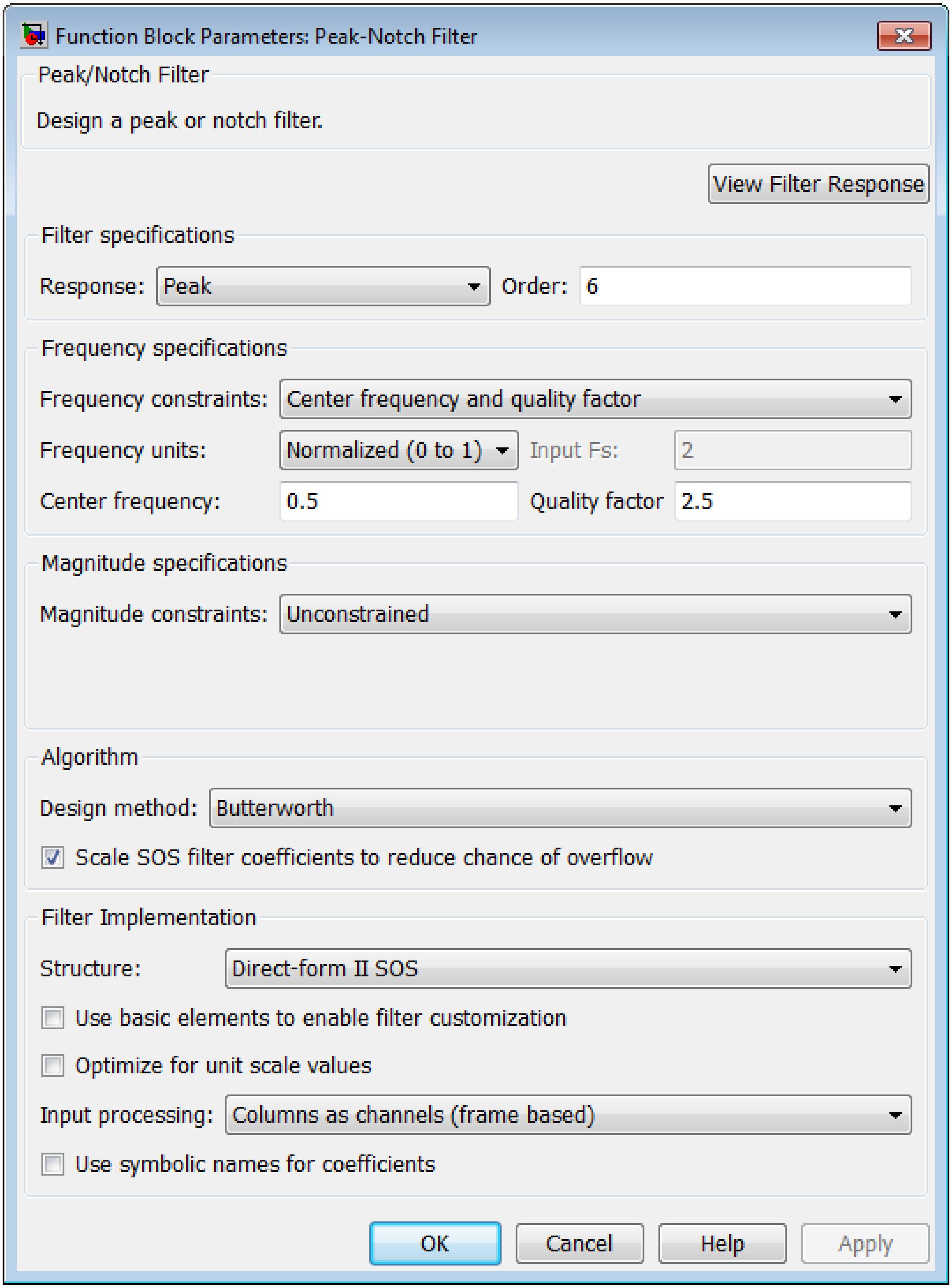
Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See "Peak/Notch Filter Design — Main Pane" on page 5-793 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Parameters of this block that do not change filter order or structure are tunable.


Function Block Parameters: Peak-Notch Filter
Peak/Notch Filter
Design a peak or notch filter.
View Filter Response
Filter specifications
Response: Peak
Order: 6
Frequency specifications
Frequency constraints: Center frequency and quality factor
Frequency units: Normalized (0 to 1)
Input Fs: 2
Center frequency: 0.5
Quality factor 2.5
Magnitude specifications
Magnitude constraints: Unconstrained
Algorithm
Design method: Butterworth
Scale SOS filter coefficients to reduce chance of overflow
Filter Implementation
Structure: Direct-form II SOS
Use basic elements to enable filter customization
Optimize for unit scale values
Input processing: Columns as channels (frame based)
Use symbolic names for coefficients
OK
Cancel
Help
Apply

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this area you can specify whether you want to design a peaking filter or a notching filter, as well as the order of the filter.

Response

Select `Peak` or `Notch` from the drop-down list. The rest of the parameters that specify are equivalent for either filter type.

Order

Enter the filter order. The order must be even.

Frequency Specifications

This group of parameters allows you to specify frequency constraints and units.

Frequency Constraints

Select the frequency constraints for filter specification. There are two choices as follows:

- `Center frequency and quality factor`
- `Center frequency and bandwidth`

Frequency units

The frequency units are normalized by default. If you specify units other than normalized, the block assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: `Normalized (0 to 1)`, `Hz`, `kHz`, `MHz`, `GHz`.

Input Fs

This input box is enabled if **Frequency units** other than `Normalized (0 to 1)` are specified. Enter the input sampling frequency.

Center frequency

Enter the center frequency in the units specified previously.

Quality Factor

This input box is enabled only when `Center frequency and quality factor` is chosen for the **Frequency Constraints**. Enter the quality factor.

Bandwidth

This input box is enabled only when `Center frequency and bandwidth` is chosen for the **Frequency Constraints**. Enter the bandwidth.

Magnitude Specifications

This group of parameters allows you to specify the magnitude constraints, as well as their values and units.

Magnitude Constraints

Depending on the choice of constraints, the other input boxes are enabled or disabled. Select from four magnitude constraints available:

- `Unconstrained`
- `Passband ripple`
- `Stopband attenuation`
- `Passband ripple and stopband attenuation`

Magnitude units

Select the magnitude units: either `dB` or `squared`.

Apass

This input box is enabled if the magnitude constraints selected are `Passband ripple` or `Passband ripple and stopband attenuation`. Enter the passband ripple.

Astop

This input box is enabled if the magnitude constraints selected are `Stopband attenuation` or `Passband ripple and stopband attenuation`. Enter the stopband attenuation.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists all design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter the methods available to design filters changes as well.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Filter Implementation

Structure

Specify filter structure. Choose from:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note: The `Inherited (this choice will be removed – see release notes)` option will be removed in a future release. See “Frame-Based Processing” in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

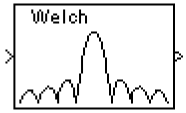
Supported Data Types

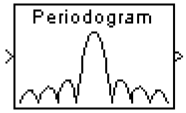
Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2007a

Periodogram

Power spectral density or mean-square spectrum estimate using periodogram method





Library

Estimation / Power Spectrum Estimation

`dspspect3`

Description

The Periodogram block estimates the power spectral density (PSD) or mean-square spectrum (MSS) of the input. It does so by using the periodogram method and Welch's averaged, modified periodogram method. The block averages the squared magnitude of the FFT computed over windowed sections of the input. It then normalizes the spectral average by the square of the sum of the window samples. See "Periodogram" (Signal Processing Toolbox) and "Welch's Method" (Signal Processing Toolbox) for more information.

The block treats M-by-N frame-based matrix input and M-by-N sample-based matrix input as M sequential time samples from N independent channels. The block computes a separate estimate for each of the N independent channels and generates an N_{fft}-by-N matrix output.

Each column of the output matrix contains the estimate of the power spectral density of the corresponding input column at N_{fft} equally spaced frequency points. The frequency points are in the range $[0,F_s)$, where F_s is the sampling frequency of the signal. The block always outputs sample-based data.

Parameters

Measurement

Specify the type of measurement for the block to perform: `Power spectral density` or `Mean-square spectrum`. Tunable (Simulink).

Window

Select the type of window to apply. See the `Window Function` block reference page for more details. Tunable (Simulink).

Stopband attenuation in dB

Enter the level, in decibels (dB), of stopband attenuation, R_s, for the Chebyshev window. This parameter becomes visible if, for the **Window** parameter, you choose `Chebyshev`. Tunable (Simulink).

Beta

Enter the β parameter for the Kaiser window. This parameter becomes visible if, for the **Window** parameter, you chose `Kaiser`. Increasing **Beta** widens the mainlobe and decreases the amplitude of the sidelobes in the displayed frequency magnitude response. Tunable (Simulink). See the `Window Function` block reference page for more details.

Window sampling

From the list, choose `Symmetric` or `Periodic`. See the `Window Function` block reference page for more details.Tunable (Simulink).

FFT implementation

Set this parameter to `FFTW` to support an arbitrary length input signal. The block restricts generated code with FFTW implementation to MATLAB host computers.

Set this parameter to `Radix-2` for bit-reversed processing, fixed or floating-point data, or for portable C-code generation using the Simulink Coder. The first dimension *M*, of the input matrix must be a power of two. To work with other input sizes, use the `Pad` block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation.

Set this parameter to `Auto` to let the block choose the FFT implementation. For non-power-of-two transform lengths, the block restricts generated code to MATLAB host computers.

Inherit FFT length from input dimensions

When you select this check box, the block uses the input frame size as the number of data points, N_{fft}, on which to perform the FFT. To specify the number of points on which to perform the FFT, clear the **Inherit FFT length from estimation order** check box. You can then specify a power of two FFT length using the **FFT length** parameter.

FFT length

Enter the number of data points on which to perform the FFT, N_{fft}. When N_{fft} is larger than the input frame size, the block zero-pads each frame as needed. When N_{fft} is smaller than the input frame size, the block wraps each frame as needed. This parameter becomes visible only when you clear the **Inherit FFT length from input dimensions** check box.

When you set the **FFT implementation** parameter to `Radix-2`, this value must be a power of two.

Number of spectral averages

Specify the number of spectra to average. When you set this value to `1`, the block computes the periodogram of the input. When you set this value greater `1`, the block implements "Welch's Method" (Signal Processing Toolbox) to compute a modified periodogram of the input.

Inherit sample time from input

If you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must hold:

- The input to the block is the original signal, with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

Sample time of original time series

Specify the sample time of the original time-domain signal. This parameter becomes visible only when you clear the **Inherit sample time from input** check box.

Example

The dspstfft example provides an illustration of using the Periodogram and Matrix Viewer blocks to create a spectrogram. The dspsacomp example compares the Periodogram block with several other spectral estimation methods.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

References

[1] FFTW (http://www.fftw.org)

[2] Frigo, M. and S. G. Johnson, “FFTW: An Adaptive Software Architecture for the FFT,”*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

[3] Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[4] Orfanidis, S. J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[5] Proakis, J. and D. Manolakis. *Digital Signal Processing.* 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- When the following conditions apply, the executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB:

  - **FFT implementation** is set to `FFTW`.
  - **Inherit FFT length from input dimensions** is cleared, and **FFT length** is set to a value that is not a power of two.

  Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "How To Run a Generated Executable Outside MATLAB".
- When the FFT length is a power of two, you can generate standalone C and C++ code from this block.

See Also

See Also

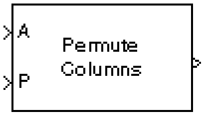
Blocks
`Burg Method` | `Inverse Short-Time FFT` | `Magnitude FFT` | `Short-Time FFT` | `Spectrum Analyzer` | `Window Function` | `Yule-Walker Method`

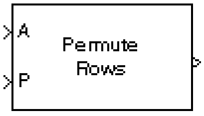
Topics
"Spectral Analysis"

Introduced before R2006a

Permute Matrix

Reorder matrix rows or columns





Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

Description

The Permute Matrix block reorders the rows or columns of M-by-N input matrix `A` as specified by indexing input `P`.

When the **Permute** parameter is set to `Rows`, the block uses the rows of `A` to create a new matrix with the same column dimension. Input `P` is a length-L vector whose elements determine where each row from `A` should be placed in the L-by-N output matrix.

```
% Equivalent MATLAB code
y = [A(P(1),:) ; A(P(2),:) ; A(P(3),:) ; ... ; A(P(end),:)]
```

For row permutation, the block treats length-M unoriented vector input at the `A` port as an M-by-1 matrix.

When the **Permute** parameter is set to `Columns`, the block uses the columns of `A` to create a new matrix with the same row dimension. Input `P` is a length-L vector whose elements determine where each column from `A` should be placed in the M-by-L output matrix.

```
% Equivalent MATLAB code
y = [A(:,P(1)) A(:,P(2)) A(:,P(3)) ... A(:,P(end))]
```

For column permutation, the block treats length-N unoriented vector input at the A port as a 1-by-N matrix.

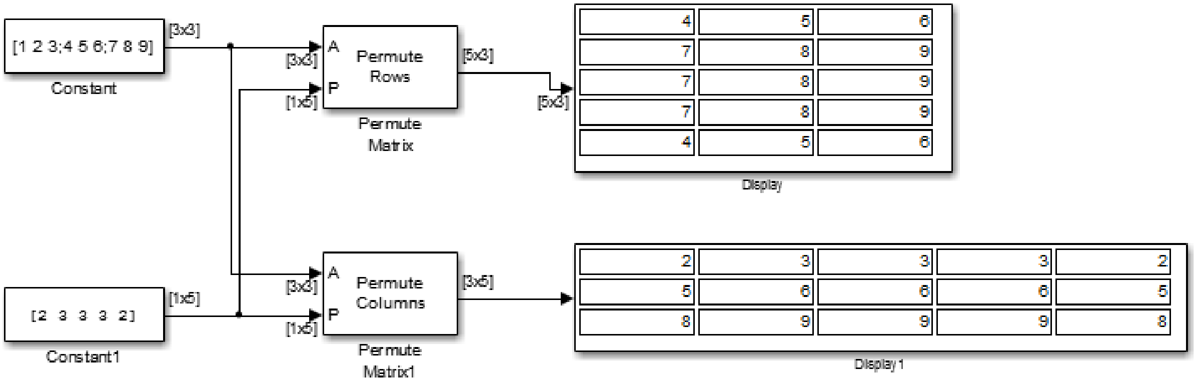
When an index value in input P references a nonexistent row or column of matrix A, the block reacts with the behavior specified by the **Invalid permutation index** parameter. The following options are available:

- `Clip index` — Clip the index to the nearest valid value (1 or M for row permutation, and 1 or N for column permutation), and *do not* issue an alert. Example: For a 3-by-7 input matrix, a column index of 9 is clipped to 7, and a row index of -2 is clipped to 1.
- `Clip and warn` — Display a warning message in the MATLAB command window, and clip the index as described above.
- `Generate error` — Display an error dialog box and terminate the simulation.

When length of the permutation vector P is not equal to the number of rows or columns of the input matrix A, you can choose to get an error dialog box and terminate the simulation by selecting **Error when length of P is not equal to Permute dimension size**.

Examples

In the model below, the top Permute Matrix block places the second row of the input matrix in both the first and fifth rows of the output matrix, and places the third row of the input matrix in the three middle rows of the output matrix. The bottom Permute Matrix block places the second column of the input matrix in both the first and fifth columns of the output matrix, and places the third column of the input matrix in the three middle columns of the output matrix.



As shown in the example above, rows and columns of A can appear any number of times in the output, or not at all.

Parameters

Permute

Method of constructing the output matrix; by permuting rows or columns of the input.

Index mode

When set to `One-based`, a value of 1 in the permutation vector P refers to the first row or column of the input matrix A. When set to `Zero-based`, a value of 0 in P refers to the first row or column of A.

Invalid permutation index

Response to an invalid index value. Tunable (Simulink).

Error when length of P is not equal to Permute dimension size

Option to display an error dialog box and terminate the simulation when the length of the permutation vector P is not equal to the number of rows or columns of the input matrix A.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated
P	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated

See Also

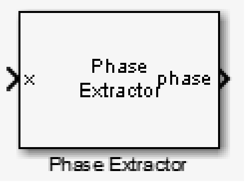
Submatrix	DSP System Toolbox
Variable Selector	DSP System Toolbox
`permute`	MATLAB

See “Reorder Channels in Multichannel Signals” for related information.

Introduced before R2006a

Phase Extractor

Extract the unwrapped phase of a complex input



Library

Signal Operations

`dspsigops`

Description

The Phase Extractor block extracts the unwrapped phase of a complex input. The input can be a vector or matrix. For 2D inputs, the block treats each column as an independent channel. The first dimension is the length of the channel. The second dimension is the number of channels. The block treats 1D inputs as one channel.

The block preserves the input size and dimension, and the output port rate equals the input port rate.

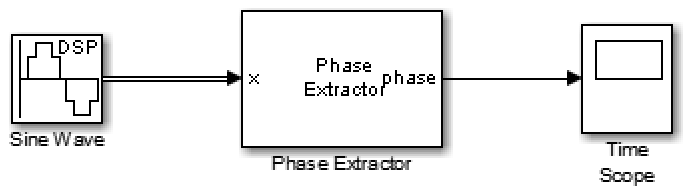
Examples

This example shows how to use the Phase Extractor block to extract the phase of a sign wave. The DSP Sine Wave block represents the system input signal. Set the DSP Sine Wave block parameters to the following:

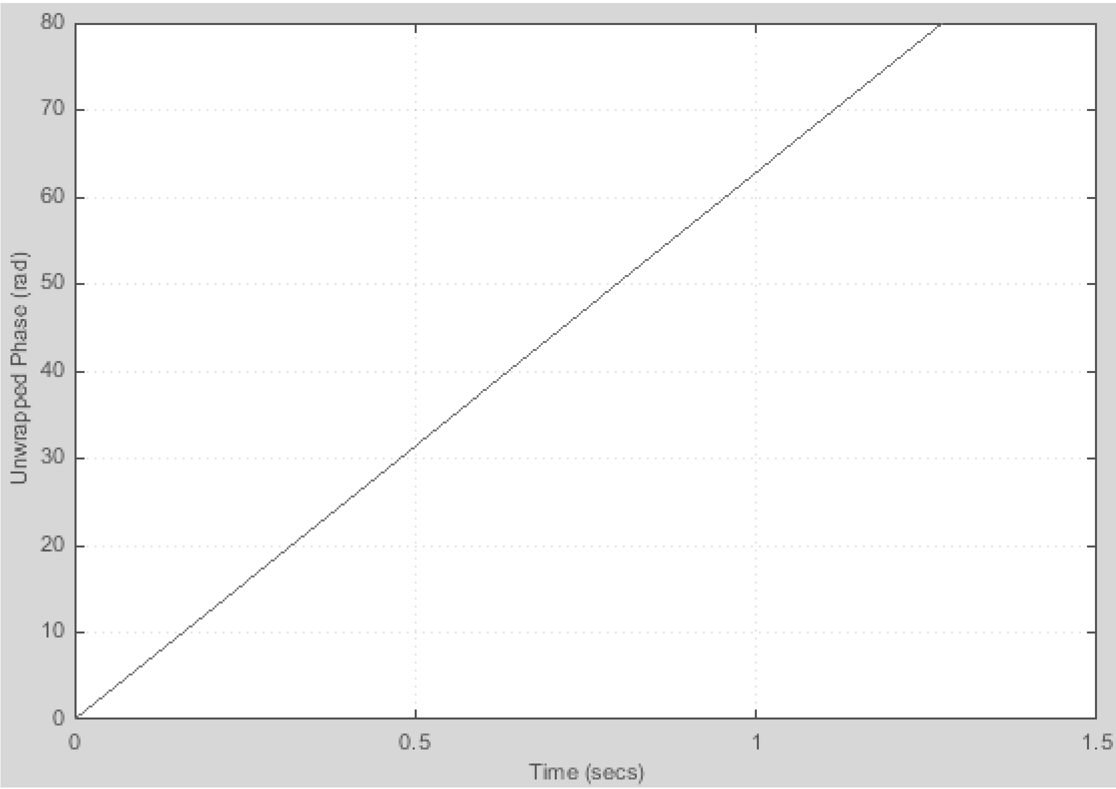
- **Frequency** set to `10` Hz
- **Sample mode** set to `Discrete`

- **Output complexity** set to `Complex`
- **Sample time** set to `1/1000`
- **Sample per frame** set to `128`

Do not select the Phase Extractor block parameter **Unwrap phase only within the frame**.



The Time Scope block displays the extracted phase.



Parameters

Unwrap phase only within the frame

When you clear this check box, the block ignores boundaries between the input frames. When you select this check box, the block treats each frame of input data independently, and resets the initial phase value for each new input frame.

Simulate using

Select the simulation type from the following:

- `Code generation`
- `Interpreted execution`

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

See Also

dsp.PhaseExtractor	DSP System Toolbox

Algorithm

Consider an input frame of length N:

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix}$$

The `step` method acts on this frame and produces this output:

$$\begin{pmatrix} \Phi_1 \\ \Phi_2 \\ \vdots \\ \Phi_N \end{pmatrix}$$

where:

$$\Phi_i = \Phi_{i-1} + \text{angle}(x_{i-1}^{*} x_i)$$

Here, i runs from 1 to N. The `angle` function returns the phase angle in radians.

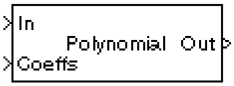
If the input signal consists of multiple frames:

- If you set `TreatFramesIndependently` to `true`, the `step` method treats each frame independently. Therefore, in each frame, the `step` method calculates the phase using the preceding formula where:
  - Φ_0 is 0.
  - x_0 is 1.
- If you set `TreatFramesIndependently` to `false`, the `step` method ignores boundaries between frames. Therefore, in each frame, the `step` method calculates the phase using the preceding formula where:
  - Φ_0 is the last unwrapped phase from the previous frame.
  - x_0 is the last sample from the previous frame.

Introduced in R2014b

Polynomial Evaluation

Evaluate polynomial expression



Library

Math Functions / Polynomial Functions

`dsppolyfun`

Description

The Polynomial Evaluation block applies a polynomial function to the real or complex input at the `In` port.

```
y = polyval(u)      % Equivalent MATLAB code
```

The Polynomial Evaluation block performs these types of operation more efficiently than the equivalent construction using Simulink Sum and Math Function blocks.

When you select the **Use constant coefficients** check box, you specify the polynomial expression in the **Constant coefficients** parameter. When you do not select **Use constant coefficients**, a variable polynomial expression is specified by the input to the `Coeffs` port. In both cases, the polynomial is specified as a vector of real or complex coefficients in order of descending exponents.

The table below shows some examples of the block's operation for various coefficient vectors.

Coefficient Vector	Equivalent Polynomial Expression
`[1 2 3 4 5]`	$y = u^4 + 2u^3 + 3u^2 + 4u + 5$
`[1 0 3 0 5]`	$y = u^4 + 3u^2 + 5$

Coefficient Vector	Equivalent Polynomial Expression
`[1 2+i 3 4-3i 5i]`	$y = u^4 + (2+i)u^3 + 3u^2 + (4-3i)u + 5i$

Each element of a vector or matrix input to the `In` port is processed independently, and the output size is the same as the input.

Parameters

Use constant coefficients

Select to enable the **Constant coefficients** parameter and disable the `Coeffs` input port.

Constant coefficients

Specify the vector of polynomial coefficients to apply to the input, in order of descending exponents. This parameter is enabled when you select the **Use constant coefficients** check box.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

Least Squares Polynomial Fit	DSP System Toolbox
Math Function	Simulink
Sum	Simulink
`polyval`	MATLAB

Introduced before R2006a

Polynomial Stability Test

Use Schur-Cohn algorithm to determine whether all roots of input polynomial are inside unit circle

Library

Math Functions / Polynomial Functions

`dsppolyfun`

Description

The Polynomial Stability Test block uses the Schur-Cohn algorithm to determine whether all roots of a polynomial are within the unit circle.

```
y = all(abs(roots(u)) < 1)        % Equivalent MATLAB code
```

Each column of the M-by-N input matrix u contains M coefficients from a distinct polynomial,

$$f(x) = u_1 x^{M-1} + u_2 x^{M-2} + \ldots + u_M$$

arranged in order of descending exponents, u_1, u_2, ..., u_M. The polynomial has order M-1 and positive integer exponents.

Inputs to the block represent the polynomial coefficients as shown in the previous equation. The block always treats length-M unoriented vector input as an M-by-1 matrix.

The output is a 1-by-N matrix with each column containing the value `1` or `0`. The value `1` indicates that the polynomial in the corresponding column of the input is stable; that is, the magnitudes of all solutions to $f(x) = 0$ are less than 1. The value `0` indicates that the polynomial in the corresponding column of the input might be unstable; that is, the magnitude of at least one solution to $f(x) = 0$ is greater than or equal to 1.

Applications

This block is most commonly used to check the pole locations of the denominator polynomial, $A(z)$, of a transfer function, $H(z)$.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}}{a_1 + a_2 z^{-1} + \ldots + a_n z^{-(n-1)}}$$

The poles are the n-1 roots of the denominator polynomial, $A(z)$. When any poles are located outside the unit circle, the transfer function $H(z)$ is unstable. As is typical in DSP applications, the transfer function above is specified in descending powers of z^{-1} rather than z.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean — Block outputs are always Boolean.

See Also

`Least Squares Polynomial Fit`	DSP System Toolbox
`Polynomial Evaluation`	DSP System Toolbox
`polyfit`	MATLAB

Introduced before R2006a

Pseudoinverse

Compute Moore-Penrose pseudoinverse of matrix

Library

Math Functions / Matrices and Linear Algebra / Matrix Inverses

`dspinverses`

Description

The Pseudoinverse block computes the Moore-Penrose pseudoinverse of input matrix A.

```
[U,S,V] = svd(A,0)       % Equivalent MATLAB code
```

The pseudoinverse of A is the matrix $A^{\dagger}$ such that

$$A^{\dagger} = VS^{\dagger}U^{*}$$

where U and V are orthogonal matrices, and S is a diagonal matrix. The pseudoinverse has the following properties:

- $AA^{\dagger} = (AA^{\dagger})^{*}$
- $A^{\dagger}A = (A^{\dagger}A)^{*}$
- $AA^{\dagger}A = A$
- $A^{\dagger}AA^{\dagger} = A^{\dagger}$

Parameters

Show error status port

Select to enable the E output port, which reports a failure to converge. The possible values you can receive on the port are:

- `0` — The pseudoinverse calculation converges.
- `1` — The pseudoinverse calculation does not converge.

If the pseudoinverse calculation fails to converge, the output at port X is an undefined matrix of the correct size.

Simulate using

Type of simulation to run. You can set this parameter to:

- `Interpreted execution` (default)

  Simulate model using the MATLAB interpreter. This option shortens startup time. For this block, the simulation speed in this mode is faster than in `Code generation`.
- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the simulation speed increases with subsequent simulations.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
X	• Double-precision floating point • Single-precision floating point
E	• Boolean

See Also

`Cholesky Inverse`	DSP System Toolbox
`LDL Inverse`	DSP System Toolbox
`LU Inverse`	DSP System Toolbox
`Singular Value Decomposition`	DSP System Toolbox
`inv`	MATLAB

See “Matrix Inverses” for related information.

Introduced before R2006a

Pulse Shaping Filter

Design pulse shaping filter

Note: The Pulse Shaping Filter block has been removed from DSP System Toolbox block library. Existing instances of the Pulse Shaping Filter block will continue to operate. For new models, use the Raised Cosine Receive Filter and Raised Cosine Transmit Filter blocks from the Communications System Toolbox library. These blocks replace the functionality of Pulse Shaping Filter block, when **Filter Type** is set to `Decimator` and `Interpolator`, respectively.

Library

Filtering / Filter Designs

`dspfdesign`

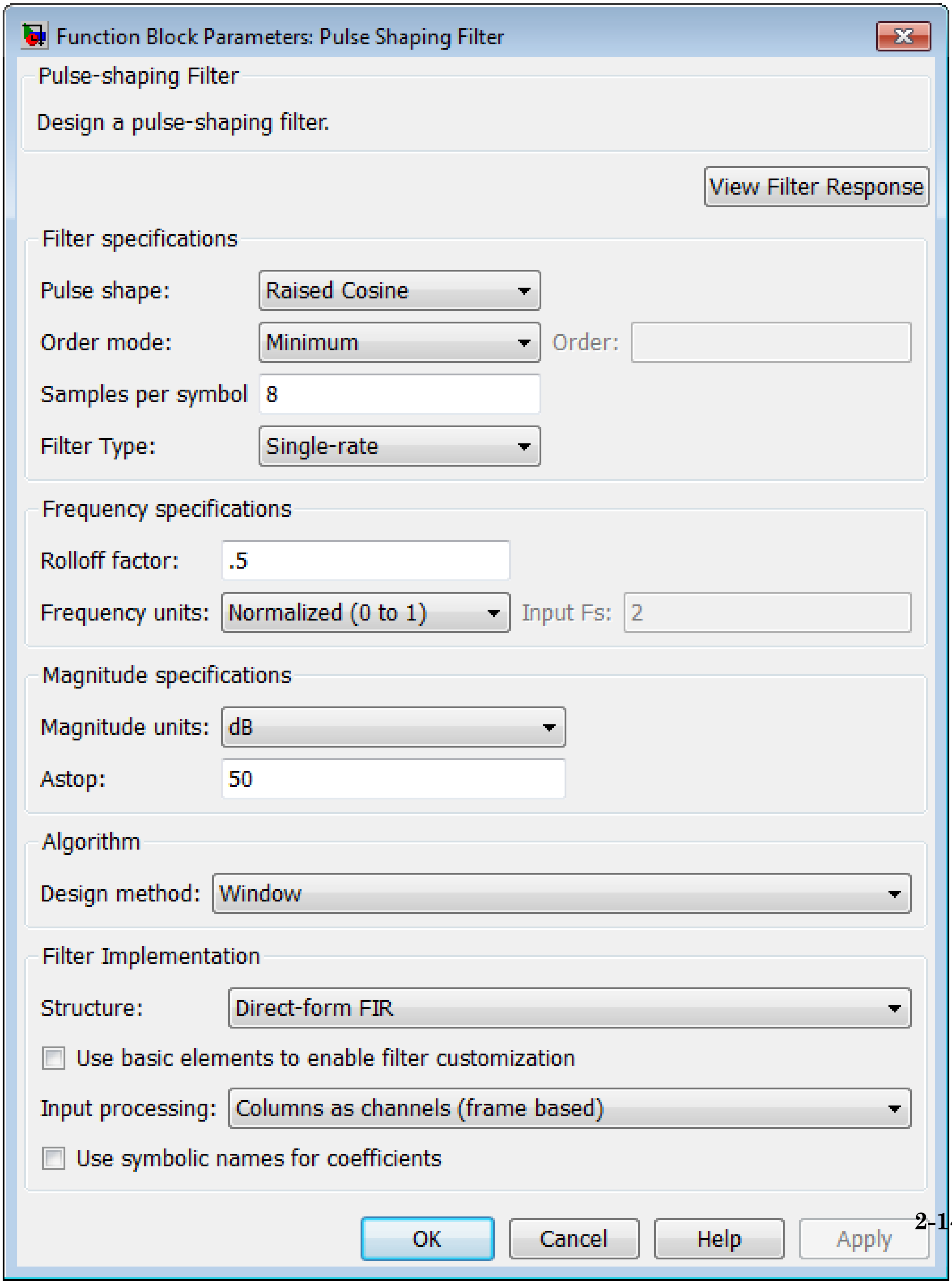
Description

This block brings the filter design capabilities of the `filterBuilderfilterBuilder` function to the Simulink environment.

Dialog Box

See "Pulse-shaping Filter Design —Main Pane" on page 5-797 for more information about the parameters of this block. The **Data Types** and **Code** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Parameters of this block that do not change filter order or structure are tunable.


Function Block Parameters: Pulse Shaping Filter
Pulse-shaping Filter
Design a pulse-shaping filter.
View Filter Response
Filter specifications
Pulse shape:
Raised Cosine
Order mode:
Minimum
Order:
Samples per symbol
8
Filter Type:
Single-rate
Frequency specifications
Rolloff factor:
.5
Frequency units:
Normalized (0 to 1)
Input Fs:
2
Magnitude specifications
Magnitude units:
dB
Astop:
50
Algorithm
Design method:
Window
Filter Implementation
Structure:
Direct-form FIR
Use basic elements to enable filter customization
Input processing:
Columns as channels (frame based)
Use symbolic names for coefficients
OK
Cancel
Help
Apply

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.
- Impulse response and step response in the time domain.
- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify the shape and length of the filter.

Pulse shape

Select the shape of the impulse response from the following options:

- `Raised Cosine`
- `Square Root Raised Cosine`
- `Gaussian`

Order mode

This specification is only available for raised cosine and square root raised cosine filters. For these filters, select one of the following options:

- `Minimum`— This option will result in the minimum-length filter satisfying the user-specified **Frequency specifications**.
- `Specify order`—This option allows the user to construct a raised cosine or square root cosine filter of a specified order by entering an even number in the **Order** input box. The length of the impulse response will be `Order+1` .
- `Specify symbols`—This option enables the user to specify the length of the impulse response in an alternative manner. If `Specify symbols` is chosen, the **Order** input box changes to the **Number of symbols** input box.

Samples per symbol

Specify the oversampling factor. Increasing the oversampling factor guards against aliasing and improves the FIR filter approximation to the ideal frequency response.

If **Order** is specified in **Number of symbols**, the filter length will be **Number of symbols*Samples per symbol**+1. The product **Number of symbols*Samples per symbol** must be an even number.

If a Gaussian filter is specified, the filter length must be specified in **Number of symbols** and **Samples per symbol**. The product **Number of symbols*Samples per symbol** must be an even number. The filter length will be **Number of symbols*Samples per symbol**+1.

Frequency specifications

In this group, you specify the frequency response of the filter. For raised cosine and square root raised cosine filters, the frequency specifications include:

`Rolloff factor`

The rolloff factor takes values in the range [0,1]. The smaller the rolloff factor, the steeper the transition in the stopband.

`Frequency units`

The frequency units are normalized by default. If you specify units other than normalized, the block assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: `Normalized (0 to 1), Hz, kHz, MHz, GHz`

For a Gaussian pulse shape, the available frequency specifications are:

`Bandwidth-time product`

This option allows the user to specify the width of the Gaussian filter. Note that this is independent of the length of the filter. The bandwidth-time product (`BT`) must be a positive real number. Smaller values of the bandwidth-time product result in larger pulse widths in time and steeper stopband transitions in the frequency response.

`Frequency units`

The frequency units are normalized by default. If you specify units other than normalized, the block assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: `Normalized (0 to 1), Hz, kHz, MHz, GHz`

Magnitude specifications

If the **Order mode** is specified as `minimum`, the magnitude units may be selected from:

- `dB` — Specify the magnitude in decibels (default).
- `Linear` — Specify the magnitude in linear units.

Algorithm

The only design method available for FIR pulse-shaping filters is the window method.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. FIR filters use direct-form structure.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for**

filter customization parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note: The `Inherited (this choice will be removed – see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.
- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

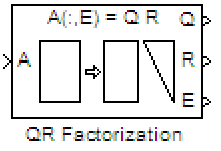
Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Introduced in R2009b

QR Factorization

Factor arbitrary matrix into unitary and upper triangular components



Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

`dspfactors`

Description

The QR Factorization block uses a sequence of Householder transformations to triangularize the input matrix A. The block factors a column permutation of the M-by-N input matrix A as
$A_e = QR$

The column-pivoted matrix A_e contains the columns of A permuted as indicated by the contents of length-N permutation vector E.

```
Ae = A(:,E)     % Equivalent MATLAB code
```

The block selects a column permutation vector E, which ensures that the diagonal elements of matrix R are arranged in order of decreasing magnitude.

$$|r_{i+1,j+1}| < |r_{i,j}| \qquad i = j$$

The size of matrices Q and R depends on the setting of the **Output size** parameter:

- When you select `Economy` for the output size, Q is an M-by-min(M,N) unitary matrix, and R is a min(M,N)-by-N upper-triangular matrix.

```
[Q R E] = qr(A,0)   % Equivalent MATLAB code
```

- When you select `Full` for the output size, Q is an M-by-M unitary matrix, and R is a M-by-N upper-triangular matrix.

```
[Q R E] = qr(A)      % Equivalent MATLAB code
```

The block treats length-M unoriented vector input as an M-by-1 matrix.

QR factorization is an important tool for solving linear systems of equations because of good error propagation properties and the invertability of unitary matrices:
$Q^{-1} = Q'$

where Q' is the complex conjugate transpose of Q.

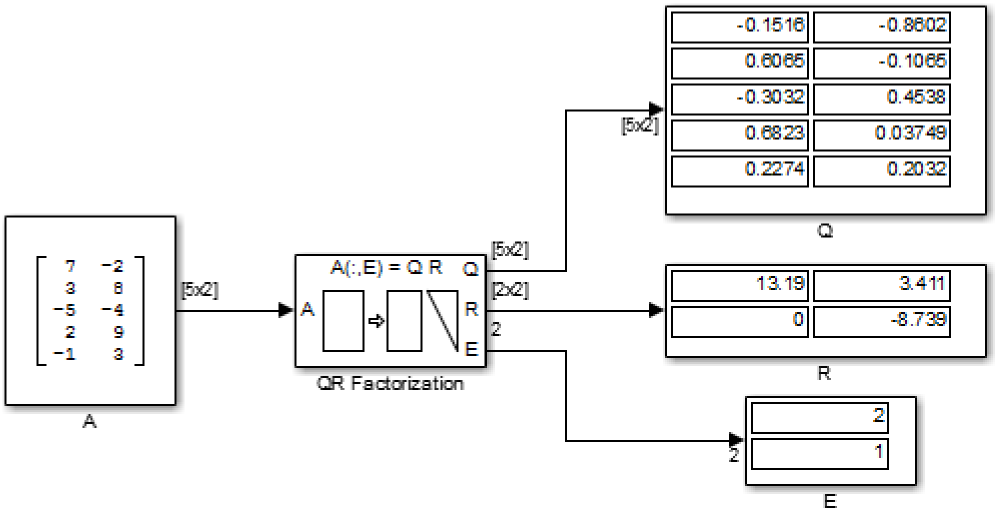
Unlike LU and Cholesky factorizations, the matrix A does not need to be square for QR factorization. However, QR factorization requires twice as many operations as LU Factorization (Gaussian elimination).

Examples

The **Output size** parameter of the QR factorization block has two settings: `Economy` and `Full`. When the M-by-N input matrix A has dimensions such that $M > N$, the dimensions of output matrices Q and R differ depending on the setting of the **Output size** parameter. If, however, the size of the input matrix A is such that $M \leq N$, output matrices Q and R have the same dimensions, regardless of whether the **Output size** is set to `Economy` or `Full`.

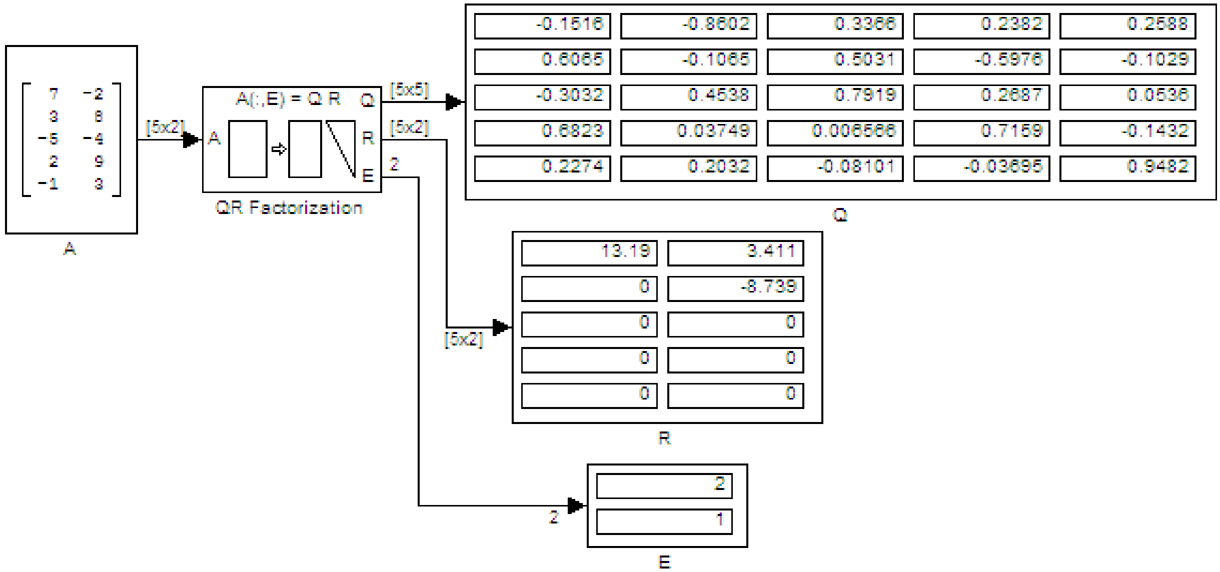
The input to the QR Factorization block in the following model is a `5`-by-`2` matrix A. When you change the setting of the **Output size** parameter from `Economy` to `Full`, the dimensions of the output given by the QR Factorization block also change.

1. Open the model by typing `ex_qrfactorization_ref` at the MATLAB command line.
2. Double-click the QR Factorization block, set the **Output size** parameter to `Economy`, and run the model.



The QR Factorization block outputs a 5-by-2 matrix Q and a 2-by-2 matrix R.

3 Change the **Output size** parameter of the QR Factorization block to `Full` and rerun the model.



The QR Factorization block outputs a 5-by-5 matrix Q and a 5-by-2 matrix R.

Parameters

Output size

Specify the size of output matrices Q and R:

- `Economy` — When this output size is selected, the block outputs an M-by-min(M,N) unitary matrix Q and a min(M,N)-by-N upper-triangular matrix R.
- `Full` — When this output size is selected, the block outputs an M-by-M unitary matrix Q and a M-by-N upper-triangular matrix R.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

See Also

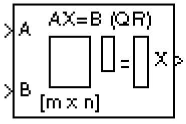
`Cholesky Factorization`	DSP System Toolbox
`LU Factorization`	DSP System Toolbox
`QR Solver`	DSP System Toolbox
`Singular Value Decomposition`	DSP System Toolbox
`qr`	MATLAB

See “Matrix Factorizations” for related information.

Introduced before R2006a

QR Solver

Find minimum-norm-residual solution to A*X*=B



Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

`dspsolvers`

Description

The QR Solver block solves the linear system A*X*=B, which can be overdetermined, underdetermined, or exactly determined. The system is solved by applying QR factorization to the M-by-N matrix, A, at the `A` port. The input to the `B` port is the right side M-by-L matrix, B. The block treats length-M unoriented vector input as an M-by-1 matrix.

The output at the `X` port is the N-by-L matrix, X. X is chosen to minimize the sum of the squares of the elements of B-AX. When B is a vector, this solution minimizes the vector 2-norm of the residual (B-AX is the residual). When B is a matrix, this solution minimizes the matrix Frobenius norm of the residual. In this case, the columns of X are the solutions to the L corresponding systems $AX_k=B_k$, where B_k is the kth column of B, and X_k is the kth column of *X*.

X is known as the minimum-norm-residual solution to AX=B. The minimum-norm-residual solution is unique for overdetermined and exactly determined linear systems, but it is not unique for underdetermined linear systems. Thus when the QR Solver is applied to an underdetermined system, the output *X* is chosen such that the number of nonzero entries in X is minimized.

Algorithm

QR factorization factors a column-permuted variant (A_e) of the M-by-N input matrix A as
$A_e = QR$

where Q is a M-by-min(M,N) unitary matrix, and R is a min(M,N)-by-N upper-triangular matrix.

The factored matrix is substituted for A_e in
$A_e X = B_e$

and
$QRX = B_e$

is solved for X by noting that $Q^{-1} = Q^*$ and substituting $Y = Q^* B_e$. This requires computing a matrix multiplication for Y and solving a triangular system for X.
$RX = Y$

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

`Levinson-Durbin`	DSP System Toolbox
`LDL Solver`	DSP System Toolbox
`LU Solver`	DSP System Toolbox
`QR Factorization`	DSP System Toolbox
`SVD Solver`	DSP System Toolbox

See “Linear System Solvers” for related information.

Introduced before R2006a

Quantizer

Discretize input at specified interval

Library

Quantizers

`dspquant2`

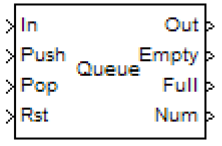
Description

The Quantizer block is an implementation of the Simulink Quantizer block. See Quantizer for more information.

Introduced in R2008a

Queue

Store inputs in FIFO register



Library

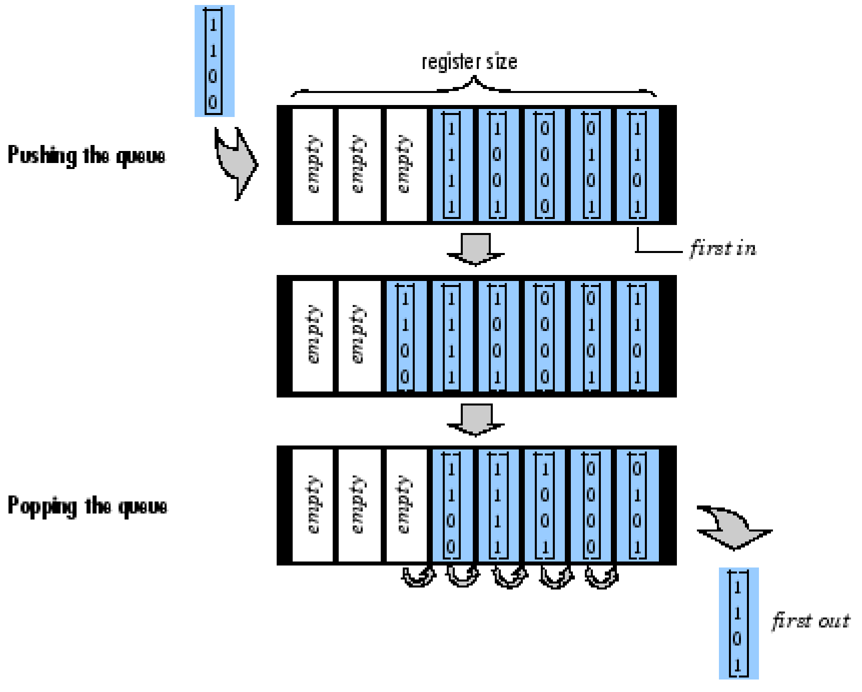
Signal Management / Buffers

`dspbuff3`

Description

The Queue block stores a sequence of input samples in a first in, first out (FIFO) register. The register capacity is set by the **Register size** parameter, and inputs can be scalars, vectors, or matrices.

The block *pushes* the input at the `In` port onto the end of the queue when a trigger event is received at the `Push` port. When a trigger event is received at the `Pop` port, the block *pops* the first element off the queue and holds the `Out` port at that value. The first input to be pushed onto the queue is always the first to be popped off.



A trigger event at the optional `Rst` port empties the queue contents. When you select **Clear output port on reset**, then a trigger event at the `Rst` port empties the queue *and* sets the value at the `Out` port to zero. This setting also applies when a disabled subsystem containing the Queue block is reenabled; the `Out` port value is only reset to zero in this case when you select **Clear output port on reset**.

When you select the **Allow direct feedthrough** check box and two or more of the control input ports are triggered at the same time step, the operations are executed in the following order:

1. `Rst`
2. `Push`
3. `Pop`

When you clear the **Allow direct feedthrough** check box and two or more of the control input ports are triggered at the same time step, the operations are executed in the following order:

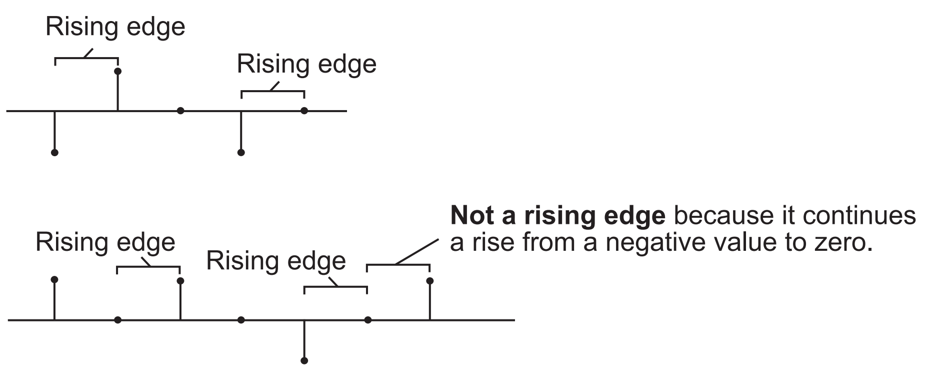
1 `Rst`

2 `Pop`

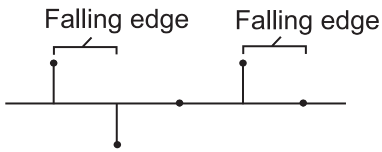
3 `Push`

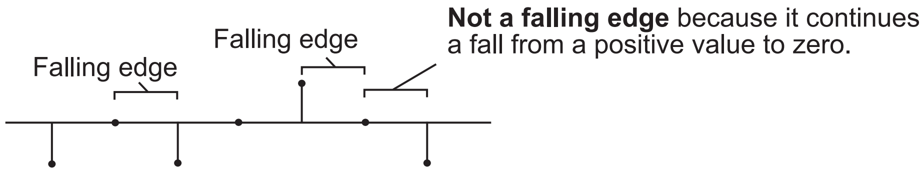
The rate of the trigger signal must be the same as the rate of the data signal input. You specify the triggering event for the `Push`, `Pop`, `and Rst` ports by the **Trigger type** pop-up menu:

- `Rising edge` — Triggers execution of the block when the trigger input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero; see the following figure



- `Falling edge` — Triggers execution of the block when the trigger input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero; see the following figure





- `Either edge` — Triggers execution of the block when the trigger input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers execution of the block at each sample time that the trigger input is not zero.

Note: If your model contains any referenced models that use a Queue block with the **Push onto full register** parameter set to `Dynamic reallocation`, you cannot simulate your top-level model in Simulink Accelerator mode.

The **Push onto full register** parameter specifies the block's behavior when a trigger is received at the `Push` port but the register is full. The **Pop empty register** parameter specifies the block's behavior when a trigger is received at the `Pop` port but the register is empty. The following options are available for both cases:

- `Ignore` — Ignore the trigger event, and continue the simulation.
- `Warning` — Ignore the trigger event, but display a warning message in the MATLAB Command Window.
- `Error` — Display an error dialog box and terminate the simulation.

Note The **Push onto full register** and **Pop empty register** parameters are diagnostic parameters. Like all diagnostic parameters on the Configuration Parameters dialog box, they are set to `Ignore` in the code generated for this block by Simulink Coder code generation software.

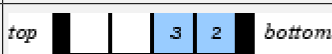
The **Push onto full register** parameter additionally offers the **Dynamic reallocation** option, which dynamically resizes the register to accept as many additional inputs as memory permits. To find out how many elements are on the queue at a given time, enable the `Num` output port by selecting the **Show number of register entries port** parameter.

Note: When `Dynamic reallocation` is selected, the **System target file** parameter on the **Code Generation** pane of the Model Configuration Parameters dialog box must be set to `grt_malloc.tlc — Generic Real-Time Target with dynamic memory allocation`.

Examples

Example 1

The table below illustrates the Queue block's operation for a **Register size** of 4, **Trigger type** of `Either edge`, and **Clear output port on reset** enabled. Because the block triggers on both rising and falling edges in this example, each transition from `1` to `0` or `0` to `1` in the `Push`, `Pop`, and `Rst` columns below represents a distinct trigger event. A `1` in the `Empty` column indicates an empty queue, while a `1` in the `Full` column indicates a full queue.

In	Push	Pop	Rst	Queue	Out	Empty	Full	Num
1	0	0	0		0	1	0	0
2	1	0	0		0	0	0	1
3	0	0	0		0	0	0	2
4	1	0	0		0	0	0	3
5	0	0	0		0	0	1	4
6	0	1	0		2	0	0	3
7	0	0	0		3	0	0	2

In	Push	Pop	Rst	Queue	Out	Empty	Full	Num
8	0	1	0	top 5 bottom	4	0	0	1
9	0	0	0	top bottom	5	1	0	0
10	1	0	0	top 10 bottom	5	0	0	1
11	0	0	0	top 11 10 bottom	5	0	0	2
12	1	0	1	top 12 bottom	0	0	0	1

Note that at the last step shown, the `Push` and `Rst` ports are triggered simultaneously. The `Rst` trigger takes precedence, and the queue is first cleared and then pushed.

Example 2

The dspqdemo example provides another example of the operation of the Queue block.

Parameters

Register size

The number of entries that the FIFO register can hold.

Trigger type

The type of event that triggers the block's execution. The rate of the trigger signal must be the same as the rate of the data signal input.

Push onto full register

Response to a trigger received at the `Push` port when the register is full. Inputs to this port must have the same built-in data type as inputs to the `Pop` and `Rst` input ports.

When `Dynamic reallocation` is selected, the **System target file** parameter on the **Code Generation** pane of the Model Configuration Parameters dialog box must be set to `grt_malloc.tlc — Generic Real-Time Target with dynamic memory allocation`.

Pop empty register

Response to a trigger received at the `Pop` port when the register is empty. Inputs to this port must have the same built-in data type as inputs to the `Push` and `Rst` input ports.

Show empty register indicator port

Enable the `Empty` output port, which is high (`1`) when the queue is empty, and low (`0`) otherwise.

Show full register indicator port

Enable the `Full` output port, which is high (`1`) when the queue is full, and low (`0`) otherwise. The `Full` port remains low when you select `Dynamic reallocation` from the **Push onto full register** parameter.

Show number of register entries port

Enable the `Num` output port, which tracks the number of entries currently on the queue. When inputs to the `In` port are double-precision values, the outputs from the `Num` port are double-precision values. Otherwise, the outputs from the `Num` port are 32-bit unsigned integer values.

Show reset port to clear internal stack buffer

Enable the `Rst` input port, which empties the queue when the trigger specified by the **Trigger type** is received. Inputs to this port must have the same built-in data type as inputs to the `Push` and `Pop` input ports.

Clear output port on reset

Reset the `Out` port to zero, in addition to clearing the queue, when a trigger is received at the `Rst` input port.

Allow direct feedthrough

When you select this check box, the input data is available immediately at the output port of the block. You can turn off direct feedthrough and delay the input data by an extra frame by clearing the **Allow direct feedthrough** check box.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean

Port	Supported Data Types
	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Push	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Inputs to this port must have the same built-in data type as inputs to the Pop and Rst input ports
Pop	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Inputs to this port must have the same built-in data type as inputs to the Push and Rst input ports.
Rst	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Inputs to this port must have the same built-in data type as inputs to the Push and Pop input ports.

Port	Supported Data Types
Out	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Empty	• Double-precision floating point • Boolean
Full	• Double-precision floating point • Boolean
Num	• Double-precision floating point The block outputs a double-precision floating-point value at this port when the data type of the In port is double-precision floating-point. • 32-bit unsigned integers The block outputs a 32-bit unsigned integer value at this port when the data type of the In port is anything other than double-precision floating-point.

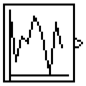
See Also

Buffer	DSP System Toolbox
Delay Line	DSP System Toolbox
Stack	DSP System Toolbox

Introduced before R2006a

Random Source

Generate randomly distributed values



Library

Sources

`dspsrcs4`

Description

The Random Source block generates a frame of M values drawn from a uniform or Gaussian pseudorandom distribution, where you specify M in the **Samples per frame** parameter.

This reference page contains a detailed discussion of the following Random Source block topics:

- "Distribution Type" on page 2-1446
- "Output Complexity" on page 2-1447
- "Output Repeatability" on page 2-1448
- "Specifying the Initial Seed" on page 2-1449
- "Sample Period" on page 2-1450
- "Parameters" on page 2-1450
- "Supported Data Types" on page 2-1452
- "See Also" on page 2-1453

Distribution Type

When the **Source type** parameter is set to `Uniform`, the output samples are drawn from a uniform distribution whose minimum and maximum values are specified by the **Minimum** and **Maximum** parameters, respectively. All values in this range are equally likely to be selected. A length-N vector specified for one or both of these parameters generates an N-channel output (M-by-N matrix) containing a unique random distribution in each channel.

For example, specify

- **Minimum** = `[ 0 0 -3 -3]`
- **Maximum** = `[10 10 20 20]`

to generate a four-channel output whose first and second columns contain random values in the range [0, 10], and whose third and fourth columns contain random values in the range [-3, 20]. When you specify only one of the **Minimum** and **Maximum** parameters as a vector, the block scalar expands the other parameter so it is the same length as the vector.

When the **Source type** parameter is set to `Gaussian`, you must also set the **Method** parameter, which determines the method by which the block computes the output, and has the following settings:

- `Ziggurat` — Produces Gaussian random values by using the Ziggurat method.
- `Sum of uniform values` — Produces Gaussian random values by adding and scaling uniformly distributed random signals based on the central limit theorem. This theorem states that the probability distribution of the sum of a sufficiently high number of random variables approaches the Gaussian distribution. You must set the **Number of uniform values to sum** parameter, which determines the number of uniformly distributed random numbers to sum to produce a single Gaussian random value.

For both settings of the **Method** parameter, the output samples are drawn from the normal distribution defined by the **Mean** and **Variance** parameters. A length-N vector specified for one or both of the **Mean** and **Variance** parameters generates an N-channel output (M-by-N frame matrix) containing a distinct random distribution in each column. When you specify only one of these parameters as a vector, the block scalar expands the other parameter so it is the same length as the vector.

Output Complexity

The block's output can be either real or complex, as determined by the `Real` and `Complex` options in the **Complexity** parameter. These settings control all channels of the output, so real and complex data cannot be combined in the same output. For complex output with a `Uniform` distribution, the real and imaginary components in each channel are both drawn from the same uniform random distribution, defined by the **Minimum** and **Maximum** parameters for that channel.

For complex output with a `Gaussian` distribution, the real and imaginary components in each channel are drawn from normal distributions with different means. In this case, the **Mean** parameter for each channel should specify a complex value; the real component of the **Mean** parameter specifies the mean of the real components in the channel, while the imaginary component specifies the mean of the imaginary components in the channel. When either the real or imaginary component is omitted from the **Mean** parameter, a default value of 0 is used for the mean of that component.

For example, a **Mean** parameter setting of `[5+2i 0.5 3i]` generates a three-channel output with the following means.

Channel 1 mean	*real* = 5	*imaginary* = 2
Channel 2 mean	*real* = 0.5	*imaginary* = 0
Channel 3 mean	*real* = 0	*imaginary* = 3

For complex output, the **Variance** parameter, σ^2, specifies the *total variance* for each output channel. This is the sum of the variances of the real and imaginary components in that channel.

$$\sigma^2 = \sigma_{\mathrm{Re}}^2 + \sigma_{\mathrm{Im}}^2$$

The specified variance is equally divided between the real and imaginary components, so that

$$\sigma_{\mathrm{Re}}^2 = \frac{\sigma^2}{2}$$
$$\sigma_{\mathrm{Im}}^2 = \frac{\sigma^2}{2}$$

Output Repeatability

The **Repeatability** parameter determines whether or not the block outputs the same signal each time you run the simulation. You can set the parameter to one of the following options:

- `Repeatable` — Outputs the same signal each time you run the simulation. The first time you run the simulation, the block randomly selects an initial seed. The block reuses these same initial seeds every time you rerun the simulation.
- `Specify seed` — Outputs the same signal each time you run the simulation. Every time you run the simulation, the block uses the initial seed(s) specified in the **Initial seed** parameter. Also see "Specifying the Initial Seed" on page 2-1449.
- `Not repeatable` — Does not output the same signal each time you run the simulation. Every time you run the simulation, the block randomly selects an initial seed.

Specifying the Initial Seed

When you set the **Repeatability** parameter to `Specify seed`, you must set the **Initial seed** parameter. The **Initial seed** parameter specifies the initial seed for the pseudorandom number generator. The generator produces an identical sequence of pseudorandom numbers each time it is executed with a particular initial seed.

Specifying Initial Seeds for Real Outputs

To specify the N initial seeds for an N-channel real-valued output, **Complexity** parameter set to `Real`, provide one of the following in the **Initial seed** parameter:

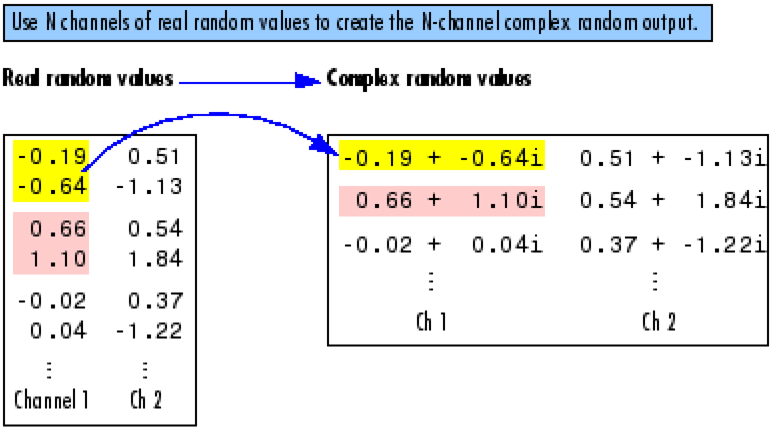
- Length-N vector of initial seeds — Uses each vector element as an initial seed for the corresponding channel in the N-channel output.
- Single scalar — Uses the scalar to generate N random values, which it uses as the seeds for the N-channel output.

Specifying Initial Seeds for Complex Outputs

To specify the initial seeds for an N-channel complex-valued output, **Complexity** parameter set to `Complex`, provide one of the following in the **Initial seed** parameter:

- Length-N vector of initial seeds — Uses each vector element as an initial seed for generating N channels of *real* random values. The block uses pairs of adjacent values in each of these channels as the real and imaginary components of the final output, as illustrated in the following figure.
- Single scalar — Uses the scalar to generate N random values, which it uses as the seeds for generating N channels of *real* random values. The block uses pairs of

adjacent values in each of these channels as the real and imaginary components of the final output, as illustrated in the following figure.



Sample Period

The **Sample time** parameter value, T_s, specifies the random sequence sample period when the **Sample mode** parameter is set to `Discrete`. In this mode, the block generates the number of samples specified by the **Samples per frame** parameter value, M, and outputs this frame with a period of $M*T_s$.

When **Sample mode** is set to `Continuous`, the block is configured for continuous-time operation, and the **Sample time** and **Samples per frame** parameters are disabled. Note that many DSP System Toolbox blocks do not accept continuous-time inputs.

Parameters

Source type

The distribution from which to draw the random values, `Uniform` or `Gaussian`. For more information, see "Distribution Type" on page 2-1446.

Method

The method by which the block computes the Gaussian random values, `Ziggurat` or `Sum of uniform values`. This parameter is enabled when **Source type** is set to `Gaussian`. For more information, see "Distribution Type" on page 2-1446.

Minimum

The minimum value in the uniform distribution. This parameter is enabled when you select `Uniform` from the **Source type** parameter. Tunable (Simulink) in Simulation mode only.

Maximum

The maximum value in the uniform distribution. This parameter is enabled when you select you select `Uniform` from the **Source type** parameter. Tunable (Simulink) in Simulation mode only.

Number of uniform values to sum

The number of uniformly distributed random values to sum to compute a single number in a Gaussian random distribution. This parameter is enabled when the **Source type** parameter is set to `Gaussian`, and the **Method** parameter is set to `Sum of uniform values`. For more information, see "Distribution Type" on page 2-1446.

Mean

The mean of the Gaussian (normal) distribution. This parameter is enabled when you select `Gaussian` from the **Source type** parameter. Tunable (Simulink) in Simulation mode only.

Variance

The variance of the Gaussian (normal) distribution. This parameter is enabled when you select `Gaussian` from the **Source type** parameter. Tunable (Simulink) in Simulation mode only.

Repeatability

The repeatability of the block output: `Not repeatable`, `Repeatable`, or `Specify seed`. In the `Repeatable` and `Specify seed` settings, the block outputs the same signal every time you run the simulation. For details, see "Output Repeatability" on page 2-1448.

Initial seed

The initial seed(s) to use for the random number generator when you set the **Repeatability** parameter to `Specify seed`. For details, see "Specifying the Initial Seed" on page 2-1449. Tunable (Simulink) in Simulation mode only.

Inherit output port attributes

When you select this check box, block inherits the sample mode, sample time, output data type, complexity, and signal dimensions of the signal from the downstream block. When you select this check box, the **Sample mode**, **Sample time**, **Samples per frame**, **Output data type**, and **Complexity** parameters are disabled.

Suppose you want to back propagate a 1-D vector. The output of the Random Source block is a 1-D vector of length M, where length M is inherited from the downstream block. When the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter specifies N channels, the 1-D vector output contains M/N samples from each channel. An error occurs in this case when M is not an integer multiple of N.

Suppose you want to back propagate a M-by-N signal. When N>1, your signal has N channels. When N = 1, your signal has M channels. The value of the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter can be a scalar or a vector of length equal to the number of channels. You can specify these parameters as either row or column vectors, except when the signal is a row vector. In this case, the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter must also be specified as a row vector.

Sample mode

The sample mode, `Continuous` or `Discrete`. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Sample time

The sample period, T_s, of the random output sequence. The output frame period is $M*T_s$. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Samples per frame

The number of samples, M, in each output frame.

This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Output data type

The data type of the output, single-precision or double-precision. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Complexity

The complexity of the output, `Real` or `Complex`. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Supported Data Types

- Double-precision floating-point
- Single-precision floating-point

See Also

`Discrete Impulse`	DSP System Toolbox
`Maximum`	DSP System Toolbox
`Minimum`	DSP System Toolbox
`Signal From Workspace`	DSP System Toolbox
`Standard Deviation`	DSP System Toolbox
`Variance`	DSP System Toolbox
Constant	Simulink
Random Number	Simulink
Signal Generator	Simulink
`rand`	MATLAB
`randn`	MATLAB
`RandStream`	MATLAB

Introduced before R2006a

Real Cepstrum

Compute real cepstrum of input

Library

Transforms

`dspxfrm3`

Description

The Real Cepstrum block computes the real cepstrum of each column in the real-valued M-by-N input matrix, `u`. The block treats each column of the input as an independent channel containing M consecutive samples. The block *always* processes unoriented vector inputs as a single channel, and returns the result as a length-M column vector. The block does not accept complex-valued inputs.

The output is a real M_o-by-N matrix, where you specify M_o in the **FFT length** parameter. Each output column contains the length-M_o real cepstrum of the corresponding input column.

```
y = real(ifft(log(abs(fft(u,Mo)))))
```

or, more compactly,

```
y = rceps(u,Mo)
```

When you select the **Inherit FFT length from input port dimensions** check box, the output frame size matches the input frame size (M_o=M).

The output port rate is the same as the input port rate.

Parameters

Inherit FFT length from input port dimensions

When you select this check box, the output frame size matches the input frame size.

FFT length

The number of frequency points at which to compute the FFT, which is also the output frame size, M_o. This parameter is visible only when you clear the **Inherit FFT length from input port dimensions** check box.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

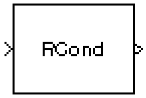
See Also

`Complex Cepstrum`	DSP System Toolbox
`DCT`	DSP System Toolbox
`FFT`	DSP System Toolbox
`rceps`	Signal Processing Toolbox

Introduced before R2006a

Reciprocal Condition

Compute reciprocal condition of square matrix in 1-norm



Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

Description

The Reciprocal Condition block computes the reciprocal of the condition number for a square input matrix A.

```
y = rcond(A)     % Equivalent MATLAB code
```

or

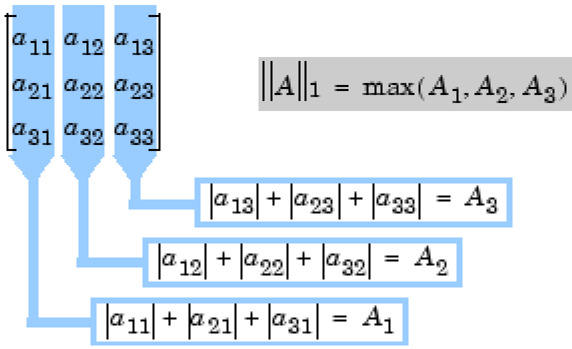
$$y = \frac{1}{\kappa} = \frac{1}{\|A^{-1}\|_1 \|A\|_1}$$

where κ is the condition number ($\kappa \geq 1$), and y is the scalar output ($0 \leq y < 1$).

The matrix 1-norm, $\|A\|_1$, is the maximum column-sum in the M-by-M matrix A.

$$\|A\|_1 = \max_{1 \leq j \leq M} \sum_{i=1}^{M} |a_{ij}|$$

For a 3-by-3 matrix:



References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

Matrix 1-Norm	DSP System Toolbox
`Normalization`	DSP System Toolbox
`rcond`	MATLAB

Introduced before R2006a

Repeat

Resample input at higher rate by repeating values

Library

Signal Operations

`dspsigops`

Description

The Repeat block upsamples each channel of the M_i-by-N input to a rate L times higher than the input sample rate. To do so, the block repeats each consecutive input sample L times at the output. You specify the integer L in the **Repetition count** parameter.

You can use the Repeat block inside of triggered subsystems when you set the **Rate options** parameter to `Enforce single-rate processing`.

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block upsamples each column of the input over time. In this mode, the block can perform either single-rate or multirate processing. You can use the **Rate options** parameter to specify how the block upsamples the input:

- When you set the **Rate options** parameter to `Enforce single-rate processing`, the input and output of the block have the same sample rate. In this mode, the block outputs a signal with a proportionally larger frame *size* than the input. The block upsamples each channel independently by repeating each row of the input matrix L times at the output. For upsampling by a factor of L, the output frame size is L times larger than the input frame size ($M_o = M_i*L$), but the input and output frame rates are equal.

For an example of single-rate upsampling, see Example: Single-Rate Processing.

- When you set the **Rate options** parameter to `Allow multirate processing`, the block treats an M_i-by-N matrix input as N independent channels. The block generates the output at the faster (upsampled) rate by using a proportionally shorter frame *period* at the output port than at the input port. For L repetitions of the input, the output frame period is L times shorter than the input frame period ($T_{fo} = T_{fi}/L$). In this mode, the output always has the same frame size as the input.

  See Example: Multirate, Frame-Based Processing for an example that uses the Repeat block in this mode.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an M-by-N matrix input as $M*N$ independent channels, and upsamples each channel over time. The block upsamples each channel over time such that the output sample rate is L times higher than the input sample rate ($T_{so} = T_{si}/L$). In this mode, the output is always the same size as the input.

Zero Latency

The Repeat block has *zero-tasking latency* for all single-rate operations. The block is in a single-rate mode if you set the **Repetition count** parameter to `1` or if you set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`.

The Repeat block also has zero-tasking latency for multirate operations if you run your model in Simulink single-tasking mode.

Zero-tasking latency means that the block repeats the first input (received at t=**0**) for the first L output samples, the second input for the next L output samples, and so on.

Nonzero Latency

The Repeat block has tasking latency for multirate, multitasking operation:

- In multirate, sample-based processing mode, the initial condition for each channel is repeated for the first L output samples. The channel's first input appears as output sample L+1. The **Initial conditions** parameter can be an M_i-by-N matrix containing one value for each channel, or a scalar to be applied to all signal channels.

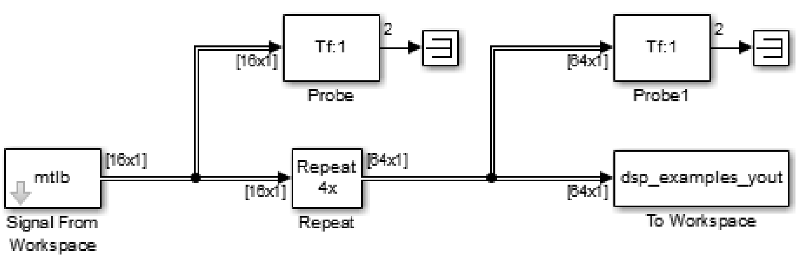
- In multirate, frame-based processing mode, the first row of the initial condition matrix is repeated for the first L output samples, the second row of the initial condition matrix is repeated for the next L output samples, and so on. The first row of the first input matrix appears in the output as sample M_iL+1. The **Initial conditions** parameter can be an M_i-by-N matrix, or a scalar to be repeated across all elements of the M_i-by-N matrix.

Note: For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Time-Based Scheduling and Code Generation" (Simulink Coder).

Examples

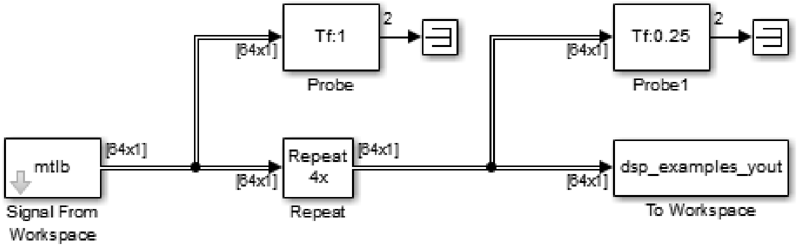
Example: Single-Rate Processing

In the ex_repeat_ref2 model, the Repeat block resamples a single-channel input with a frame size of 16. The block repeats input values to upsample the input by a factor of 4. Thus, the output of the block has a frame size of 64. The input and output frame rates are identical.



Example: Multirate, Frame-Based Processing

In the ex_repeat_ref1 model, the Repeat block resamples a single-channel input with a frame period of 1 second. The block repeats input values to upsample the input by a factor of 4. Thus, the output of the block has a frame period of 0.25 seconds. The input and output frame sizes are identical.



Parameters

Repetition count

The integer number of times, L, that the input value is repeated at the output. This is the factor by which the block increases the output frame size or sample rate.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel. In this mode, the block can perform single-rate or multirate processing.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel. In this mode, the block always performs multirate processing.

Rate options

Specify the method by which the block upsamples the input. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate by increasing the output frame size by a factor of L. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block resamples the signal such that the output sample rate is L times faster than the input sample rate.

Initial conditions

The value with which the block is initialized for cases of nonzero latency; a scalar or matrix. This parameter appears only when you configure the block to perform multirate processing.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Repeat.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

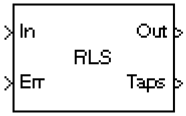
See Also

`FIR Interpolation`	DSP System Toolbox
`Upsample`	DSP System Toolbox

Introduced before R2006a

RLS Adaptive Filter (Obsolete)

Compute filter estimates for input using RLS adaptive filter algorithm



Library

`dspobslib`

Description

Note The RLS Adaptive Filter block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the `RLS Filter` block.

The RLS Adaptive Filter block recursively computes the recursive least squares (RLS) estimate of the FIR filter coefficients.

The corresponding RLS filter is expressed in matrix form as

$$
\begin{aligned}
k(n) &= \frac{\lambda^{-1}P(n-1)u(n)}{1+\lambda^{-1}u^{H}(n)P(n-1)u(n)} \\
y(n) &= \hat{w}^{H}(n-1)u(n) \\
e(n) &= d(n)-y(n) \\
\hat{w}(n) &= \hat{w}(n-1)+k(n)e^{*}(n) \\
P(n) &= \lambda^{-1}P(n-1)-\lambda^{-1}k(n)u^{H}(n)P(n-1)
\end{aligned}
$$

where λ^{-1} denotes the reciprocal of the exponential weighting factor. The variables are as follows

Variable	Description
n	The current algorithm iteration
$u(n)$	The buffered input samples at step n
$P(n)$	The inverse correlation matrix at step n
$k(n)$	The gain vector at step n
$\hat{w}(n)$	The vector of filter-tap estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
λ	The exponential memory weighting factor

The block icon has port labels corresponding to the inputs and outputs of the RLS algorithm. Note that inputs to the `In` and `Err` ports must be sample-based scalars. The signal at the `Out` port is a scalar, while the signal at the `Taps` port is a sample-based vector.

Block Ports	Corresponding Variables
`In`	u, the scalar input, which is internally buffered into the vector $u(n)$
`Out`	$y(n)$, the filtered scalar output
`Err`	$e(n)$, the scalar estimation error
`Taps`	$\hat{w}(0)$, the vector of filter-tap estimates

An optional `Adapt` input port is added when you select the **Adapt input** check box in the dialog box. When this port is enabled, the block continuously adapts the filter coefficients while the `Adapt` input is nonzero. A zero-valued input to the `Adapt` port causes the block to stop adapting, and to hold the filter coefficients at their current values until the next nonzero `Adapt` input.

The implementation of the algorithm in the block is optimized by exploiting the symmetry of the inverse correlation matrix $P(n)$. This decreases the total number of computations by a factor of two.

The **FIR filter length** parameter specifies the length of the filter that the RLS algorithm estimates. The **Memory weighting factor** corresponds to λ in the equations, and specifies how quickly the filter "forgets" past sample information. Setting λ=1 specifies an infinite memory; typically, $0.95 \le \lambda \le 1$.

The **Initial value of filter taps** specifies the initial value $\hat{w}(0)$ as a vector, or as a scalar to be repeated for all vector elements. The initial value of $P(n)$ is

$$I\frac{1}{\hat{\sigma}^2}$$

where you specify $\hat{\sigma}^2$ in the **Initial input variance estimate** parameter.

Examples

The rlsdemo example illustrates a noise cancellation system built around the RLS Adaptive Filter block.

Parameters

FIR filter length

The length of the FIR filter.

Memory weighting factor

The exponential weighting factor, in the range `[0,1]`. A value of `1` specifies an infinite memory. Tunable (Simulink).

Initial value of filter taps

The initial FIR filter coefficients.

Initial input variance estimate

The initial value of 1/P(n).

Adapt input

Enables the `Adapt` port.

References

Haykin, S. *Adaptive Filter Theory*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1996.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

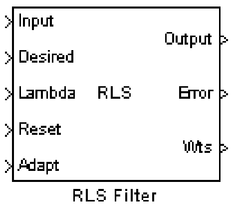
Kalman Adaptive Filter (Obsolete)	DSP System Toolbox
LMS Adaptive Filter (Obsolete)	DSP System Toolbox

See "Adaptive Filters in Simulink" for related information.

Introduced in R2008b

RLS Filter

Compute filtered output, filter error, and filter weights for given input and desired signal using RLS adaptive filter algorithm



Library

Filtering / Adaptive Filters

`dspadpt3`

Description

The RLS Filter block recursively computes the least squares estimate (RLS) of the FIR filter weights. The block estimates the filter weights, or coefficients, needed to convert the input signal into the desired signal. Connect the signal you want to filter to the Input port. The input signal can be a scalar or a column vector. Connect the signal you want to model to the Desired port. The desired signal must have the same data type, complexity, and dimensions as the input signal. The Output port outputs the filtered input signal. The Error port outputs the result of subtracting the output signal from the desired signal.

The corresponding RLS filter is expressed in matrix form as

$$\mathbf{k}(n) = \frac{\lambda^{-1}\mathbf{P}(n-1)\mathbf{u}(n)}{1+\lambda^{-1}\mathbf{u}^{H}(n)\mathbf{P}(n-1)\mathbf{u}(n)}$$
$$y(n) = \mathbf{w}(n-1)\mathbf{u}(n)$$
$$e(n) = d(n) - y(n)$$
$$\mathbf{w}(n) = \mathbf{w}(n-1) + \mathbf{k}^{H}(n)e(n)$$
$$\mathbf{P}(n) = \lambda^{-1}\mathbf{P}(n-1) - \lambda^{-1}\mathbf{k}(n)\mathbf{u}^{H}(n)\mathbf{P}(n-1)$$

where λ^{-1} denotes the reciprocal of the exponential weighting factor. The variables are as follows

Variable	Description
n	The current time index
$\mathbf{u}(n)$	The vector of buffered input samples at step n
$\mathbf{P}(n)$	The inverse covariance matrix at step n
$\mathbf{k}(n)$	The gain vector at step n
$\mathbf{w}(n)$	The vector of filter-tap estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
λ	The forgetting factor

The implementation of the algorithm in the block is optimized by exploiting the symmetry of the inverse covariance matrix $P(n)$. This decreases the total number of computations by a factor of two.

Use the **Filter length** parameter to specify the length of the filter weights vector.

The **Forgetting factor (0 to 1)** parameter corresponds to λ in the equations. It specifies how quickly the filter "forgets" past sample information. Setting λ=1 specifies an infinite memory. Typically, $1-\frac{1}{2L}<\lambda<1$, where L is the filter length. You can specify a forgetting factor using the input port, Lambda, or enter a value in the **Forgetting factor (0 to 1)** parameter in the Block Parameters: RLS Filter dialog box.

Enter the initial filter weights, $\widehat{w}(0)$, as a vector or a scalar for the **Initial value of filter weights** parameter. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value.

The initial value of $P(n)$ is

$$\frac{1}{\sigma^2} I$$

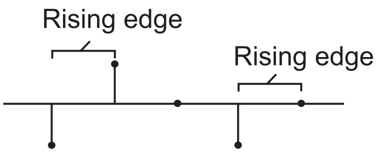
where you specify σ^2 in the **Initial input variance estimate** parameter.

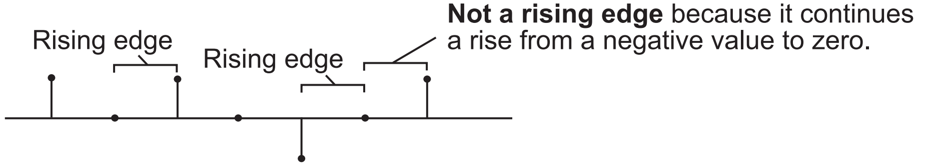
When you select the **Adapt port** check box, an Adapt port appears on the block. When the input to this port is nonzero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain at their current values.

When you want to reset the value of the filter weights to their initial values, use the **Reset input** parameter. The block resets the filter weights whenever a reset event is detected at the Reset port. The reset signal rate must be the same rate as the data signal input.

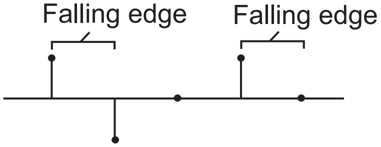
From the **Reset input** list, select `None` to disable the Reset port. To enable the Reset port, select one of the following from the **Reset input** list:

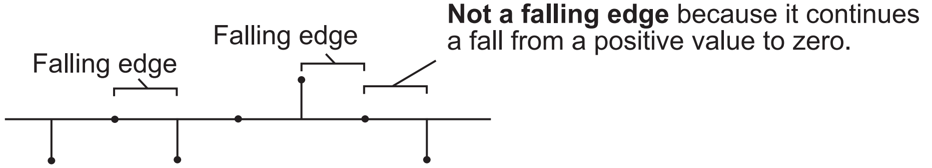
- `Rising edge` — Triggers a reset operation when the Reset input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero; see the following figure





- `Falling edge` — Triggers a reset operation when the Reset input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero; see the following figure





- `Either edge` — Triggers a reset operation when the Reset input is a `Rising edge` or `Falling edge`, as described above
- `Non-zero sample` — Triggers a reset operation at each sample time that the Reset input is not zero

Select the **Output filter weights** check box to create a Wts port on the block. For each iteration, the block outputs the current updated filter weights from this port.

Examples

The rlsdemo example illustrates a noise cancellation system built around the RLS Filter block.

Parameters

Filter length

Enter the length of the FIR filter weights vector.

Specify forgetting factor via

Select `Dialog` to enter a value for the forgetting factor in the Block parameters: RLS Filter dialog box. Select `Input port` to specify the forgetting factor using the Lambda input port.

Forgetting factor (0 to 1)

Enter the exponential weighting factor in the range $0 \leq \lambda \leq 1$. A value of `1` specifies an infinite memory. Tunable (Simulink).

Initial value of filter weights

Specify the initial values of the FIR filter weights.

Initial input variance estimate

The initial value of $1/P(n)$.

Adapt port

Select this check box to enable the Adapt input port.

Reset input

Select this check box to enable the Reset input port.

Output filter weights

Select this check box to export the filter weights from the Wts port.

References

Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

See Also

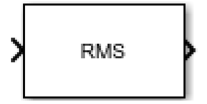
Kalman Adaptive Filter (Obsolete)	DSP System Toolbox
LMS Filter	DSP System Toolbox
Block LMS Filter	DSP System Toolbox
Fast Block LMS Filter	DSP System Toolbox

See “Adaptive Filters in Simulink” for related information.

Introduced before R2006a

RMS

Root mean square value of input or sequence of inputs
Library: DSP System Toolbox / Statistics



Description

The RMS block computes the root mean square (RMS) value of each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the RMS value of the entire input. You can specify the dimension using the **Find the RMS value over** parameter. The RMS block can also track the RMS value in a sequence of inputs over a period of time. To track the RMS value in a sequence of inputs, select the **Running RMS** parameter.

Note: The **Running** mode in the RMS block will be removed in a future release. To compute the running RMS in Simulink, use the `Moving RMS` block instead.

Ports

Input

`In` — Data input
vector | matrix | *N*-D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs.

This port is unnamed until you select the **Running RMS** parameter and set the **Reset port** parameter to any option other than `None`.

Data Types: `single` | `double`
Complex Number Support: Yes

`Rst` — Reset port
scalar

Specify the reset event that causes the block to reset the running RMS. The sample time of the **Rst** input must be a positive integer multiple of the input sample time.

Dependencies

To enable this port, select the **Running RMS** parameter and set the **Reset port** parameter to any option other than `None`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Port_1` — RMS value along the specified dimension

scalar | vector | matrix | *N*-D array

The data type of the output matches the data type of the input.

When you do not select the **Running RMS** parameter, the block computes the RMS value in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the RMS value of the entire input at each individual sample time. Each element in the output array `y` is the RMS value of the corresponding column, row, or entire input. The output array `y` depends on the setting of the **Find the RMS value over** parameter. Consider a three-dimensional input signal of size *M*-by-*N*-by-*P*. When you set **Find the RMS value over** to:

- `Entire input` — The output at each sample time is a scalar that contains the RMS value of the *M*-by-*N*-by-*P* input matrix.
- `Each row` — The output at each sample time consists of an *M*-by-1-by-*P* array, where each element contains the RMS value of each vector over the second dimension of the input. For an *M*-by-*N* matrix input, the output at each sample time is an *M*-by-1 column vector.
- `Each column` — The output at each sample time consists of a 1-by-*N*-by-*P* array, where each element contains the RMS value of each vector over the first dimension of the input. For an *M*-by-*N* matrix input, the output at each sample time is a 1-by-*N* row vector.

  In this mode, the block treats length-*M* unoriented vector inputs as *M*-by-1 column vectors.

- `Specified dimension` — The output at each sample time depends on the value of the **Dimension** parameter. If you set the **Dimension** to `1`, the output is the same as when you select `Each column`. If you set the **Dimension** to `2`, the output is the same as when you select `Each row`. If you set the **Dimension** to `3`, the output at each sample time is an M-by-N matrix containing the RMS value of each vector over the third dimension of the input.

When you select **Running RMS**, the block tracks the RMS value of each channel in a time sequence of inputs. In this mode, you must also specify a value for the **Input processing** parameter.

- `Elements as channels (sample based)` — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size M-by-N-by-P, the block outputs an M-by-N-by-P array. Each element y_{ijk} of the output contains the RMS value of the element u_{ijk} for all inputs since the last reset.

  When a reset event occurs, the running RMS y_{ijk} in the current frame is reset to the element u_{ijk}.
- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel. This option does not support input signals with more than two dimensions. For a two-dimensional input signal of size M-by-N, the block outputs an M-by-N matrix. Each element y_{ij} of the output contains the RMS value of the elements in the jth column of all inputs since the last reset, up to and including the element u_{ij} of the current input.

  When a reset event occurs, the running RMS for each channel becomes the RMS value of all the samples in the current input frame, up to and including the current input sample.

Data Types: `single` | `double`

Parameters

Main Tab

`Running RMS` — Option to select running RMS

off (default) | on

When you select the **Running RMS** parameter, the block tracks the RMS value of each channel in a time sequence of inputs.

`Find the RMS value over` — Dimension over which the block computes the RMS value
`Each column` (default) | `Entire input` | `Each row` | `Specified dimension`

- `Each column` — The block outputs the RMS value over each column.
- `Each row` — The block outputs the RMS value over each row.
- `Entire input` — The block outputs the RMS value over the entire input.
- `Specified dimension` — The block outputs the RMS value over the dimension specified in the **Dimension** parameter.

Dependencies

To enable this parameter, clear the **Running RMS** parameter.

`Dimension` — Custom dimension
`1` (default) | scalar

Specify the dimension (one-based value) of the input signal over which the RMS value is computed. The value of this parameter must be greater than 0 and less than the number of dimensions in the input signal.

Dependencies

To enable this parameter, set **Find the RMS value over** to `Specified dimension`.

`Input processing` — Method to process the input in running mode
`Columns as channels (frame based)` (default) | `Elements as channels (sample based)`

- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel. This option does not support input signals with more than two dimensions. For a two-dimensional input signal of size M-by-N, the block outputs an M-by-N matrix. Each element y_{ij} of the output contains the RMS value of the elements in the jth column of all inputs since the last reset, up to and including the element u_{ij} of the current input.

When a reset event occurs, the running RMS for each channel becomes the RMS value of all the samples in the current input frame, up to and including the current input sample.

- `Elements as channels (sample based)` — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size *M*-by-*N*-by-*P*, the block outputs an *M*-by-*N*-by-*P* array. Each element y_{ijk} of the output contains the RMS value of the element u_{ijk} for all inputs since the last reset.

  When a reset event occurs, the running RMS y_{ijk} in the current frame is reset to the element u_{ijk}.

  Variable-Size Inputs

  When your inputs are of variable size, and you select the **Running RMS** parameter, then:

  - If you set the **Input processing** parameter to `Elements as channels (sample based)`, the state is reset.
  - If you set the **Input processing** parameter to `Columns as channels (frame based)`, then:
    - When the input size difference is in the number of channels (number of columns), the state is reset.
    - When the input size difference is in the length of channels (number of rows), there is no reset and the running operation is carried out as usual.

Dependencies

To enable this parameter, select the **Running RMS** parameter.

`Reset port` — Reset event

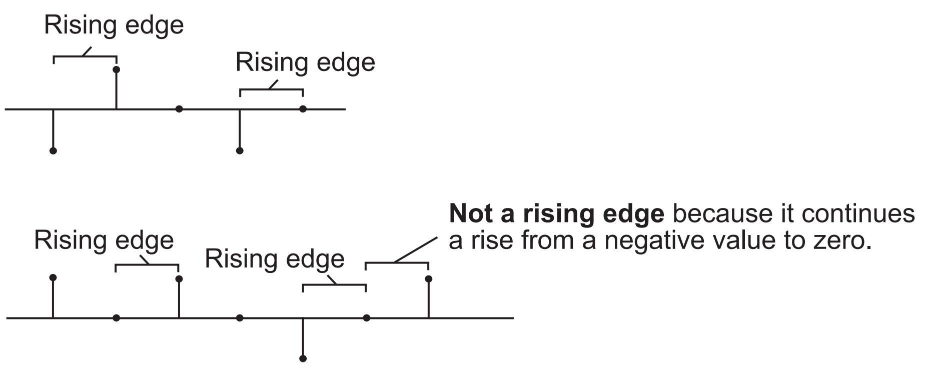
`None` (default) | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero sample`

The block resets the running RMS whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

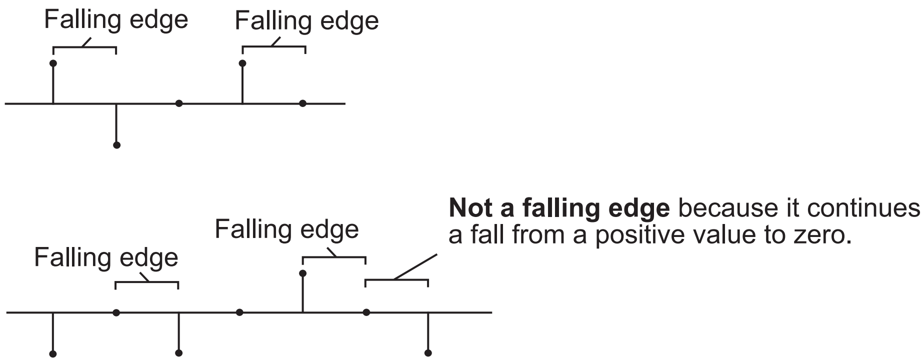
When a reset event occurs while the **Input processing** parameter is set to `Elements as channels (sample based)`, the running RMS for each channel is initialized to the value in the corresponding channel of the current input. Similarly, when the **Input processing** parameter is set to `Columns as channels (frame based)`, the running RMS for each channel becomes the RMS value of all the samples in the current input frame, up to and including the current input sample.

Use this parameter to specify the reset event.

- `None` — Disables the **Rst** port.
- `Rising edge` — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to either a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.



- `Falling edge` — Triggers a reset operation when the **Rst** input does one of the following:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.



- `Either edge` — Triggers a reset operation when the **Rst** input is a `Rising edge` or `Falling edge`.
- `Non-zero sample` — Triggers a reset operation at each sample time, when the **Rst** input is not zero.

Note: When running simulations in the Simulink multitasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Time-Based Scheduling and Code Generation" (Simulink Coder).

Dependencies

To enable this parameter, select the **Running RMS** parameter.

Definitions

Root Mean Square (RMS)

The RMS value of a discrete-time signal is the square root of the arithmetic mean of the squares of the signal sample values.

For an M-by-N input matrix u, the RMS value of the jth column of the input is given by:

$$y_j = \sqrt{\frac{\sum_{i=1}^{M} |u_{ij}|^2}{M}} \qquad 1 \le j \le N$$

Algorithms

Root Mean Square (RMS)

When you clear the **Running RMS** parameter in the block and specify a dimension, the block produces results identical to the MATLAB `rms` function, when it is called as `y = rms(u,D)`.

- `u` is the data input.
- `D` is the dimension.
- `y` is the RMS value.

The RMS value along the entire input is identical to calling the `rms` function as `y = rms(u(:))`.

When inputs are complex, the block computes the RMS value of the magnitude of the complex input.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Functions

`rms`

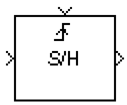
System Objects
dsp.RMS | dsp.MovingRMS

Blocks
`Mean` | `Moving RMS`

Introduced before R2006a

Sample and Hold

Sample and hold input signal



Library

Signal Operations

`dspsigops`

Description

The Sample and Hold block acquires the input at the signal port whenever it receives a trigger event at the trigger port (marked by). The block then holds the output at the acquired input value until the next triggering event occurs.

The trigger input must be a sample-based scalar with sample rate equal to the input frame rate at the signal port. You specify the trigger event using the **Trigger type** parameter:

- `Rising edge` triggers the block to acquire the signal input when the trigger input rises from a negative value or zero to a positive value.
- `Falling edge` triggers the block to acquire the signal input when the trigger input falls from a positive value or zero to a negative value.
- `Either edge` triggers the block to acquire the signal input when the trigger input either rises from a negative value or zero to a positive value or falls from a positive value or zero to a negative value.

You specify the block's output prior to the first trigger event using the **Initial condition** parameter. When the acquired input is an M-by-N matrix, the **Initial condition** can be an M-by-N matrix, or a scalar to be repeated across all elements of the matrix. When the input is a length-M unoriented vector, the **Initial condition** can be a length-M row or column vector, or a scalar to be repeated across all elements of the vector.

If you select the **Latch (buffer) input** check box, the block outputs the value of the input from the previous time step until the next triggering event occurs. To use this block in a loop, select this check box.

Parameters

Trigger type

The type of event that triggers the block to acquire the input signal.

Initial condition

The block's output prior to the first trigger event.

Latch (buffer) input

If you select this check box, the block outputs the value of the input from the previous time step until the next triggering event occurs.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Trigger	• Any data type supported by the Trigger block
Outputs	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

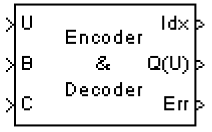
See Also

Downsample	DSP System Toolbox
N-Sample Switch	DSP System Toolbox

Introduced before R2006a

Scalar Quantizer (Obsolete)

Convert input signal into set of quantized output values or index values, or convert set of index values into quantized output signal



Library

dspobslib

Description

Note The Scalar Quantizer block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the `Scalar Quantizer Encoder` block or the `Scalar Quantizer Decoder` block.

The Scalar Quantizer block has three modes of operation. In `Encoder` mode, the block maps each input value to a quantization region by comparing the input value to the quantizer boundary points defined in the **Boundary points** parameter. The block outputs the index of the associated region. In `Decoder` mode, the block transforms the input index values into quantized output values, defined in the **Codebook** parameter. In the `Encoder and Decoder` mode, the block performs both the encoding and decoding operations. The block outputs the index values and the quantized output values.

You can select how you want to enter the **Boundary points** and/or **Codebook** values using the **Source of quantizer** parameters. When you select `Specify via dialog`, type the parameters into the block parameters dialog box. Select `Input ports`, and port B and/or C appears on the block. In `Encoder` and `Encoder and decoder` mode, the input to port B is used as the **Boundary points**. In `Decoder` and `Encoder and decoder` mode, the input to port C is used as the **Codebook**.

In `Encoder` and `Encoder and decoder` mode, the **Boundary points** are the values used to break up the input signal into regions. Each region is specified by an index number. When your first boundary point is `-inf` and your last boundary point is `inf`, your quantizer is unbounded. When your first and last boundary point is finite, your quantizer is bounded. When only your first or last boundary point is `-inf` or `inf`, your quantizer is semi-bounded.

For instance, when your input signal ranges from 0 to 11, you can create a bounded quantizer using the following boundary points:

```
[0 0.5 3.7 5.8 6.0 11]
```

The boundary points can have equal or varied spacing. Any input values between 0 and 0.5 would correspond to index 0. Input values between 0.5 and 3.7 would correspond to index 1, and so on.

Suppose you wanted to create an unbounded quantizer with the following boundary points:

```
[-inf 0 2 5.5 7.1 10 inf]
```

When your input signal has values less than 0, these values would be assigned to index 0. When your input signal has values greater than 10, these values would be assigned to index 6.

When an input value is the same as a boundary point, the **Tie-breaking rule** parameter defines the index to which the value is assigned. When you want the input value to be assigned to the lower index value, select `Choose the lower index`. To assign the input value with the higher index, select `Choose the higher index`.

In `Decoder` and `Encoder and decoder` mode, the **Codebook** is a vector of quantized output values that correspond to each index value.

In `Encoder` and `Encoder and decoder` mode, the **Searching method** determines how the appropriate quantizer index is found. Select `Linear` and the Scalar Quantizer block compares the input value to the first region defined by the first two boundary points. When the input value does not fall within this region, the block then compares the input value to the next region. This process continues until the input value is determined to be within a region and is associated with the appropriate index value. The computational cost of this process is of the order P, where P is the number of boundary points.

Select Binary for the **Searching method** and the block compares the input value to the middle value of the boundary points vector. When the input value is larger than this boundary point, the block discards the boundary points that are lower than this middle value. The block then compares the input value to the middle boundary point of the new range, defined by the remaining boundary points. This process continues until the input value is associated with the appropriate index value. The computational cost of this process is of the order $\log_2 P$, where P is the number of boundary points. In most cases, the Binary option is faster than the Linear option.

In Decoder mode, the input to this block is a vector of index values, where $0 \leq$ index $< N$ and N is the length of the codebook vector. Use the **Action for out of range input** parameter to determine what happens when an input index value is out of this range. When you want any index values less than 0 to be set to 0 and any index values greater than or equal to N to be set to N -1, select Clip. When you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to N are set to N -1, select Clip and warn. When you want the simulation to stop and display an error when the index values are out of range, select Error.

In Encoder and decoder mode, you can select the **Output the quantization error** check box. The quantization error is the difference between the input value and the quantized output value. Select this check box to output the quantization error for each input value from the Err port on this block.

Data Type Support

In Encoder mode, the input data values and the boundary points can be the input to the block at ports U and B. Similarly, in Encoder and decoder mode, the codebook values can also be the input to the block at port C. The data type of the input data values, boundary points, and codebook values can be double, single, uint8, uint16, uint32, int8, int16, or int32. In Decoder mode, the input to the block can be the index values and the codebook values. The data type of the index input to the block at port Idx can be uint8, uint16, uint32, int8, int16, or int32. The data type of the codebook values can be double, single, uint8, uint16, uint32, int8, int16, or int32.

In Encoder mode, the output of the block is the index values. In Encoder and decoder mode, the output can also include the quantized output values and the quantization error. In Encoder and Encoder and decoder mode, use the **Output index data type** parameter to specify the data type of the index output from the block at port Idx. The data type of the index output can be uint8, uint16, uint32, int8, int16, or int32. The data type of the quantized output and the quantization error can be `double`, `single`, `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. In `Decoder` mode, the output of the block is the quantized output values. Use the **Output data type** parameter to specify the data type of the quantized output values. The data type can be `double`, `single`, `uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`.

Note The input data, codebook values, boundary points, quantization error, and the quantized output values must have the same data type whenever they are present in any of the quantizer modes.

Parameters

Quantizer mode

Specify `Encoder`, `Decoder`, or `Encoder and decoder` as a mode of operation.

Source of quantizer parameters

Choose `Specify via dialog` to type the parameters into the block parameters dialog box. Select `Input ports` to specify the parameters using the block's input ports. In `Encoder` and `Encoder and decoder` mode, input the **Boundary points** using port B. In `Decoder` and `Encoder and decoder` mode, input the **Codebook** values using port C.

Boundary points

Enter a vector of values that represent the boundary points of the quantizer regions. Tunable (Simulink).

Codebook

Enter a vector of quantized output values that correspond to each index value. Tunable (Simulink).

Searching method

Select `Linear` and the block finds the region in which the input value is located using a linear search. Select `Binary` and the block finds the region in which the input value is located using a binary search.

Tie-breaking rule

Set this parameter to determine the behavior of the block when the input value is the same as the boundary point. When you select `Choose the lower index`, the input value is assigned to lower index value. When you select `Choose the higher index`, the value is assigned to the higher index.

Action for out of range input

Choose the block's behavior when an input index value is out of range, where 0 ≤ index <*N* and *N* is the length of the codebook vector. Select `Clip`, when you want any index values less than 0 to be set to 0 and any index values greater than or equal to *N* to be set to *N* -1. Select `Clip and warn`, when you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to *N* are set to *N* -1. Select `Error`, when you want the simulation to stop and display an error when the index values are out of range.

Output the quantization error

In `Encoder and decoder` mode, select this check box to output the quantization error from the Err port on this block.

Output index data type

In `Encoder` and `Encoder and decoder` mode, specify the data type of the index output from the block at port Idx. The data type can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. This parameter becomes visible when you select the **Show additional parameters** check box.

Output data type

In `Decoder` mode, specify the data type of the quantized output. The data type can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`, `single`, or `double`. This parameter becomes visible when you select `Specify via dialog` for the **Source of quantizer parameters** and you select the **Show additional parameters** check box.

References

Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

- 8-, 16-, and 32-bit unsigned integers

For more information on what data types are supported for each quantizer mode, see "Data Type Support" on page 2-1488.

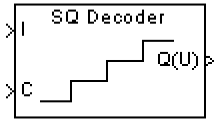
See Also

Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Scalar Quantizer Design	DSP System Toolbox
Scalar Quantizer Encoder	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
Uniform Decoder	DSP System Toolbox

Introduced in R2008b

Scalar Quantizer Decoder

Convert each index value into quantized output value



Scalar Quantizer Decoder

Library

Quantizers

`dspquant2`

Description

The Scalar Quantizer Decoder block transforms the zero-based input index values into quantized output values. The set of all possible quantized output values is defined by the **Codebook values** parameter.

Use the **Codebook values** parameter to specify a matrix containing all possible quantized output values. You can select how you want to enter the codebook values using the **Source of codebook** parameter. When you select `Specify via dialog`, type the codebook values into the block parameters dialog box. When you select `Input port`, port C appears on the block. The block uses the input to port C as the **Codebook values** parameter.

The input to this block is a vector of integer index values, where $0 \leq \text{index} < N$ and N is the number of distinct codeword vectors in the codebook matrix. Use the **Action for out of range index value** parameter to determine what happens when an input index value is outside this range. When you want any index value less than 0 to be set to 0 and any index value greater than or equal to N to be set to N -1, select `Clip`. When you want to be warned when clipping occurs, select `Clip and warn`. When you want the simulation to stop and the block to display an error when the index values are out of range, select `Error`.

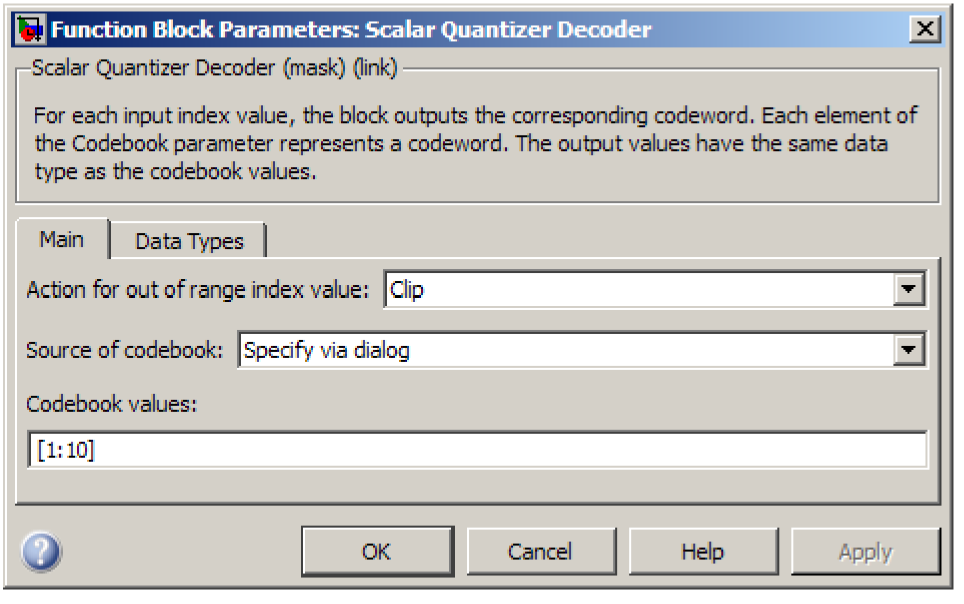
Data Type Support

The data type of the index values input at port I can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. The data type of the codebook values input at port C can be `double`, `single`, or Fixed-point.

The output of the block is the quantized output values. If, for the **Source of codebook** parameter, you select `Specify via dialog`, the **Codebook and output data type** parameter appears. You can use this parameter to specify the data type of the codebook and quantized output values. In this case, the data type of the output values can be `Same as input`, `double`, `single`, `Fixed-point`, or `User-defined`. If, for the **Source of codebook** parameter you select `Input port`, the quantized output values have the same data type as the codebook values input at port C.

Dialog Box

The **Main** pane of the Scalar Quantizer Decoder block dialog appears as follows.



Action for out of range index value

Use this parameter to determine the block's behavior when an input index value is out of range, where $0 \leq$ index $< N$ and N is the length of the codebook vector. Select `Clip`, when you want any index values less than 0 to be set to 0 and any index values greater than or equal to N to be set to N -1. Select `Clip and warn`, when you want to be warned when clipping occurs. Select `Error`, when you want the simulation to stop and the block to display an error when the index values are outside the range.

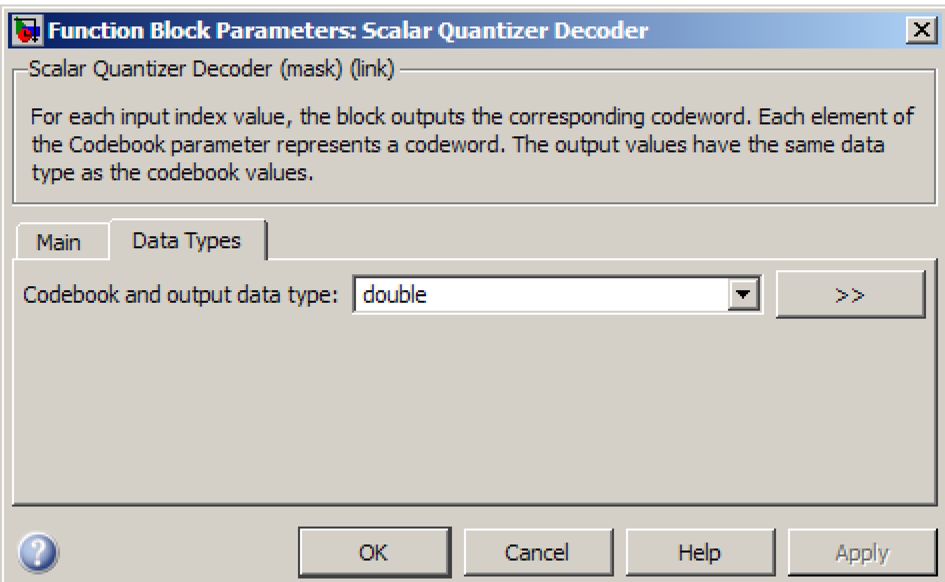
Source of codebook

Choose `Specify via dialog` to type the codebook values into the block parameters dialog box. Select `Input port` to specify the codebook using input port C.

Codebook values

Enter a vector of quantized output values that correspond to each index value. Tunable (Simulink).

The **Data Types** pane of the Scalar Quantizer Decoder block dialog appears as follows.



Codebook and output data type

Specify the data type of the codebook and quantized output values. You can select one of the following:

- A rule that inherits a data type, for example, `Inherit: Same as input`.

- A built in data type, such as `double`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

This parameter is available only when you set the **Source of codebook** parameter to `Specify via dialog`. If you set the **Source of codebook** parameter to `Input port`, the output values have the same data type as the input codebook values.

References

Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

Port	Supported Data Types
I	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
C	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers
Q(U)	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

For more information on what data types are supported for each quantizer mode, see “Data Type Support” on page 2-1493.

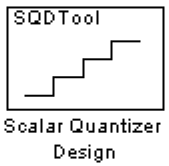
See Also

Quantizer	Simulink
Scalar Quantizer Design	DSP System Toolbox
Scalar Quantizer Encoder	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
Uniform Decoder	DSP System Toolbox

Introduced before R2006a

Scalar Quantizer Design

Start Scalar Quantizer Design Tool (SQDTool) to design scalar quantizer using Lloyd algorithm



Library

Quantizers

`dspquant2`

Description

Double-click on the Scalar Quantizer Design block to start SQDTool, a GUI that allows you to design and implement a scalar quantizer. Based on your input values, SQDTool iteratively calculates the codebook values that minimize the mean squared error until the stopping criteria for the design process is satisfied. The block uses the resulting quantizer codebook values and boundary points to implement your scalar quantizer encoder and/or decoder.

For the **Training Set** parameter, enter a set of observations, or samples, of the signal you want to quantize. This data can be any variable defined in the MATLAB workspace including a variable created using a MATLAB function, such as the default value `randn(10000,1)`.

You have two choices for the **Source of initial codebook** parameter. Select `Auto-generate` to have the block choose the values of the initial codebook vector. In this case, the minimum training set value becomes the first codeword, and the maximum training set value becomes the last codeword. Then, the remaining initial codewords are equally spaced between these two values to form a codebook vector of length N, where N is the **Number of levels** parameter. When you select `User defined`, enter the initial codebook values in the **Initial codebook** field. Then, set the **Source**

of initial boundary points parameter. You can select `Mid-points` to locate the boundary points at the midpoint between the codewords. To calculate the mid-points, the block internally arranges the initial codebook values in ascending order. You can also choose `User defined` and enter your own boundary points in the **Initial boundary points (unbounded)** field. Only one boundary point can be located between two codewords. When you select `User defined` for the **Source of initial boundary points** parameter, the values you enter in the **Initial codebook** and **Initial boundary points (unbounded)** fields must be arranged in ascending order.

Note This block assumes that you are designing an unbounded quantizer. Therefore, the first and last boundary points are always `-inf` and `inf` regardless of any other boundary point values you might enter.

After you have specified the quantization parameters, the block performs an iterative process to design the optimal scalar quantizer. Each step of the design process involves using the Lloyd algorithm to calculate codebook values and quantizer boundary points. Then, the block calculates the squared quantization error and checks whether the stopping criteria has been satisfied.

The two possible options for the **Stopping criteria** parameter are `Relative threshold` and `Maximum iteration`. When you want the design process to stop when the fractional drop in the squared quantization error is below a certain value, select `Relative threshold`. Then, for **Relative threshold**, type the maximum acceptable fractional drop. When you want the design process to stop after a certain number of iterations, choose `Maximum iteration`. Then, enter the maximum number of iterations you want the block to perform in the **Maximum iteration** field. For **Stopping criteria**, you can also choose `Whichever comes first` and enter a **Relative threshold** and **Maximum iteration** value. The block stops iterating as soon as one of these conditions is satisfied.

With each iteration, the block quantizes the training set values based on the newly calculated codebook values and boundary points. When the training point lies on a boundary point, the algorithm uses the **Tie-breaking rules** parameter to determine which region the value is associated with. When you want the training point to be assigned to the lower indexed region, select `Lower indexed codeword`. To assign the training point with the higher indexed region, select `Higher indexed codeword`.

The **Searching methods** parameter determines how the block compares the training points to the boundary points. Select `Linear search` and SQDTool compares each training point to each quantization region sequentially. This process continues until all the training points are associated with the appropriate regions.

Select `Binary search` for the **Searching methods** parameter and the block compares the training point to the middle value of the boundary points vector. When the training point is larger than this boundary point, the block discards the lower boundary points. The block then compares the training point to the middle boundary point of the new range, defined by the remaining boundary points. This process continues until the training point is associated with the appropriate region.

Click **Design and Plot** to design the quantizer with the parameter values specified on the left side of the GUI. The performance curve and the staircase character of the quantizer are updated and displayed in the figures on the right side of the GUI.

Note You must click **Design and Plot** to apply any changes you make to the parameter values in the SQDTool dialog box.

SQDTool can export parameter values that correspond to the figures displayed in the GUI. Click the **Export Outputs** button, or press **Ctrl+E**, to export the **Final Codebook**, **Final Boundary Points**, and **Error** values to the workspace, a text file, or a MAT-file. The **Error** values represent the mean squared error for each iteration.

In the **Model** section of the GUI, specify the destination of the block that will contain the parameters of your quantizer. For **Destination**, select `Current model` to create a block with your parameters in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Select `New model` to create a block in a new model file.

From the **Block type** list, select `Encoder` to design a Scalar Quantizer Encoder block. Select `Decoder` to design a Scalar Quantizer Decoder block. Select `Both` to design a Scalar Quantizer Encoder block and a Scalar Quantizer Decoder block.

In the **Encoder block name** field, enter a name for the Scalar Quantizer Encoder block. In the **Decoder block name** field, enter a name for the Scalar Quantizer Decoder block. When you have a Scalar Quantizer Encoder and/or Decoder block in your destination model with the same name, select the **Overwrite target block(s)** check box to replace the block's parameters with the current parameters. When you do not select this check box, a new Scalar Quantizer Encoder and/or Decoder block is created in your destination model.

Click **Generate Model**. SQDTool uses the parameters that correspond to the current plots to set the parameters of the Scalar Quantizer Encoder and/or Decoder blocks.

Parameters

Training Set

Enter the samples of the signal you would like to quantize. This data set can be a MATLAB function or a variable defined in the MATLAB workspace. The typical length of this data vector is `1e6`.

Source of initial codebook

Select `Auto-generate` to have the block choose the initial codebook values. Select `User defined` to enter your own initial codebook values.

Number of levels

Enter the length of the codebook vector. For a b-bit quantizer, the length should be $N = 2^b$.

Initial codebook

Enter your initial codebook values. From the **Source of initial codebook** list, select `User defined` in order to activate this parameter.

Source of initial boundary points

Select `Mid-points` to locate the boundary points at the midpoint between the codebook values. Choose `User defined` to enter your own boundary points. From the **Source of initial codebook** list, select `User defined` in order to activate this parameter.

Initial boundary points (unbounded)

Enter your initial boundary points. This block assumes that you are designing an unbounded quantizer. Therefore, the first and last boundary point are `-inf` and `inf`, regardless of any other boundary point values you might enter. From the **Source of initial boundary points** list, select `User defined` in order to activate this parameter.

Stopping criteria

Choose `Relative threshold` to enter the maximum acceptable fractional drop in the squared quantization error. Choose `Maximum iteration` to specify the number of iterations at which to stop. Choose `Whichever comes first` and the block stops the iteration process as soon as the relative threshold or maximum iteration value is attained.

Relative threshold

Type the value that is the maximum acceptable fractional drop in the squared quantization error.

Maximum iteration

Enter the maximum number of iterations you want the block to perform. From the **Stopping criteria** list, select `Maximum iteration` in order to activate this parameter.

Searching methods

Choose `Linear search` to use a linear search method when comparing the training points to the boundary points. Choose `Binary search` to use a binary search method when comparing the training points to the boundary points.

Tie-breaking rules

When a training point lies on a boundary point, choose `Lower indexed codeword` to assign the training point to the lower indexed quantization region. Choose `Higher indexed codeword` to assign the training point to the higher indexed region.

Design and Plot

Click this button to display the performance curve and the staircase character of the quantizer in the figures on the right side of the GUI. These plots are based on the current parameter settings.

You must click **Design and Plot** to apply any changes you make to the parameter values in the SQDTool GUI.

Export Outputs

Click this button, or press **Ctrl+E**, to export the **Final Codebook**, **Final Boundary Points**, and **Error** values to the workspace, a text file, or a MAT-file.

Destination

Choose `Current model` to create a Scalar Quantizer block in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Choose `New model` to create a block in a new model file.

Block type

Select `Encoder` to design a Scalar Quantizer Encoder block. Select `Decoder` to design a Scalar Quantizer Decoder block. Select `Both` to design a Scalar Quantizer Encoder block and a Scalar Quantizer Decoder block.

Encoder block name

Enter a name for the Scalar Quantizer Encoder block.

Decoder block name

Enter a name for the Scalar Quantizer Decoder block.

Overwrite target block(s)

When you do not select this check box and a Scalar Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, a new Scalar Quantizer Encoder and/or Decoder block is created in the destination model. When you select this check box and a Scalar Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, the parameters of these blocks are overwritten by new parameters.

Generate Model

Click this button and SQDTool uses the parameters that correspond to the current plots to set the parameters of the Scalar Quantizer Encoder and/or Decoder blocks.

References

Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

- Double-precision floating point

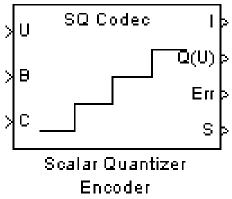
See Also

Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Scalar Quantizer Encoder	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
Uniform Decoder	DSP System Toolbox

Introduced before R2006a

Scalar Quantizer Encoder

Encode each input value by associating it with index value of quantization region



Library

Quantizers

dspquant2

Description

The Scalar Quantizer Encoder block maps each input value to a quantization region by comparing the input value to the quantizer boundary points defined in the **Boundary points** parameter. The block outputs the zero-based index of the associated region.

You can select how you want to enter the **Boundary points** using the **Source of quantizer parameters**. When you select `Specify via dialog`, type the boundary points into the block parameters dialog box. When you select `Input port`, port B appears on the block. The block uses the input to port B as the **Boundary points** parameter.

Use the **Boundary points** parameter to specify the boundary points for your quantizer. These values are used to break up the set of input numbers into regions. Each region is specified by an index number.

Let `N` be the number of quantization regions. When the codebook is defined as `[c1 c2 c3 ... cN]`, and the **Boundary points** parameter is defined as `[p0 p1 p2 p3 ... pN]`, then `p0<c1<p1<c2 ... p(N-1)<cN<pN` for a regular quantizer. When your

quantizer is bounded, from the **Partitioning** list, select `Bounded`. You need to specify N +1 boundary points, or `[p0 p1 p2 p3 ... pN]`. When your quantizer is unbounded, from the **Partitioning** list, select `Unbounded`. You need to specify N-1 boundary points, or`[p1 p2 p3 ... p(N-1)]`; the block sets `p0` equal to `-inf` and `pN` equal to `inf`.

The block uses the **Partitioning** parameter to interpret the boundary points you enter. For instance, to create a bounded quantizer, from the **Partitioning** list, select `Bounded` and enter the following boundary points:

```
[0 0.5 3.7 5.8 6.0 11]
```

The block assigns any input values between 0 and 0.5 to index 0, input values between 0.5 and 3.7 to index 1, and so on. The block assigns any values that are less than 0 to index 0, the lowest index value. The block assigns any values that are greater than 11 to index 4, the highest index value.

To create an unbounded quantizer, from the **Partitioning** list, select `Unbounded` and enter the following boundary points:

```
[0 0.5 3.7 5.8 6.0 11]
```

The block assigns any input values between 0 and 0.5 to index 1, input values between 0.5 and 3.7 to index 2, and so on. The block assigns any input values less than 0 to index 0 and any values greater than 11 to index 6.

The **Searching method** parameter determines how the appropriate quantizer index is found. When you select `Linear`, the Scalar Quantizer Encoder block compares the input value to the first region defined by the first two boundary points. When the input value does not fall within this region, the block then compares the input value to the next region. This process continues until the input value is determined to be within a region and is associated with the appropriate index value. The computational cost of this process is of the order P, where P is the number of boundary points.

When you select `Binary` for the **Searching method**, the block compares the input value to the middle value of the boundary points vector. When the input value is larger than this boundary point, the block discards the boundary points that are lower than this middle value. The block then compares the input value to the middle boundary point of the new range, defined by the remaining boundary points. This process continues until the input value is associated with the appropriate index value. The computational cost of this process is of the order $\log_2 P$, where P is the number of boundary points. In most cases, the `Binary` option is faster than the `Linear` option.

When an input value is the same as a boundary point, the **Tie-breaking rule** parameter determines the region to which the value is assigned. When you want the input value to be assigned to the lower indexed region, select `Choose the lower index`. To assign the input value with the higher indexed region, select `Choose the higher index`.

Select the **Output codeword** check box to output the codeword values that correspond to each index value at port Q(U).

Select the **Output the quantization error** check box to output the quantization error for each input value from the Err port on this block. The quantization error is the difference between the input value and the quantized output value.

When you select either the **Output codeword** check box or the **Output quantization error** check box, you must also enter your codebook values. If, from the **Source of quantizer parameters** list, you choose `Specify via dialog`, use the **Codebook** parameter to enter a vector of quantized output values that correspond to each region. If, from the **Source of quantizer parameters** list, you choose `Input port`, use input port C to specify your codebook values.

If, for the **Partitioning** parameter, you select `Bounded`, the **Output clipping status** check box and the **Action for out of range input** parameter appear. When you select the **Output clipping status** check box, port S appears on the block. Any time an input value is outside the range defined by the **Boundary points** parameter, the block outputs a 1 at the S port. When the value is inside the range, the blocks outputs a 0.

You can use the **Action for out of range input** parameter to determine the block's behavior when an input value is outside the range defined by the **Boundary points** parameter. Suppose the boundary points for a bounded quantizer are defined as `[p0 p1 p2 p3 ... pN]` and the possible index values are defined as `[i0 i1 i2 ... i(N-1)]`, where `i0=0` and `i0<i1<i2<...<i(N-1)`. When you want any input value less than `p0` to be assigned to index value `i0` and any input values greater than `pN` to be assigned to index value `i(N-1)`, select `Clip`. When you want to be warned when clipping occurs, select `Clip and warn`. When you want the simulation to stop and the block to display an error when the index values are out of range, select `Error`.

The Scalar Quantizer Encoder block accepts real floating-point and fixed-point inputs. For more information on the data types accepted by each port, see "Data Type Support" on page 2-1506 or "Supported Data Types" on page 2-1511.

Data Type Support

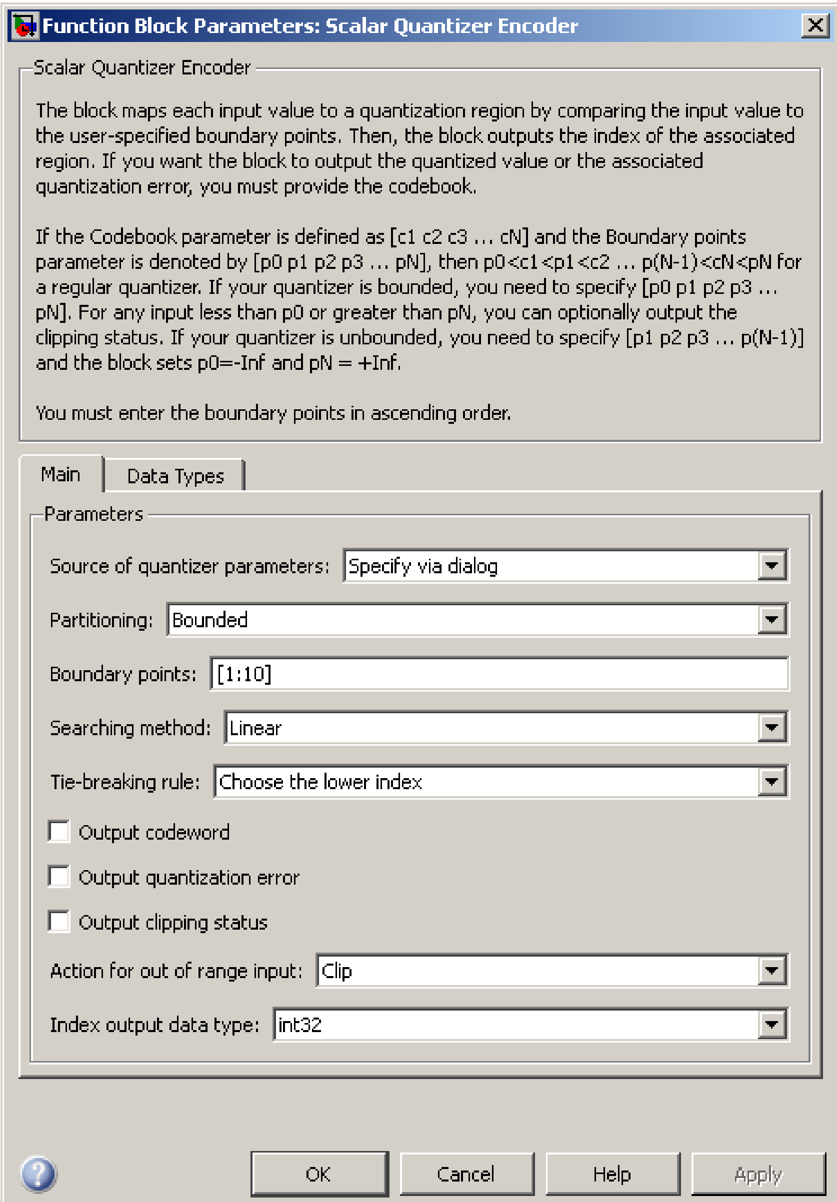
The input data values, boundary points, and codebook values can be input to the block at ports U, B, and C, respectively. The data type of the inputs can be `double`, `single`, or Fixed-point.

The outputs of the block can be the index values, the quantized output values, the quantization error, and the clipping status. Use the **Index output data type** parameter to specify the data type of the index output from the block at port I. You can choose `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`. The data type of the quantized output and the quantization error can be `double`, `single`, or Fixed-point. The clipping status values output at port S are Boolean values.

Note The input data, boundary points, codebook values, quantized output values, and the quantization error must have the same data type whenever they are present.

Dialog Box

The **Main** pane of the Scalar Quantizer Encoder block dialog appears as follows.



Source of quantizer parameters

Choose `Specify via dialog` to enter the boundary points and codebook values using the block parameters dialog box. Select `Input port` to specify the parameters using the block's input ports. Input the boundary points and codebook values using ports B and C, respectively.

Partitioning

When your quantizer is bounded, select `Bounded`. When your quantizer is unbounded, select `Unbounded`.

Boundary points

Enter a vector of values that represent the boundary points of the quantizer regions. This parameter is visible when you select `Specify via dialog` from the **Source of quantizer parameters** list. Tunable (Simulink).

Searching method

When you select `Linear`, the block finds the region in which the input value is located using a linear search. When you select `Binary`, the block finds the region in which the input value is located using a binary search.

Tie-breaking rule

Set this parameter to determine the behavior of the block when the input value is the same as the boundary point. When you select `Choose the lower index`, the input value is assigned to lower indexed region. When you select `Choose the higher index`, the value is assigned to the higher indexed region.

Output codeword

Select this check box to output the codeword values that correspond to each index value at port Q(U).

Output quantization error

Select this check box to output the quantization error for each input value at port Err.

Codebook

Enter a vector of quantized output values that correspond to each index value. If, for the **Partitioning** parameter, you select `Bounded` and your boundary points vector has length N, then you must specify a codebook of length N-1. If, for the **Partitioning** parameter, you select `Unbounded` and your boundary points vector has length N, then you must specify a codebook of length N+1.

This parameter is visible when you select `Specify via dialog` from the **Source of quantizer parameters** list and you select either the **Output codeword** or **Output quantization error** check box. Tunable (Simulink).

Output clipping status

When you select this check box, port S appears on the block. Any time an input value is outside the range defined by the **Boundary points** parameter, the block outputs a 1 at this port. When the value is inside the range, the block outputs a 0. This parameter is visible when you select `Bounded` from the **Partitioning** list.

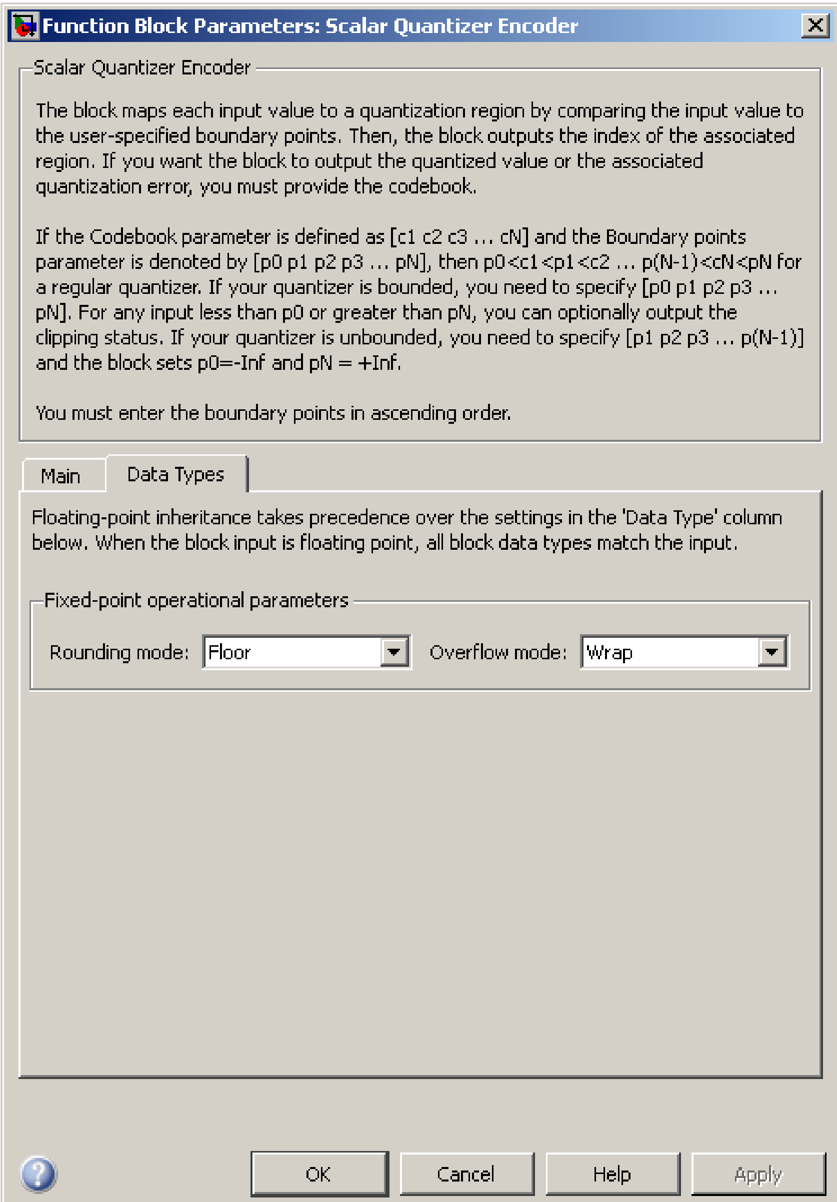
Action for out of range input

Use this parameter to determine the behavior of the block when an input value is outside the range defined by the **Boundary points** parameter. Suppose the boundary points are defined as `[p0 p1 p2 p3 ... pN]` and the index values are defined as `[i0 i1 i2 ... i(N-1)]`. When you want any input value less than `p0` to be assigned to index value `i0` and any input values greater than `pN` to be assigned to index value `i(N-1)`, select `Clip`. When you want to be warned when clipping occurs, select `Clip and warn`. When you want the simulation to stop and the block to display an error when the index values are out of range, select `Error`. This parameter is visible when you select `Bounded` from the **Partitioning** list.

Index output data type

Specify the data type of the index output from the block at port I. You can choose `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`.

The **Data Types** pane of the Scalar Quantizer Encoder block dialog appears as follows.



Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode to be used when block inputs are fixed point.

References

Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

Port	Supported Data Types
U	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
B	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
C	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
I	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Q(U)	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
Err	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
S	• Boolean

For more information on what data types are supported for each quantizer mode, see "Data Type Support" on page 2-1506.

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Scalar Quantizer Design	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
Uniform Decoder	DSP System Toolbox

Introduced before R2006a

Selector

Select input elements from vector, matrix, or multidimensional signal

Library

Signal Management / Indexing

`dspindex`

Description

The Selector block is an implementation of the Simulink Selector block. See Selector for more information.

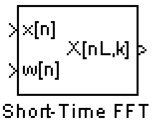
HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Selector.

Introduced in R2008a

Short-Time FFT

Nonparametric estimate of spectrum using short-time, fast Fourier transform (FFT) method



Library

Transforms

`dspxfrm3`

Description

The Short-Time FFT block computes a nonparametric estimate of the spectrum. The block buffers, applies a window, and zero pads the input signal. The block then takes the FFT of the signal, transforming it into the frequency domain.

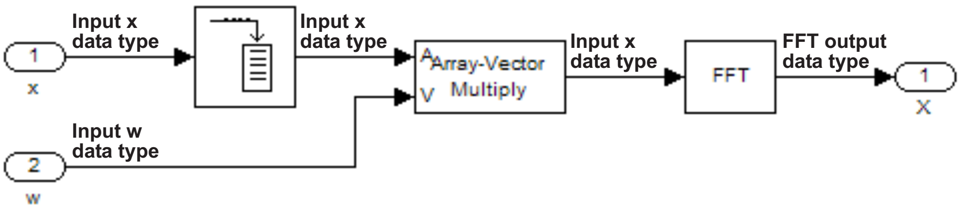
Connect your single-channel analysis window to the w(n) port. For the **Analysis window length** parameter, enter the length of the analysis window, *W*. The block buffers the input signal such that it has a frame length of *W*

Connect your single-channel or multichannel input signal to the x(n) port. After the block buffers and windows this signal, it zero-pads the signal before computing the FFT. For the **FFT length** parameter, enter the length to which the block pads the input signal. For the **Overlap between consecutive windows (in samples)** parameter, enter the number of samples to overlap each frame of the input signal.

The block outputs the complex-valued, single-channel or multichannel short-time FFT at port X(n,k).

Fixed-Point Data Types

The following diagram shows the data types used within the Short-Time FFT subsystem block for fixed-point signals.



The settings for the fixed-point parameters of the Array-Vector Multiply block in the diagram above are as follows:

- **Rounding Mode** — `Floor`
- **Overflow Mode** — `Wrap`
- **Product output** — `Inherit via internal rule`
- **Accumulator** — `Inherit via internal rule`
- **Output** — `Same as first input`

The settings for the fixed-point parameters of the FFT block in the diagram above are as follows:

- **Rounding Mode** — `Floor`
- **Overflow Mode** — `Wrap`
- **Sine table** — `Same word length as input`
- **Product output** — `Inherit via internal rule`
- **Accumulator** — `Inherit via internal rule`
- **Output** — `Inherit via internal rule`

See the `FFT` and `Array-Vector Multiply` block reference pages for more information.

Examples

The dspstsa example illustrates how to use the Short-Time FFT and Inverse Short-Time FFT blocks to remove the background noise from a speech signal. To open the `dspstsa` model, type `dspstsa` in the MATLAB command prompt.

Parameters

Analysis window length

Specify the frame length of the analysis window. The **Analysis window length** must be a positive integer value greater than one.

Overlap between consecutive windows (in samples)

Enter the number of samples of overlap for each frame of the input signal.

FFT length

Enter the length to which the block pads the input signal.

Treat Mx1 and unoriented sample-based signals as

Specify how the block treats sample-based M-by-1 column vectors and unoriented sample-based vectors of length M. You can select one of the following options:

- `One channel` — When you select this option, the block treats M-by-1 and unoriented sample-based inputs as a column vector (one channel).
- `M channels (this choice will be removed — see release notes)` — When you select this option, the block treats M-by-1 and unoriented sample-based inputs as a 1-by-M row vector.

Note: This parameter will be removed in a future release. See the *DSP System Toolbox Release Notes* for more information.

Supported Data Types

Port	Supported Data Types
x(n)	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
w(n)	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
X(n,k)	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

References

[1] Quatieri, Thomas E. *Discrete-Time Speech Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 2001.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

When the FFT length is not a power of two, the executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "How To Run a Generated Executable Outside MATLAB".

When the FFT length is a power of two, you can generate standalone C and C++ code from this block.

See Also

See Also

Functions

`pwelch`

Blocks

`Burg Method` | `Inverse Short-Time FFT` | `Magnitude FFT` | `Periodogram` | `Spectrum Analyzer` | `Window Function` | `Yule-Walker Method`

Topics

"Spectral Analysis"

Introduced before R2006a

Signal From Workspace

Import signal from MATLAB workspace

Library

Sources

`dspsrcs4`

Description

The Signal From Workspace block imports a signal from the MATLAB workspace into the Simulink model. The **Signal** parameter specifies the name of a MATLAB workspace variable containing the signal to import, or any valid MATLAB expression defining a matrix or 3-D array.

When the **Signal** parameter specifies an M-by-N matrix (M≠1), each of the N columns is treated as a distinct channel. You specify the frame size in the **Samples per frame** parameter, M_o, and the output is an M_o-by-N matrix containing M_o consecutive samples from each signal channel. You specify the output sample period in the **Sample time** parameter, T_s, and the output frame period is M_o*T_s. For convenience, an imported row vector (M=1) is treated as a single channel, so the output dimension is M_o-by-1.

When the **Signal** parameter specifies an M-by-N-by-P array, each of the P pages (an M-by-N matrix) is output in sequence with period T_s. The **Samples per frame** parameter must be set to `1`.

Initial and Final Conditions

Unlike the Simulink From Workspace block, the Signal From Workspace block holds the output value constant between successive output frames (that is, no linear interpolation takes place). Additionally, the initial signal values are always produced immediately at t=0.

When the block has output all of the available signal samples, it can start again at the beginning of the signal, or simply repeat the final value or generate zeros until the end of the simulation. (The block does not extrapolate the imported signal beyond the last sample.) The **Form output after final data value by** parameter controls this behavior:

- When you specify `Setting To Zero`, the block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.
- When you specify `Holding Final Value`, the block repeats the final sample for the duration of the simulation after generating the last frame of the signal.
- When you specify `Cyclic Repetition`, the block repeats the signal from the beginning after it reaches the last sample in the signal. If the frame size you specify in the **Samples per frame** parameter does not evenly divide the input length, a buffer block is inserted into the Signal From Workspace subsystem, and the model becomes multirate. If you do not want your model to become multirate, make sure the frame size evenly divides the input signal length.

  Select the **Warn when frame size does not evenly divide input length** parameter to be alerted when the input length is not an integer multiple of the frame size and your model will become multirate. Use the Model Explorer to turn these warnings on or off model-wide:

  a Select **Model Explorer** from the **View** menu in your model.

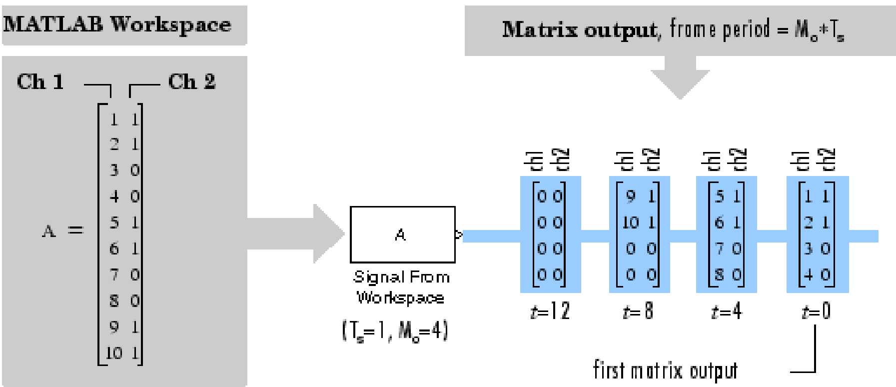
  b In the **Search** bar of the Model Explorer, search `by Property Name` for the `ignoreOrWarnInputAndFrameLengths` property. Each block with the **Warn when frame size does not evenly divide input length** check box appears in the list in the **Contents** pane.

  c Select each of the blocks for which you wish to toggle the warning parameter, and select or deselect the check box in the `ignoreOrWarnInputAndFrameLengths` column.

Examples

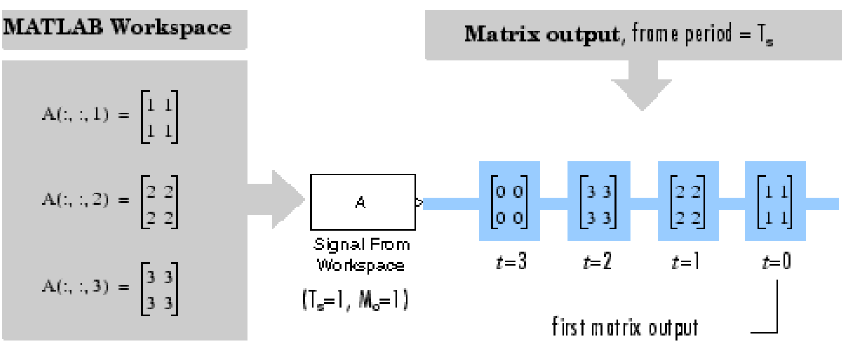
Example 1

In the first model, ex_signal_from_workspace_fb, the Signal From Workspace imports a two-channel signal from the workspace matrix,`A`. The **Sample time** is set to `1` and the **Samples per frame** is set to `4`. The output has a frame size of 4 and a frame period of 4 seconds. The **Form output after final data value by** parameter specifies `Setting To Zero`, so all outputs after the third frame (at *t*=8) are zero.



Example 2

In the second model, ex_signal_from_workspace_sb, the Signal From Workspace block imports a matrix signal from the 3-D workspace array, `A`. Again, the **Form output after final data value by** parameter specifies `Setting To Zero`, so all outputs after the third (at *t*=2) are zero.



The **Samples per frame** parameter is set to `1` for 3-D input.

Parameters

Signal

The name of the MATLAB workspace variable from which to import the signal, or a valid MATLAB expression specifying the signal.

Sample time

The sample period, T_s, of the output. The output frame period is M_o*T_s.

Samples per frame

The number of samples, M_o, to buffer into each output frame. This value must be `1` when you specify a 3-D array in the **Signal** parameter.

Form output after final data value by

Specifies the output after all of the specified signal samples have been generated. The block can output zeros for the duration of the simulation (`Setting to zero`), repeat the final data sample (`Holding Final Value`) or repeat the entire signal from the beginning (`Cyclic Repetition`).

Warn when frame size does not evenly divide input length

Select this parameter to be alerted when the input length is not an integer multiple of the frame size and your model will become multirate. For more information, see "Initial and Final Conditions" on page 2-1519.

This parameter is only visible when `Cyclic Repetition` is selected for the **Form output after final data value by** parameter.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

See Also

Signal From Workspace	DSP System Toolbox
From Workspace	Simulink
To Workspace	Simulink

`Triggered Signal From Workspace`	DSP System Toolbox

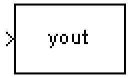
See the sections below for related information:

- "Create Signals for Sample-Based Processing"
- "Create Signals for Frame-Based Processing"
- "Import and Export Signals for Sample-Based Processing"
- "Import and Export Signals for Frame-Based Processing"

Introduced before R2006a

Signal To Workspace

Write data to MATLAB workspace



Note: The Signal To Workspace block has been replaced by the To Workspace block in Simulink. Existing instances of the Signal To Workspace block will continue to operate. For new models, use the To Workspace block.

Library

Sinks

`dspsnks4`

Description

The Signal To Workspace block writes data from your simulation into an array or structure in the main MATLAB workspace. You can specify a name for the workspace variable as well as whether the data is saved as an array, structure, or structure with time.

When the **Save format** is set to `Array` or `Structure`, the dimensions of the output depend on the input dimensions and the setting of the **Save 2-D signals as** parameter. The following table summarizes the output dimensions under various conditions. In the table, K represents the value of the **Limit data points to last** parameter.

Input Signal Dimensions	Save 2-D Signals as ...	Signal To Workspace Output Dimension
M-by-N matrix	`2-D array (concatenate along first dimension)`	K-by-N matrix.

Input Signal Dimensions	Save 2-D Signals as ...	Signal To Workspace Output Dimension
		If you set the **Limit data points to last** parameter to `inf`, *K* represents the total number of samples acquired in each column by the end of simulation. This is equivalent to multiplying the input frame size (*M*) by the total number of *M*-by-*N* inputs acquired by the block.
M-by-*N* matrix	`3-D array (concatenate along third dimension)`	*M*-by-*N*-by-*K* array. If you set the **Limit data points to last** parameter to `inf`, *K* represents the total number of *M*-by-*N* inputs acquired by the end of the simulation.
Length-*N* unoriented vector	Any setting	*K*-by-*N* matrix
N-dimensional array where $N > 2$	Any setting	Array with *N+1* dimensions, where the size of the last dimension is equal to *K*. If you set the **Limit data points to last** parameter to `inf`, *K* represents the total number of *M*-by-*N* inputs acquired by the end of simulation

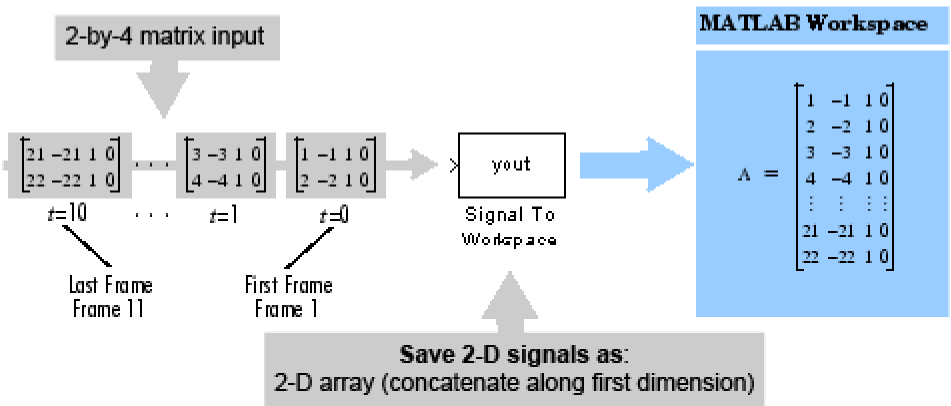
Examples

Example 1: Save 2-D Signals as a 2-D Array

In the ex_signaltoworkspace_ref2 model, the Signal To Workspace block receives a 2-by-4 matrix input and logs 11 frames (two samples per frame) by the end of the simulation.

Because the **Save 2-D signals as** parameter is set to `2-D array (concatenate along first dimension)`, the block concatenates the input along the first dimension to create a 22-by-4 matrix, *A*, in the MATLAB workspace.

The following figure illustrates the behavior of the Signal to Workspace block in this example.

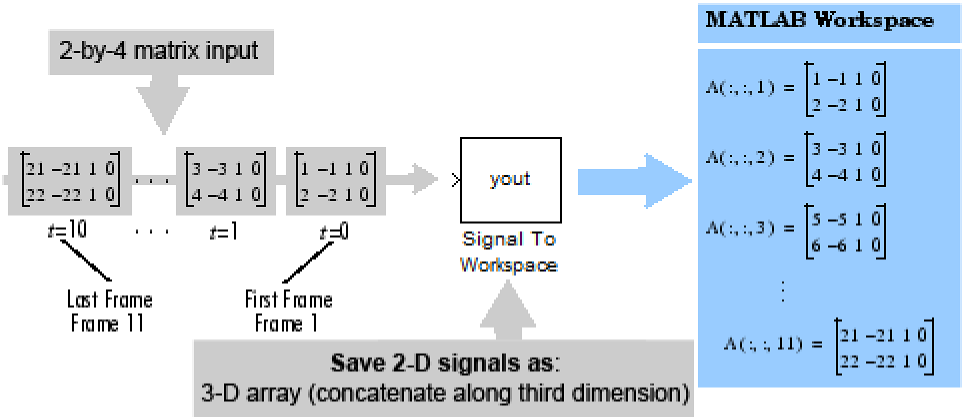


In the 2-D output mode, there is no indication of where one frame ends and another begins. To log input frames separately, set the **Save 2-D signals as** parameter to `3-D array (concatenate along third dimension)`, as shown in Example 2.

Example 2: Save 2-D Signals as a 3-D Array

In the ex_signaltoworkspace_ref1 model, the input to the Signal To Workspace block is a 2-by-4 matrix. The **Save 2-D signals as** parameter is set to `3-D array (concatenate along third dimension)`, so by the end of the simulation the Signal To Workspace block logs 11 frames of data as a 2-by-4-by-11 array, *A*, in the MATLAB workspace.

The following figure illustrates the behavior of the Signal to Workspace block in this example.



Parameters

Variable name

Specify the name of the array or structure into which the block logs the simulation data. The block creates this variable in the MATLAB workspace only after the simulation stops running. When you enter the name of an existing workspace variable, the block overwrites that variable with the simulation data.

Limit data points to last

Specify the maximum number of samples or frames the block will save. When the simulation generates more than the specified maximum number of samples or frames, the simulation saves only the most recently generated data. To capture all data, set this parameter to `inf`. See the table in the Description section for more information on how this parameter affects the dimensions of the logged data.

Decimation

Specify a positive integer *d* to determine how often the block writes data to the workspace array or structure. The block writes data to the array or structure every *d*th sample. With the default decimation value of `1`, the block writes data at every time step.

Save format

Specify the format in which to save simulation output to the workspace. You can select one of the following options:

- `Array` — Select this option to save the data as an N-dimensional array. If the input signal is an unoriented vector, the resulting workspace array is 2-D. Each input vector is saved in a row of the output matrix, vertically concatenated onto the previous vector. If the input signal is 2-dimensional, the dimensions of the resulting workspace array depend on the setting of the **Save 2-D signals as** parameter.
- `Structure` — Select this option to save the data as a structure consisting of three fields: `time`, `signals` and `blockName`. In this mode, the `time` field is empty, and the `blockName` field contains the name of the Signal To Workspace block. The `signals` field contains a structure with three additional fields: `values`, `dimensions`, and `label`. The `values` field contains the array of signal values, the `dimensions` field specifies the dimensions of the values array, and the `label` field contains the label of the input line.
- `Structure with time` — This option is the same as `Structure`, except that the `time` field contains a vector of simulation time steps. This is the only output format that can be read directly by a From Workspace block. When you select this option, the **Save 2-D signals as** parameter is not available. In this mode, the block always saves 2-D input arrays as a 3-D array.

The default setting of this parameter is `Array`.

Save 2-D signals as

Specify whether the block outputs 2-D signals as a 2-D or 3-D array in the MATLAB workspace:

- `2-D array (concatenate along first dimension)` — When you select this option, the block saves an M-by-N input signal as a (K*M)-by-N matrix, where K*M is the total number of samples acquired by the end of the simulation. The block vertically concatenates each M-by-N matrix input with the previous input to produce the 2-D output array. See "Example 1: Save 2-D Signals as a 2-D Array" on page 2-1525 for more information about this mode.
- `3-D array (concatenate along third dimension)` — When you select this option, the block saves an M-by-N input signal as an M-by-N-by-K array, where K is the number of M-by-N inputs logged by the end of the simulation. K has an upper bound equal to the value of the **Limit data points to last** parameter. See "Example 2: Save 2-D Signals as a 3-D Array" on page 2-1526 for more information about this mode.

This parameter is visible only when you set the **Save format** parameter to `Array` or `Structure`. When you set the **Save format** parameter to `Structure with time`, the block outputs the 2-D input signal as a 3-D array.

Note: The `Inherit from input (this choice will be removed - see release notes)` option will be removed in a future release. See "Signal To Workspace Block Changes" in the *DSP System Toolbox Release Notes* for more information.

Log fixed-point data as a fi object

Select this check box to log fixed-point data to the MATLAB workspace as a Fixed-Point Designer `fi` object. Otherwise, fixed-point data is logged to the workspace as `double`.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

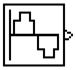
See Also

`Triggered To Workspace`	DSP System Toolbox
To Workspace	Simulink

Introduced before R2006a

Sine Wave

Generate continuous or discrete sine wave



Library

Sources

`dspsrcs4`

Description

The Sine Wave block generates a multichannel real or complex sinusoidal signal, with independent amplitude, frequency, and phase in each output channel. A real sinusoidal signal is generated when the **Output complexity** parameter is set to `Real`, and is defined by an expression of the type

$$y = A\sin(2\pi ft + \phi)$$

where you specify A in the **Amplitude** parameter, f in hertz in the **Frequency** parameter, and ϕ in radians in the **Phase offset** parameter. A complex exponential signal is generated when the **Output complexity** parameter is set to `Complex`, and is defined by an expression of the type

$$y = Ae^{j(2\pi ft+\phi)} = A\left\{\cos(2\pi ft + \phi) + j\sin(2\pi ft + \phi)\right\}$$

Sections of This Reference Page

- “Generating Multichannel Outputs” on page 2-1531
- “Output Sample Time and Samples Per Frame” on page 2-1531
- “Sample Mode” on page 2-1531
- “Discrete Computational Methods” on page 2-1532
- “Examples” on page 2-1535

- "Dialog Box" on page 2-1537
- "Supported Data Types" on page 2-1542
- "See Also" on page 2-1542

Generating Multichannel Outputs

For both real and complex sinusoids, the **Amplitude**, **Frequency**, and **Phase offset** parameter values (A, f, and ϕ) can be scalars or length-N vectors, where N is the desired number of channels in the output. When you specify at least one of these parameters as a length-N vector, scalar values specified for the other parameters are applied to every channel.

For example, to generate the three-channel output containing the real sinusoids below, set **Output complexity** to `Real` and the other parameters as follows:

- **Amplitude** = `[1 2 3]`
- **Frequency** = `[1000 500 250]`
- **Phase offset** = `[0 0 pi/2]`

$$y = \begin{cases} \sin(2000\pi t) & \text{(channel 1)} \\ 2\sin(1000\pi t) & \text{(channel 2)} \\ 3\sin\left(500\pi t + \dfrac{\pi}{2}\right) & \text{(channel 3)} \end{cases}$$

Output Sample Time and Samples Per Frame

In all discrete modes, the block buffers the sampled sinusoids into frames of size M, where you specify M in the **Samples per frame** parameter. The output is an M-by-N matrix with frame period M*T_s, where you specify T_s in the **Sample time** parameter.

Sample Mode

The **Sample mode** parameter specifies the block's sampling property, which can be `Continuous` or `Discrete`:

- `Continuous`

In continuous mode, the sinusoid in the ith channel, y_i, is computed as a continuous function,

$$y_i = A_i \sin(2\pi f_i t + \phi_i) \quad \text{(real)}$$

or

$$y_i = A_i e^{j(2\pi f_i t + \phi_i)} \qquad \text{(complex)}$$

and the block's output is continuous. In this mode, the block's operation is the same as that of a Simulink Sine Wave block with **Sample time** set to `0`. This mode offers high accuracy, but requires trigonometric function evaluations at each simulation step, which is computationally expensive. Additionally, because this method tracks absolute simulation time, a discontinuity will eventually occur when the time value reaches its maximum limit.

Note also that many DSP System Toolbox blocks do not accept continuous-time inputs.

- `Discrete`

  In discrete mode, the block's discrete-time output can be generated by directly evaluating the trigonometric function, by table lookup, or by a differential method. The three options are explained below.

Discrete Computational Methods

When you select `Discrete` from the **Sample mode** parameter, the secondary **Computation method** parameter provides three options for generating the discrete sinusoid:

- `Trigonometric Fcn`
- `Table Lookup`

- `Differential`

Note: To generate fixed-point sinusoids, you must select `Table Lookup`.

Trigonometric Fcn

The trigonometric function method computes the sinusoid in the *i*th channel, y_i, by sampling the continuous function

$$y_i = A_i \sin(2\pi f_i t + \phi_i) \quad \text{(real)}$$

or

$$y_i = A_i e^{j(2\pi f_i t + \phi_i)} \qquad \text{(complex)}$$

with a period of T_s, where you specify T_s in the **Sample time** parameter. This mode of operation shares the same benefits and liabilities as the `Continuous` sample mode described above.

At each sample time, the block evaluates the sine function at the appropriate time value *within the first cycle* of the sinusoid. By constraining trigonometric evaluations to the first cycle of each sinusoid, the block avoids the imprecision of computing the sine of very large numbers, and eliminates the possibility of discontinuity during extended operations (when an absolute time variable might overflow). This method therefore avoids the memory demands of the table lookup method at the expense of many more floating-point operations.

Table Lookup

The table lookup method precomputes the *unique* samples of every output sinusoid at the start of the simulation, and recalls the samples from memory as needed. Because a table of finite length can only be constructed when all output sequences repeat, the method requires that the period of every sinusoid in the output be evenly divisible by the sample period. That is, $1/(f_i T_s) = k_i$ must be an integer value for every channel $i = 1, 2, ..., N$.

When the **Optimize table for** parameter is set to `Speed`, the table constructed for each channel contains k_i elements. When the **Optimize table for** parameter is set to `Memory`, the table constructed for each channel contains $k_i/4$ elements.

For long output sequences, the table lookup method requires far fewer floating-point operations than any of the other methods, but can demand considerably more memory, especially for high sample rates (long tables). This is the recommended method for models that are intended to emulate or generate code for DSP hardware, and that therefore need to be optimized for execution speed.

Note: The lookup table for this block is constructed from double-precision floating-point values. Thus, when you use the `Table lookup` computation mode, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length of the **Output** or **User-defined** data type to values greater than 53 bits does not improve the precision of your output.

Differential

The differential method uses an incremental algorithm. This algorithm computes the output samples based on the output values computed at the previous sample time (and precomputed update terms) by making use of the following identities.

$$\sin(t+T_s) = \sin(t)\cos(T_s) + \cos(t)\sin(T_s)$$
$$\cos(t+T_s) = \cos(t)\cos(T_s) - \sin(t)\sin(T_s)$$

The update equations for the sinusoid in the ith channel, y_i, can therefore be written in matrix form as

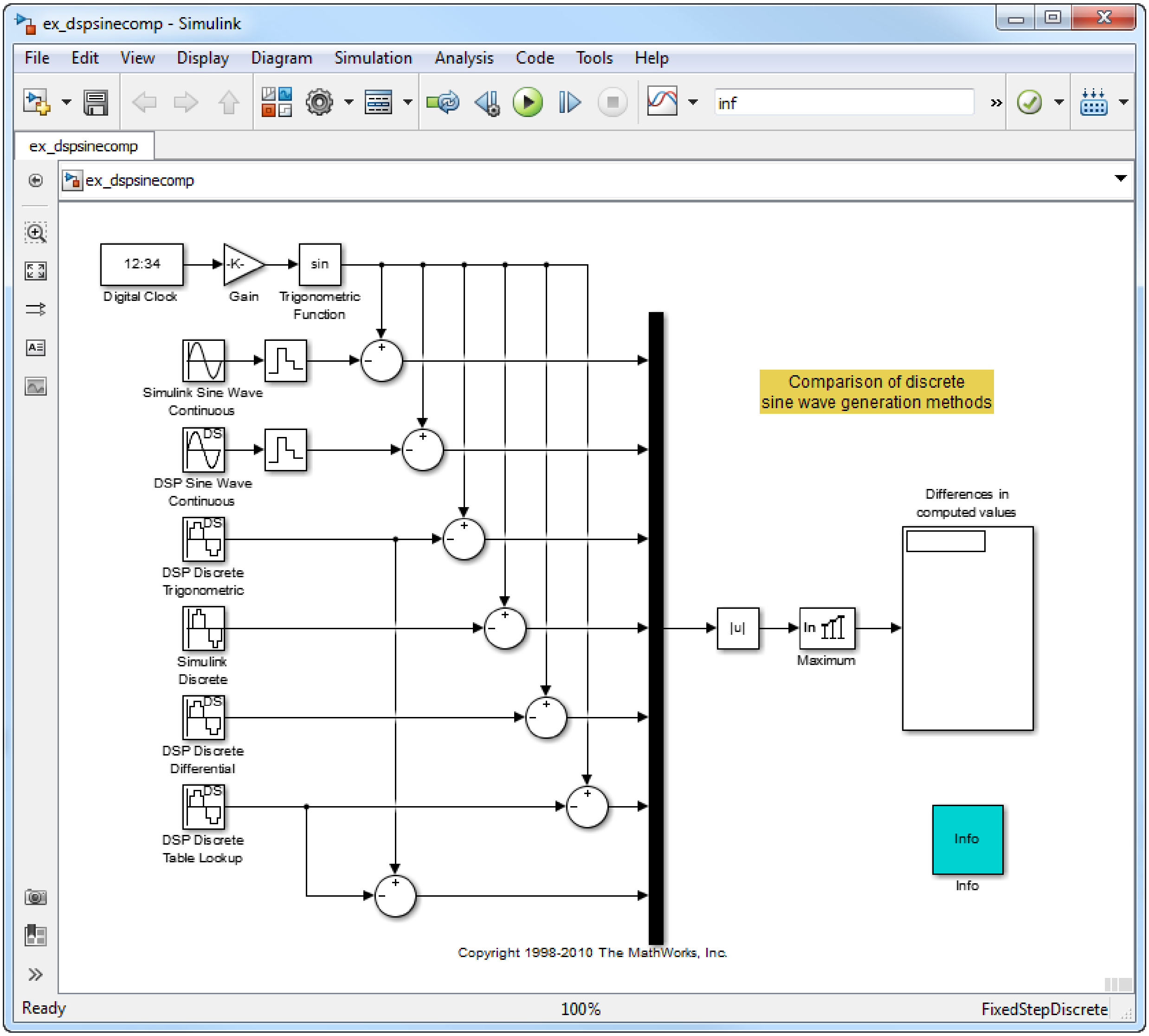
$$\begin{bmatrix} \sin\{2\pi f_i (t+T_s)+\phi_i\} \\ \cos\{2\pi f_i (t+T_s)+\phi_i\} \end{bmatrix} = \begin{bmatrix} \cos(2\pi f_i T_s) & \sin(2\pi f_i T_s) \\ -\sin(2\pi f_i T_s) & \cos(2\pi f_i T_s) \end{bmatrix} \begin{bmatrix} \sin(2\pi f_i t+\phi_i) \\ \cos(2\pi f_i t+\phi_i) \end{bmatrix}$$

where you specify T_s in the **Sample time** parameter. Since T_s is constant, the right-hand matrix is a constant and can be computed once at the start of the simulation. The value of $A_i\sin[2\pi f_i(t+T_s)+\phi_i]$ is then computed from the values of $\sin(2\pi f_i t+\phi_i)$ and $\cos(2\pi f_i t+\phi_i)$ by a simple matrix multiplication at each time step.

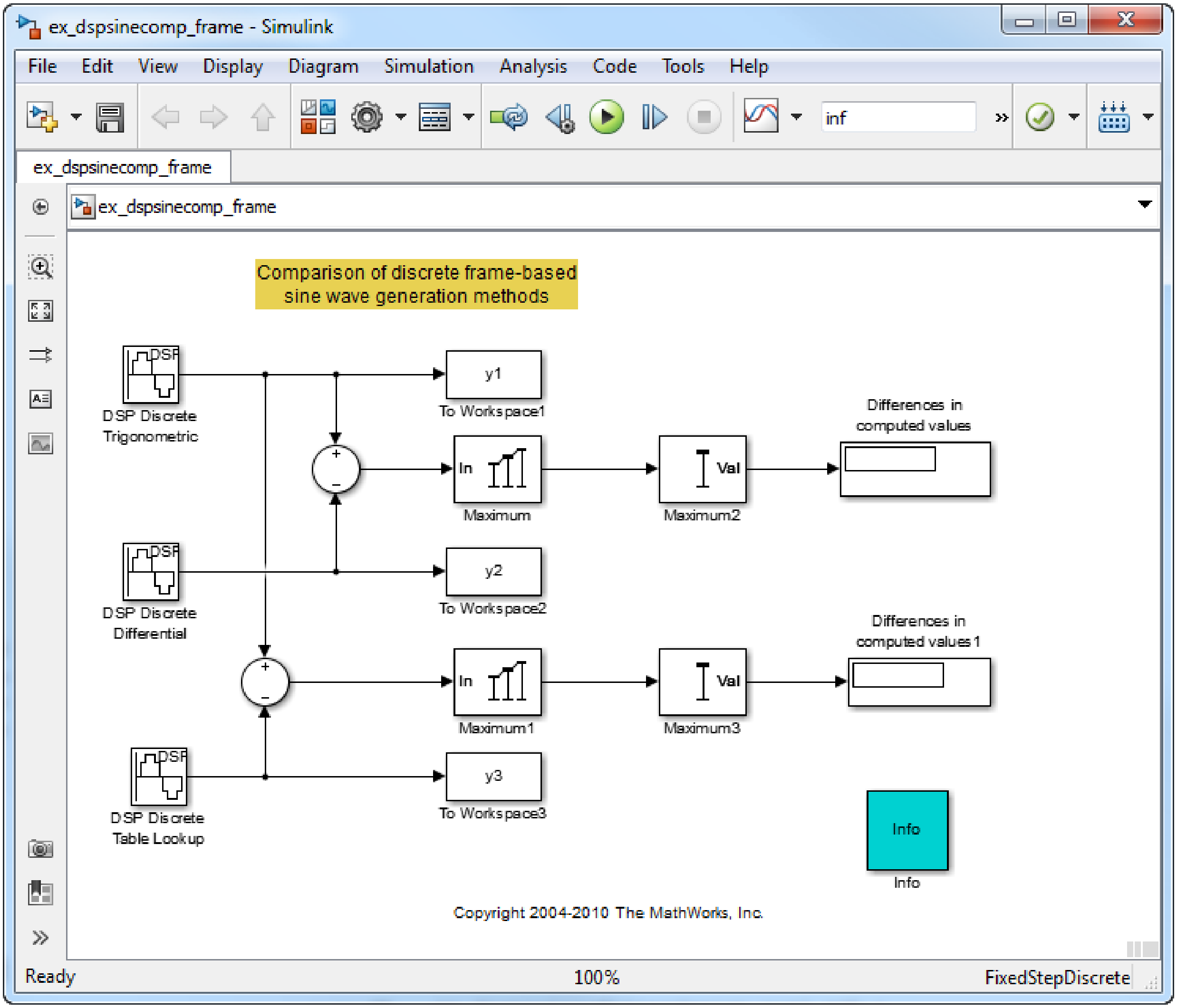
This mode offers reduced computational load, but is subject to drift over time due to cumulative quantization error. Because the method is not contingent on an absolute time value, there is no danger of discontinuity during extended operations (when an absolute time variable might overflow).

Examples

The ex_dspsinecomp and ex_dspsinecomp_frame examples provide a comparison of all the available sine generation methods for sample and frame based modes, respectively.

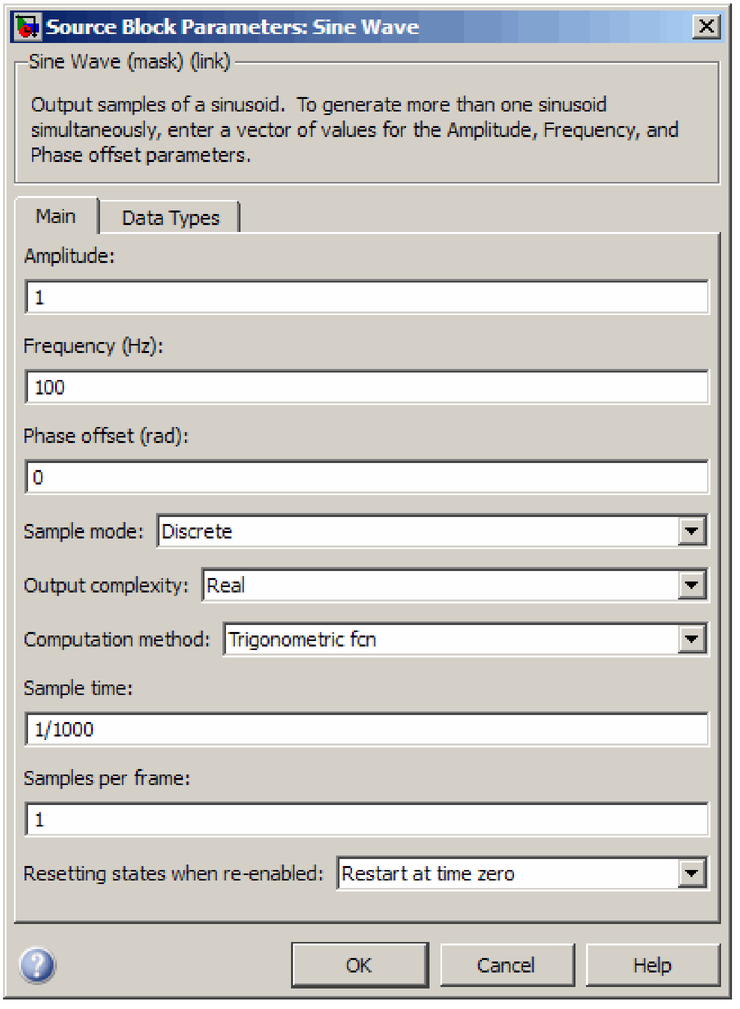


ex_dspsinecomp - Simulink
File
Edit
View
Display
Diagram
Simulation
Analysis
Code
Tools
Help
inf
ex_dspsinecomp
ex_dspsinecomp
12:34
Digital Clock
-K-
Gain
sin
Trigonometric Function
Simulink Sine Wave Continuous
DSP Sine Wave Continuous
DSP Discrete Trigonometric
Simulink Discrete
DSP Discrete Differential
DSP Discrete Table Lookup
Comparison of discrete sine wave generation methods
Differences in computed values
|u|
In
Maximum
Info
Info
Copyright 1998-2010 The MathWorks, Inc.
Ready
100%
FixedStepDiscrete



Dialog Box

The **Main** pane of the Sine Wave block dialog appears as follows.



Amplitude

A length-N vector containing the amplitudes of the sine waves in each of N output channels, or a scalar to be applied to all N channels. The vector length must be the same as that specified for the **Frequency** and **Phase offset** parameters. Tunable (Simulink) when **Computation method** is to `Trigonometric fcn` or `Differential`.

Frequency

A length-*N* vector containing frequencies, in Hertz, of the sine waves in each of *N* output channels, or a scalar to be applied to all *N* channels. The vector length must be the same as that specified for the **Amplitude** and **Phase offset** parameters. You can specify positive, zero, or negative frequencies. Tunable (Simulink) when **Sample mode** is `Continuous` or **Computation method** is `Trigonometric fcn`.

Phase offset

A length-*N* vector containing the phase offsets, in radians, of the sine waves in each of *N* output channels, or a scalar to be applied to all *N* channels. The vector length must be the same as that specified for the **Amplitude** and **Frequency** parameters. Tunable (Simulink) when **Sample mode** is `Continuous` or **Computation method** is `Trigonometric fcn`.

Sample mode

The block's sampling behavior, `Continuous` or `Discrete`. This parameter is not tunable.

Output complexity

The type of waveform to generate: `Real` specifies a real sine wave, `Complex` specifies a complex exponential. This parameter is not tunable.

Computation method

The method by which discrete-time sinusoids are generated: `Trigonometric fcn`, `Table lookup`, or `Differential`. This parameter is not tunable. For more information on each of the available options, see “Discrete Computational Methods” on page 2-1532 in the Description section.

This parameter is only visible when you set the **Sample mode** to `Discrete`.

Note: To generate fixed-point sinusoids, you must set the **Computation method** to `Table lookup`.

Optimize table for

Optimizes the table of sine values for `Speed` or `Memory` (this parameter is only visible when the **Computation method** parameter is set to `Table lookup`). When optimized for speed, the table contains *k* elements, and when optimized for memory, the table contains *k*/4 elements, where *k* is the number of input samples in one full period of the sine wave.

Sample time

The period with which the sine wave is sampled, T_s. The block's output frame period is $M*T_s$, where you specify M in the **Samples per frame** parameter. This parameter is disabled when you select `Continuous` from the **Sample mode** parameter. This parameter is not tunable.

Samples per frame

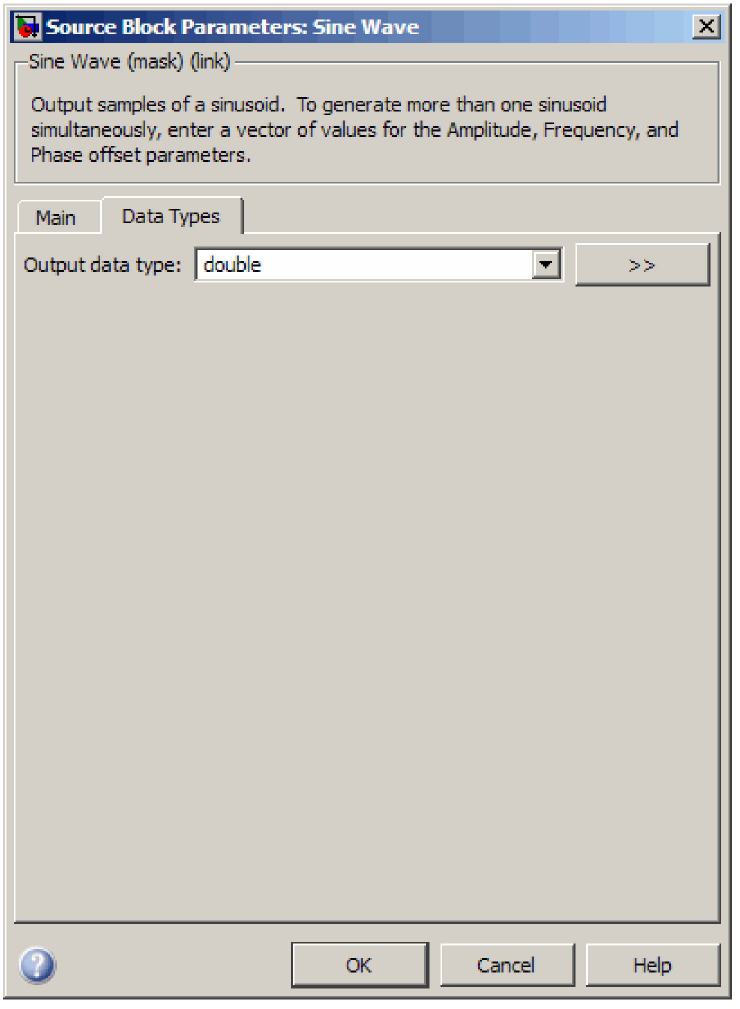
The number of consecutive samples from each sinusoid to buffer into the output frame, M.

This parameter is disabled when you select `Continuous` from the **Sample mode** parameter.

Resetting states when re-enabled

This parameter only applies when the Sine Wave block is located inside an enabled subsystem and the **States when enabling** parameter of the Enable block is set to `reset`. This parameter determines the behavior of the Sine Wave block when the subsystem is re-enabled. The block can either reset itself to its starting state (`Restart at time zero`), or resume generating the sinusoid based on the current simulation time (`Catch up to simulation time`). This parameter is disabled when you select `Continuous` from the **Sample mode** parameter.

The **Data Types** pane of the Sine Wave block dialog appears as follows.



Output data type

Specify the output data type for this block. You can select one of the following:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`. When you select this option, the output data type and scaling matches that of the next downstream block.
- A built in data type, such as `double`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Note: The lookup table for this block is constructed from double-precision floating-point values. Thus, when you use the `Table lookup` computation mode, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length of the **Output** or **User-defined** data type to values greater than 53 bits does not improve the precision of your output.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Sine Wave.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

See Also

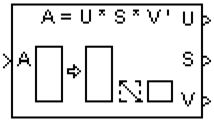
`Chirp`	DSP System Toolbox
`Signal From Workspace`	DSP System Toolbox
Signal Generator	Simulink

Sine Wave	Simulink
sin	MATLAB

Introduced before R2006a

Singular Value Decomposition

Factor matrix using singular value decomposition



Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

`dspfactors`

Description

The Singular Value Decomposition block factors the M-by-N input matrix A such that

$$A = U \cdot diag(S) \cdot V^*$$

where

- U is an M-by-P matrix
- V is an N-by-P matrix
- S is a length-P vector
- P is defined as min(M,N)

When

- $M = N$, U and V are both M-by-M unitary matrices
- $M > N$, V is an N-by-N unitary matrix, and U is an M-by-N matrix whose columns are the first N columns of a unitary matrix
- $N > M$, U is an M-by-M unitary matrix, and V is an N-by-M matrix whose columns are the first M columns of a unitary matrix

In all cases, S is an unoriented vector of positive singular values having length P.

Length-N row inputs are treated as length-N columns.

Note that the first (maximum) element of output S is equal to the 2-norm of the matrix A.

Parameters

Show singular vector ports

Select to enable the `U` and `V` output ports.

Show error status port

Select to enable the E output port, which reports a failure to converge. The possible values you can receive on the port are:

- `0` — The singular value decomposition calculation converges.
- `1` — The singular value decomposition calculation does not converge.

If the singular value decomposition calculation fails to converge, the output at ports U, S, and V are undefined matrices of the correct size.

Simulate using

Type of simulation to run. You can set this parameter to:

- `Interpreted execution` (default)

  Simulate model using the MATLAB interpreter. This option shortens startup time.

- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point
U	• Double-precision floating point • Single-precision floating point
S	• Double-precision floating point • Single-precision floating point
V	• Double-precision floating point • Single-precision floating point
E	• Boolean

See Also

`Autocorrelation LPC`	DSP System Toolbox
`Cholesky Factorization`	DSP System Toolbox
`LDL Factorization`	DSP System Toolbox
`LU Inverse`	DSP System Toolbox
`Pseudoinverse`	DSP System Toolbox
`QR Factorization`	DSP System Toolbox
`SVD Solver`	DSP System Toolbox
`svd`	MATLAB

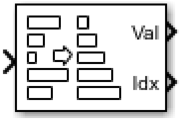
See “Matrix Factorizations” for related information.

Introduced before R2006a

Sort

Sort input elements by value

Library: DSP System Toolbox / Statistics



Description

The Sort block ranks the values of the input elements along each channel (column) in an `Ascending` or a `Descending` order, based on the **Sort order** you specify. Complex inputs are sorted by their magnitude, which is the sum of the squares of the real and imaginary components of the input. You can choose the **Sort algorithm** to be either `Quick sort` or `Insertion sort`. The quick sort algorithm uses a recursive sort method and is faster at sorting more than 32 elements. The insertion sort algorithm uses a nonrecursive method and is faster at sorting fewer than 32 elements. When you generate code, to avoid recursive function calls, use the insertion sort algorithm.

The **Mode** parameter specifies the block's mode of operation, which you can set to `Value`, `Index`, or `Value and Index`.

Ports

Input

`Port_1` — Data input

vector | matrix

The block accepts real-valued or complex-valued multichannel inputs. The input data type must be double precision, single precision, integer, or fixed point, with power-of-two slope and zero bias.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed_point`

Complex Number Support: Yes

Output

`Val` — Sorted data
vector | matrix

The block sorts the data along each channel, and outputs the sorted data through this port. The size, data type, and complexity of the sorted data matches that of the input data. The block sorts complex inputs according to their magnitude.

Dependencies

To enable this port, set the **Mode** parameter to `Value and index` or `Value`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed_point`
Complex Number Support: Yes

`Idx` — Index of the sorted data
vector | matrix

The output at this port contains the indices of the sorted data.

Dependencies

To enable this port, set the **Mode** parameter to `Value and index` or `Index`.

Data Types: `uint32`

Parameters

Main Tab

`Mode` — Mode in which the block operates
`Value and Index` (default) | `Value` | `Index`

When the **Mode** parameter is set to:

- `Value` — The block sorts the elements in each channel of the M-by-N input matrix in an ascending or descending order, based on what you specify in the **Sort order** parameter. The output at each sample time, Val, is an M-by-N matrix that contains the sorted columns of the input.

  The block sorts complex inputs according to their magnitude.
- `Index` — The block sorts the elements in each channel of the M-by-N input matrix, and outputs the index array, I. Each element in I is an integer of type `uint32` that indexes the sorted value in the corresponding column of the input.
- `Value and index` — The block outputs the sorted values of the input data, Val, and the corresponding indices in the index array, I.

`Sort order` — Order to sort the data in
`Ascending` (default) | `Descending`

Specify to sort the input data in either ascending or descending order.

`Sort algorithm` — Sort method
`Quick sort` (default) | `Insertion sort`

The quick sort algorithm uses a recursive sort method and is faster at sorting more than 32 elements. The insertion sort algorithm uses a nonrecursive method and is faster at sorting fewer than 32 elements. When you generate code, to avoid recursive function calls, use the insertion sort algorithm.

Data Types Tab

Note: To use these parameters, the data input must be complex and fixedpoint. For all other inputs, the parameters on the **Data Types** tab are ignored.

`Rounding mode` — Method of rounding operation
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Select the rounding mode for fixed-point operations.

`Overflow mode` — Method of overflow action
`Wrap` (default) | `Saturate`

Select the overflow mode for fixed-point operations.

`Product output` — Product output data type
`Inherit: Same as input` (default) | `fixdt([],16,0)`

The squares of the real and imaginary parts of the complex input are stored in the **Product output** data type.

You can set this parameter to:

- `Inherit: Same as input` — The block specifies the product output data type to be the same as the input data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Product output** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink).

`Accumulator` — Accumulator data type
`Inherit: Same as product output` (default) | `Inherit: Same as input` | `fixdt([],16,0)`

The result of the sum of the squares of the real and imaginary parts of the complex input are stored in the **Accumulator** data type.

You can set this parameter to:

- `Inherit: Same as product output` — The block specifies the accumulator data type to be the same as the product output data type.
- `Inherit: Same as input` — The block specifies the accumulator data type to be the same as the input data type.
- `fixdt([],16,0)` — The block specifies an autosigned, binary-point, scaled, fixed-point data type with a word length of 16 bits and a fraction length of 0.

Alternatively, you can set the **Accumulator** data type by using the **Data Type Assistant**. To use the assistant, click the **Show data type assistant** button >>.

For more information on the data type assistant, see “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink).

Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block.

Algorithms

Sort

The block produces results identical to the MATLAB `sort` function.

The output of the block is equivalent to the following MATLAB code, when **Sort order** is set to:

- `Ascending` — `[Val,I] = sort(u,'ascend')`
- `Descending` — `[Val,I] = sort(u,'descend')`

where:

- `u` is the data input.
- `Val` is the sorted output.
- `I` is the index of the sorted output.

When the input is complex, the block sorts the data according to the magnitude. The block computes the magnitude by taking the sum of the squares of the real and imaginary components of the complex input. This is identical to calling the `sort` function as `[Val,I] = sort(u,...,'ComparisonMethod','abs')`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion

Design and simulate fixed-point algorithms using Fixed-Point Designer™.

The parameters on the **Data Types** tab are used only for complex fixed-point inputs. Complex inputs are sorted by their magnitude, which is the sum of the squares of the real and imaginary components of the input. The results of the squares of the real and imaginary parts are stored in the **Product output** data type. The result of the sum of the squares is stored in the **Accumulator** data type. The parameters on the **Data Types** tab are ignored for all other inputs.

See Also

See Also

Functions

`sort`

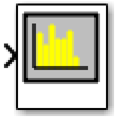
Blocks

`Histogram` | `Median` | `Median Filter`

Introduced before R2006a

Spectrum Analyzer

Display frequency spectrum of time-domain signals



Library

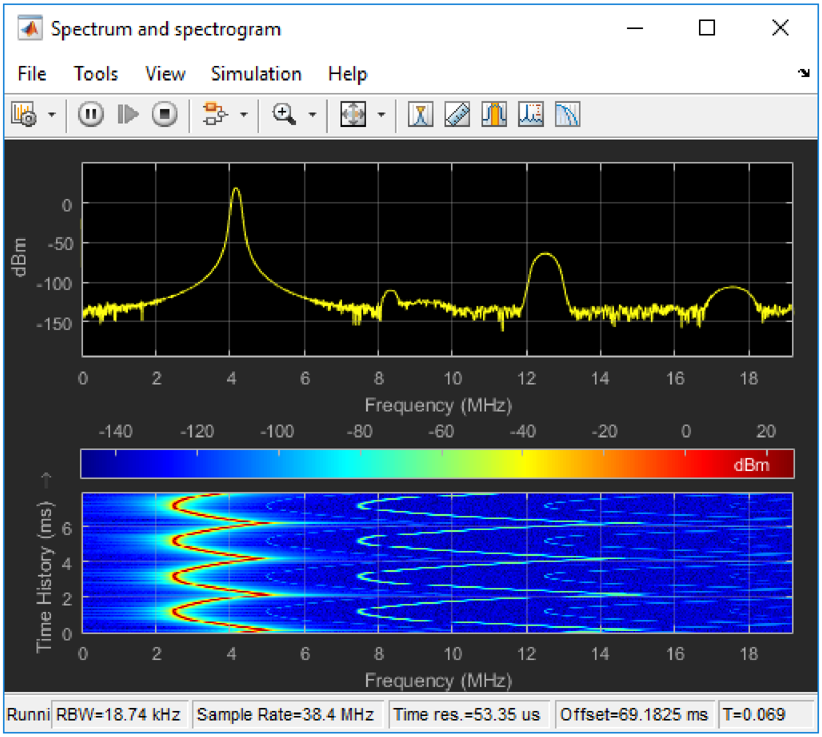
Sinks

`dspsnks4`

Description

The Spectrum Analyzer block, referred to here as the scope, displays frequency spectra of signals. The Spectrum Analyzer block accepts input signals, through one or more input ports, with the following characteristics:

- Discrete sample time
- Real- or complex-valued
- Fixed number of channels of variable length
- Floating- or fixed-point data type



You can use the Spectrum Analyzer block in models running in Normal or Accelerator simulation modes. You can also use the Spectrum Analyzer block in models running in Rapid Accelerator or External simulation modes, with some limitations. See the "Supported Simulation Modes" on page 2-1605 section for more information.

You can use the Spectrum Analyzer block inside of all subsystems and conditional subsystems. *Conditional subsystems* include enabled subsystems, triggered subsystems, enabled and triggered subsystems, and function-call subsystems. See "Conditional Subsystems" (Simulink) for more information.

You can configure and display Spectrum Analyzer settings from the command line with spbscopes.SpectrumAnalyzerConfiguration.

For an example that uses the Spectrum Analyzer, see the "Display Frequency-Domain Data in Spectrum Analyzer".

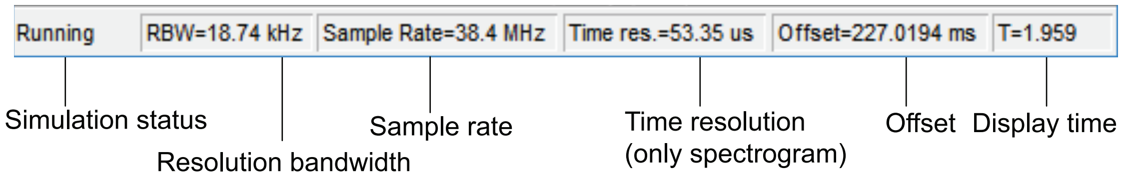
See the following sections for more information on the Spectrum Analyzer:

- “Signal Display” on page 2-1555
- “Spectrum Settings” on page 2-1562
- “Measurements Panels” on page 2-1570
- “Visuals — Spectrum Properties” on page 2-1584
- “Style Dialog Box” on page 2-1587
- “Tools — Axes Scaling Properties” on page 2-1588
- “Algorithms” on page 2-1592
- “Differences from Spectrum Scope Block” on page 2-1600
- “Supported Data Types” on page 2-1605
- “Supported Simulation Modes” on page 2-1605

Note: For information about the Spectrum Analyzer System object, see dsp.SpectrumAnalyzer.

Signal Display

The Spectrum Analyzer indicates the spectrum computation settings that are represented in the current display. Check the **Resolution Bandwidth**, **Time Resolution**, and **Offset** indicators on the status bar in the scope window for this information. These indicators relate to the Minimum Frequency-Axis limit and Maximum Frequency-Axis limit values on the *frequency*-axis of the scope window. The values specified by these indicators may be changed by modifying parameters in the **Spectrum Settings** panel. You can also view the object state and the amount of time data that correspond to the current display. Check the Simulation Status and Simulation time indicators on the status bar in the scope window for this information. The following figure highlights these aspects of the Spectrum Analyzer window.



Note: To prevent the scope from opening when you run your model, right-click the scope icon and select `Comment Out`. If the scope is already open, and you comment it out in the model, the scope displays, "No data can be shown because this scope is commented out." Select `Uncomment` to turn the scope back on.

- *Frequency Span* — The range of values shown on the *frequency*-axis on the Spectrum Analyzer window.

  Details

  Spectrum Analyzer sets the frequency span using the values of parameters on the **Main options** pane of the **Spectrum Settings** panel.

  - `Span (Hz)` and `CF (Hz)` visible — The *Frequency Span* value equals the **Span** parameter in the **Main options** pane.
  - `FStart (Hz)` and `FStop (Hz)` — The frequency span value equals the difference of the **FStop** and **FStart** parameters in the **Main options** pane, as given by the formula: $f_{span} = f_{stop} - f_{start}$.

  By default, the **Full Span** check box in the **Main options** pane is enabled. In this case, the Spectrum Analyzer computes and plots the spectrum over the entire *Nyquist* frequency interval. When the **Two-sided spectrum** check box in the **Trace options** pane is enabled, the Nyquist interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz.

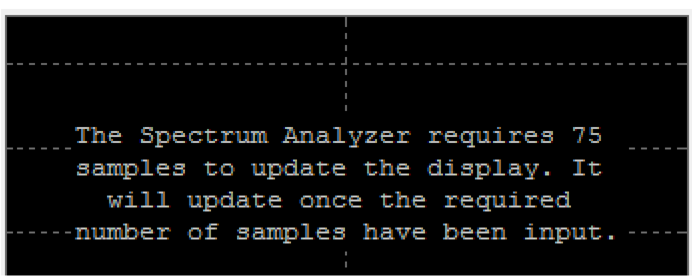
- *Resolution Bandwidth* — The smallest positive frequency or frequency interval that can be resolved.

Details

Spectrum Analyzer sets the resolution bandwidth using the value of the frequency resolution parameter on the **Main options** pane of the **Spectrum Settings** panel. By default, this parameter is set to `RBW (Hz)` and `'Auto'`. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified *Frequency Span*.

You can set the resolution bandwidth to whatever value you choose. For this reason, there is a minimum boundary on the number of input samples required to compute a spectral update. This number of input samples required to compute one spectral update is shown as **Samples/update** in the **Main options** pane. This value is directly related to *RBW* by the following equation:

$N_{samples} = \frac{\left(1 - \frac{O_p}{100}\right) \times NENBW \times F_s}{RBW}$. Overlap percentage, O_p, is the value of the **Overlap %** parameter in the **Window Options** pane of the **Spectrum Settings** panel. *NENBW* is the normalized effective noise bandwidth, a factor of the windowing method used, which is shown in the **Window Options** pane. F_s is the sample rate. In some cases, the number of samples provided in the input are not sufficient to achieve the resolution bandwidth that you specify. When this situation occurs, Spectrum Analyzer produces a message on the display, as shown in the following figure.



Spectrum Analyzer removes this message and displays a spectral estimate as soon as enough data has been input.

If the frequency resolution setting on the **Main options** pane of the **Spectrum Settings** is `Window length`, you specify the window length and the resulting RBW is $\frac{NENBW * Fs}{N_{window}}$. The **Samples/update** in this case is directly related to *RBW* by the following equation: $N_{samples} = \left(1 - \frac{O_p}{100}\right) N_{window}$

- *Time Resolution* — The time resolution for a spectrogram line.

  Details

  Time resolution is the amount of data, in seconds, used to compute a spectrogram line. The minimum attainable resolution is the amount of data time it takes to compute a single spectral estimate. *Time Resolution* is displayed only when the spectrum **Type** is `Spectrogram`.

- *Offset* — The constant frequency offset to apply to the entire spectrum or a vector of frequency offsets to apply to each spectrum for multiple inputs.

  Details

  Spectrum Analyzer adds this constant offset or the vector of offsets to the values on the *frequency*-axis using the value of **Offset** on the **Trace options** pane of the **Spectrum Settings** panel. The offset is the current time value at the middle of the interval of the line displayed at 0 seconds. The actual time of a particular spectrogram line is the offset minus the *y*-axis time listing. You must take this parameter into consideration when you set the **Span (Hz)** and **CF (Hz)** parameters on the **Main options** pane of the **Spectrum Settings** panel to ensure that the frequency span is within Nyquist limits. The offset is displayed on the plot only when the spectrum **Type** is `Spectrogram`.

- *Simulation Status* — Provides the current status of the model simulation.

  Details

  The status can be one of the following conditions:

  - Processing — Occurs after you construct the `SpectrumAnalyzer` object.
  - Stopped — Occurs after you run the release method.

The *Simulation Status* is part of the *Status Bar* in the Spectrum Analyzer window. You can choose to hide or display the entire *Status Bar*. From the Spectrum Analyzer menu, select **View > Status Bar**.

- *Display time* — The amount of time that has progressed since the last update to the Spectrum Analyzer display.

  Details

  Every time data is processed by the block, the simulation time increases by the number of rows in the input signal divided by the sample rate, as given by the following formula: $t_{sim} = t_{sim} + \frac{\text{length(0:length(xsine))-1}}{\text{SampleRate}}$. When **Reduce Plot Rate to Improve Performance** is checked, the simulation time and display time might differ. At the beginning of a simulation, you can modify the **SampleRate** parameter on the **Main options** pane of the **Spectrum Settings** panel.

  The *Display time* indicator is a component of the *Status Bar* in the Spectrum Analyzer window. You can choose to hide or display the entire *Status Bar*. From the Spectrum Analyzer menu, select **View > Status Bar** .

For more information, see "Spectrum Settings" on page 2-1562.

Reduce Plot Rate to Improve Performance

By default, Spectrum Analyzer updates the display at fixed intervals of time at a rate not exceeding 20 hertz. If you want Spectrum Analyzer to plot a spectrum on every simulation time step, you can disable the **Reduce Plot Rate to Improve Performance** option. In the Spectrum Analyzer menu, select **Simulation > Reduce Plot Rate to Improve Performance** to clear the check box. Tunable (Simulink).

Note: When this check box is selected, Spectrum Analyzer may display a misleading spectrum in some situations. For example, if the input signal is wide-band with non-stationary behavior, such as a chirp signal, Spectrum Analyzer might display a stationary spectrum. The reason for this behavior is that Spectrum Analyzer buffers the input signal data and only updates the display periodically at approximately 20 times per second. Therefore, Spectrum Analyzer does not render changes to the spectrum that occur and elapse between updates, which gives the impression of an incorrect spectrum. To ensure that spectral estimates are as accurate as possible, clear the **Reduce Plot**

Rate to Improve Performance check box. When you clear this box, Spectrum Analyzer calculates spectra whenever there is enough data, rendering results correctly.

Spectral Masks

You can add upper and lower spectral mask lines to spectrum plots. You use spectral masks to enhance visualizing spectrum limits. Spectral masks also are useful for comparing spectrum values to specification values.

You must use commands at the MATLAB command line to setup or configure spectral masks for a Spectrum Analyzer block. You use `get_param` to create a Spectrum Analyzer configuration object to control the block. The configuration object has a `SpectralMask` property. The `SpectralMask` property uses `SpectralMaskSpecfication` properties to enable and configure the spectral masks. The `SpectralMaskSpecfication` properties, which are described in spbscopes.SpectrumAnalyzerConfiguration, are

- `EnabledMasks` — Turn on spectral mask.
- `UpperMask` — Set upper mask values.
- `LowerMask` — Set lower mask values.
- `ReferenceLevel` — Select the mask reference level to use a specific value or the spectrum peak value.
- `CustomReferenceLevel` — Set mask reference level value.
- `SelectedChannel` — Set channel to use for mask reference.
- `MaskFrequencyOffset` — Set frequency offset for mask.

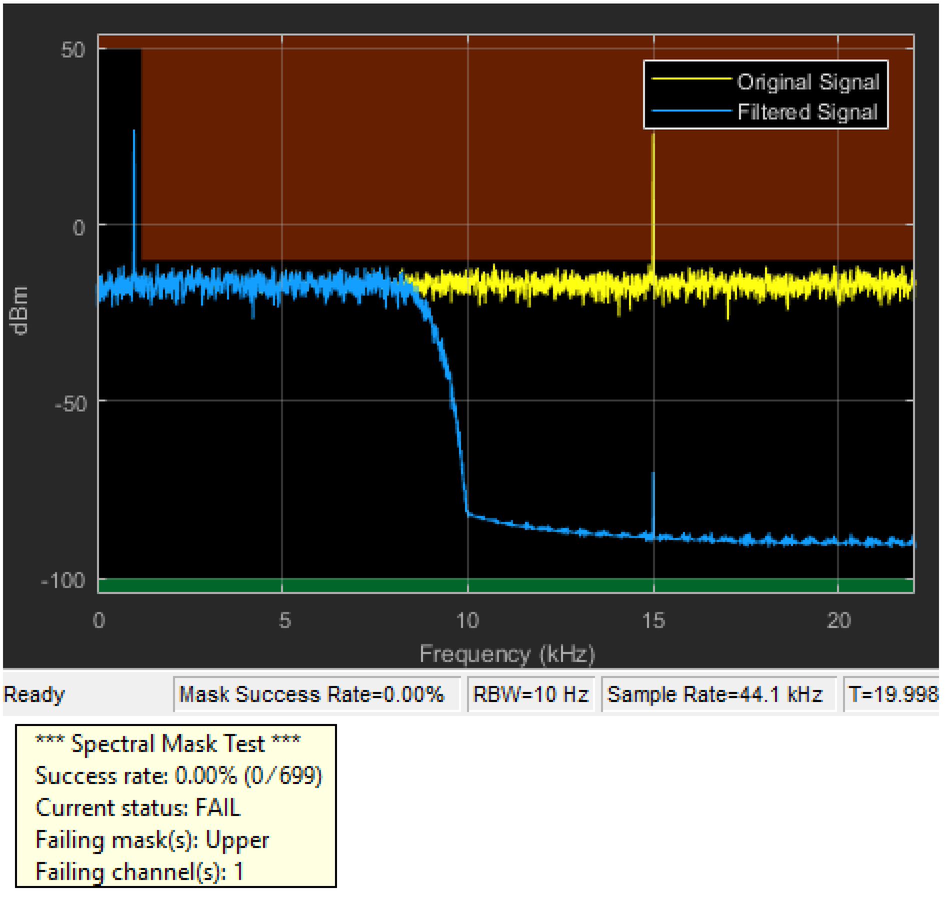
You can check the status of the spectral mask using a method or an event listener.

- To get the status of the spectral masks, call getSpectralMaskStatus.
- To perform an action every time the mask fails, use the `MaskTestFailed` event. To trigger a function when the mask fails, create a listener to the `MaskTestFailed` event and define a callback function to trigger. For more details about using events, see “Events” (MATLAB).

Example of Spectral Mask

This example shows how to create a new model based on the `dsp_basic_filter` template, add a spectral mask to its Spectrum Analyzer block, and run the model.

```
[~,mdl] = fileparts(tempname);
open_system(new_system(mdl,'FromTemplate','dsp_basic_filter'));
saBlock = find_system(mdl,'BlockType','SpectrumAnalyzer');

scopeConfig = get_param(saBlock{1},'ScopeConfiguration');
upperMask = [0 50; 1200 50; 1200 -10; 24000 -10];
scopeConfig.SpectralMask.UpperMask = upperMask;
scopeConfig.SpectralMask.LowerMask = -100;
scopeConfig.SpectralMask.EnabledMasks = 'Upper and lower';

sim(mdl,'StopTime','20');
```

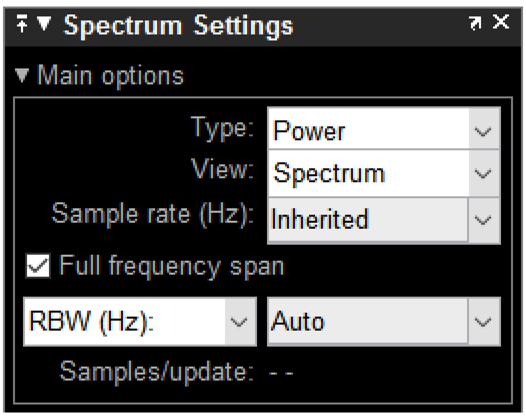


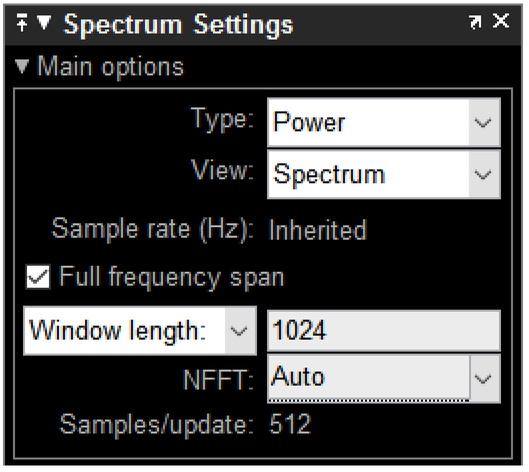
Masks are overlaid on the spectrum. If the mask is green, the signal is passing. If the mask is red, the signal is failing.

At the bottom of the window, the **Mask success rate** shows what percentage of the time the mask is succeeding. A tooltip gives you details about which mask is failing, how many times the mask(s) failed, and which channels are causing the failure.

Spectrum Settings

The **Spectrum Settings** panel appears at the right side of the Spectrum Analyzer figure. This panel enables you to modify settings to control the manner in which the spectrum is calculated. You can choose to hide or display the **Spectrum Settings** panel. In the Spectrum Analyzer menu, select **View** > **Spectrum Settings**. Alternatively, in the Spectrum Analyzer toolbar, select the Spectrum Settings  button.





The **Spectrum Settings** panel is separated into three panes, labeled **Main Options**, **Window Options**, and **Trace Options**. You can expand each pane to see the available options.

Main Options Pane

The **Main Options** pane enables you to modify the main options.

- **Type** — The type of spectrum to display. Available options are `Power`, `Power density`, and `RMS`. When you set this parameter to `Power`, the Spectrum Analyzer shows the power spectrum. When you set this parameter to `Power density`, the Spectrum Analyzer shows the power spectral density. The power spectral density is the magnitude of the spectrum normalized to a bandwidth of 1 hertz. When you set this parameter to `RMS`, the Spectrum Analyzer shows the root mean squared spectrum. Tunable (Simulink)
- **View** — The spectrum view to display. Available options are `Spectrum`, `Spectrogram`, and `Spectrum and spectrogram`. When you set this parameter to `Spectrum`, the Spectrum Analyzer shows the spectrum. When you set this parameter to `Spectrogram`, the Spectrum Analyzer shows the spectrogram, which displays frequency content over time. The most recent spectrogram update is at the bottom of the display and time scrolls from the bottom to the top of the display. When you set this parameter to `Spectrum and spectrogram`, the Spectrum Analyzer shows both the spectrum and spectrogram. Tunable (Simulink).

- **Sample rate (Hz)** — The sample rate, in hertz, of the input signals. Select `Inherited` to use the same sample rate as the input signal. To specify a sample rate, enter its value.
- **Full frequency span** — Enable this check box to have Spectrum Analyzer compute and plot the spectrum over the entire *Nyquist* frequency interval. By default, when the **Two-sided spectrum** check box is also enabled, the Nyquist interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. If you clear the **Two-sided spectrum** check box, the Nyquist interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. Tunable (Simulink).
- **Span (Hz)** and **CF (Hz)**, or **FStart (Hz)** and **FStop (Hz)** — When **Span (Hz)** is showing in the **Main Options** pane, you define the range of values shown on the *frequency*-axis on the Spectrum Analyzer window using frequency span and center frequency. From the drop-down list, select **FStart (Hz)** to define the range of *frequency*-axis values using start frequency and stop frequency instead.
  - **Span (Hz)** — The frequency span, in hertz. This parameter defines the range of values shown on the *frequency*-axis on the Spectrum Analyzer window. Tunable (Simulink).
  - **CF (Hz)** — The center frequency, in hertz. This parameter defines the value shown at the middle point of the *frequency*-axis on the Spectrum Analyzer window. Tunable (Simulink).
  - **FStart (Hz)** — The start frequency, in hertz. This parameter defines the value shown at the leftmost side of the *frequency*-axis on the Spectrum Analyzer window. Tunable (Simulink).
  - **FStop (Hz)** — The stop frequency, in hertz. The parameter defines the value shown at the rightmost side of the *frequency*-axis on the Spectrum Analyzer window. Tunable (Simulink).
- **RBW (Hz) / Window length** — The frequency resolution method.

  If set to **RBW (Hz)**, the resolution bandwidth, in hertz. This property defines the smallest positive frequency that can be resolved. By default, this property is set to `Auto`. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 *RBW* intervals over the specified frequency span.

If you set this property to a numeric value, then you must specify a value that ensures there are at least two *RBW* intervals over the specified frequency span. In other words, the ratio of the overall frequency span to *RBW* must be at least two: $\frac{span}{RBW} > 2$.

Tunable (Simulink).

If set to **Window length**, the length of the window, in samples, used to control the frequency resolution and compute the spectral estimates. The window length must be an integer scalar greater than 2.

Tunable (Simulink).

The time resolution value is determined based on frequency resolution method, the RBW setting, and the time resolution setting.

Frequency Resolution	RBW Setting	Time Resolution Setting	Time Resolution
RBW (Hz)	`Auto`	`Auto`	1/RBW s
RBW (Hz)	`Auto`	Manually entered	Time Resolution s
RBW (Hz)	Manually entered	`Auto`	1/RBW s
RBW (Hz)	Manually entered	Manually entered	Must be equal to or greater than the minimum attainable time resolution, 1/RBW s. Several spectral estimates are combined into one spectrogram line to obtain the desired time resolution. Interpolation is used to obtain time resolution values that are not integer multiples of 1/RBW s.

Frequency Resolution	RBW Setting	Time Resolution Setting	Time Resolution
Window length	—	`Auto`	1/RBW s RBW = *(NENBW**Fs)/Window Length, where *NENBW* is the normalized effective noise bandwidth of the specified window.
Window length	—	Manually entered	Must be equal to or greater than the minimum attainable time resolution, *(NENBW**Fs)/Window Length. Several spectral estimates are combined into one spectrogram line to obtain the desired time resolution. Interpolation is used to obtain time resolution values that are not integer multiples of 1/RBW s.

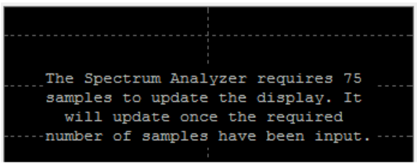
- **NFFT** — The number of Fast Fourier Transform (FFT) points. You can set the **NFFT** only when in Window length mode. This property defines the length of the FFT that Spectrum Analyzer uses to compute spectral estimates. Acceptable options are `Auto` or a positive, scalar integer. The **NFFT** value must be greater than or equal to the **Window length**. By default, when **NFFT** is set to `Auto`, Spectrum Analyzer sets the number of FFT points to the window length. When in RBW mode, an FFT length is used that equals the window length required to achieve the specified RBW value.

  When this property is set to a positive integer, this property is equivalent to the `n` parameter that you can set when you run the MATLAB `fft` function. Tunable (Simulink).

- **Samples/update** — The number of input samples required to compute one spectral update. You cannot modify this property; it is shown here for display purposes only. This property is directly related to *RBW* by the following equation:

  $N_{samples} = \dfrac{\left(1 - \dfrac{O_p}{100}\right) \times NENBW \times F_s}{RBW}$ or to the window length by this equation:

$N_{samples} = \left(1-\frac{O_p}{100}\right) x\ WindowLength$. $NENBW$ is the normalized effective noise bandwidth, a factor of the windowing method used, which is shown in the **Window Options** pane. F_s is the sample rate. If the number of samples provided in the input are not sufficient to achieve the resolution bandwidth that you specify, Spectrum Analyzer produces a message on the display as shown in the following figure.

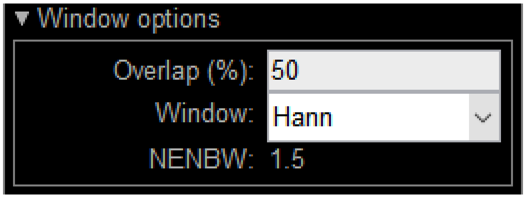


Spectrogram Options Pane

- **Channel** — Select the signal channel for which the spectrogram settings apply. This option displays only when the **Type** is `Spectrogram` and only if there is more than one signal channel input.
- **Time res. (s)** — The time resolution, in seconds. Time resolution is the amount of data, in seconds, used to compute a spectrogram line. The minimum attainable resolution is the amount of data time it takes to compute a single spectral estimate. The tooltip displays the minimum attainable resolution given the current settings. This property applies only to spectrograms. Tunable (Simulink)
- **Time span (s)** — The time span over which the Spectrum Analyzer displays the spectrogram, in seconds. The time span is the product of the desired number of spectral lines and the time resolution. The tooltip displays the minimum allowable time span, given the current settings. If the time span is set to `Auto`, 100 spectral lines are used. This property applies only to spectrograms.Tunable (Simulink)

Window Options Pane

The **Window Options** pane enables you to modify the window options.



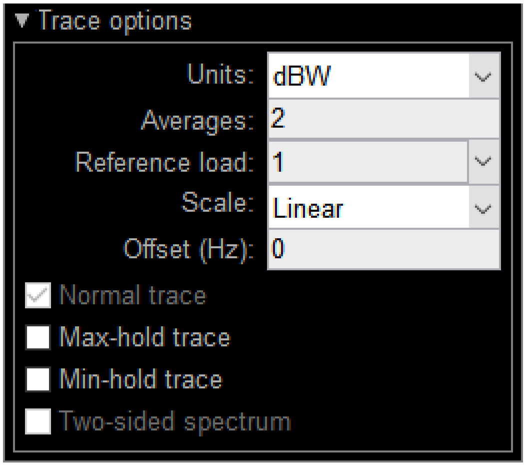
- **Overlap (%)** — The segment overlap percentage. This parameter defines the amount of overlap between the previous and current buffered data segments. The overlap creates a window segment that is used to compute a spectral estimate. The value must be greater than or equal to zero and less than 100. Tunable (Simulink).
- **Window** — The windowing method to apply to the spectrum. Windowing is used to control the effect of sidelobes in spectral estimation. The window you specify affects the window length required to achieve a resolution bandwidth and the required number of samples per update. For more information about windowing, see "Windows" (Signal Processing Toolbox). Tunable (Simulink).
- **Attenuation (dB)** — The sidelobe attenuation, in decibels (dB). This property applies only when you set the **Window** parameter to `Chebyshev` or `Kaiser`. You must specify a value greater than or equal to `45`. Tunable (Simulink).
- **NENBW** — Normalized Effective Noise Bandwidth of the window. You cannot modify this parameter; it is a readout shown here for display purposes only. This parameter is a measure of the noise performance of the window. It is the width of a rectangular filter that accumulates the same noise power with the same peak power gain. NENBW can be calculated from the windowing function using the following equation:

$$NENBW = N_{window} \frac{\sum_{n=1}^{N_{window}} w^2(n)}{\left[\sum_{n=1}^{N_{window}} w(n)\right]^2}$$

The rectangular window has the smallest NENBW, with a value of 1. All other windows have a larger NENBW value. For example, the Hann window has an NENBW value of approximately 1.5.

Trace Options Pane

The **Trace Options** pane enables you to modify the trace options.



- **Units** — The units of the spectrum. The available values depends on the value of **Type**. Available options include `dBm`, `dBW`, `Watts`, `Vrms`, and `dBV`. Tunable (Simulink).
- **Averages** — Specify as a positive, scalar integer the number of spectral averages. This property applies only when the Spectrum **View** is `Spectrum` or `Spectrum and spectrogram`. Spectrum Analyzer computes the current power spectrum estimate by computing a running average of the last *N* power spectrum estimates. This property defines the number of spectral averages, *N*. Tunable (Simulink).
- **Reference load** — The reference load, in ohms, used to scale the spectrum. Specify as a real, positive scalar the load, in ohms, that the Spectrum Analyzer uses as a reference to compute power values. Tunable (Simulink).
- **Scale** — Linear or logarithmic scale. When the frequency span contains negative frequency values, Spectrum Analyzer disables the logarithmic option. Tunable (Simulink).
- **Offset** — The constant frequency offset to apply to the entire spectrum or a vector of frequencies to apply to each spectrum for multiple inputs. The offset parameter is added to the values on the *frequency*-axis in the Spectrum Analyzer window. It is not used in any spectral computations. You must take the parameter into consideration when you set the **Span (Hz)** and **CF (Hz)** parameters to ensure that the frequency span is within Nyquist limits. The Nyquist interval is

$\left[-\frac{SampleRate}{2},\frac{SampleRate}{2}\right]+FrequencyOffset$ hertz if **Two-sided spectrum** is selected, and $\left[0,\frac{SampleRate}{2}\right]+FrequencyOffset$ hertz otherwise. Tunable (Simulink)

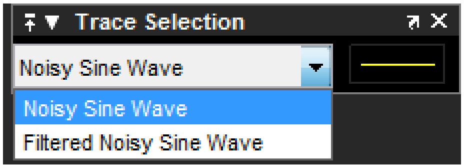
- **Normal trace** — Normal trace view. This property applies only when the Spectrum **View** is `Spectrum` or `Spectrum and spectrogram`. By default, when this check box is enabled, Spectrum Analyzer calculates and plots the power spectrum or power spectrum density. Spectrum Analyzer performs a smoothing operation by averaging a number of spectral estimates. To clear this check box, you must first select either the **Max hold trace** or the **Min hold trace** check box. Tunable (Simulink).
- **Max hold trace** — Maximum hold trace view. This property applies only when the Spectrum **View** is `Spectrum` or `Spectrum and spectrogram`. Select this check box to enable Spectrum Analyzer to plot the maximum spectral values of all the estimates obtained. Tunable (Simulink).
- **Min hold trace** — Minimum hold trace view. This property applies only when the Spectrum **View** is `Spectrum` or `Spectrum and spectrogram`. Select this check box to enable Spectrum Analyzer to plot the minimum spectral values of all the estimates obtained. Tunable (Simulink)
- **Two-sided spectrum** — Select this check box to enable two-sided spectrum view. In this view, both negative and positive frequencies are shown. If you clear this check box, Spectrum Analyzer shows a one-sided spectrum with only positive frequencies. Spectrum Analyzer requires that this parameter is selected when the input signal is complex-valued.

Measurements Panels

The Measurements panels are the other four panels that appear to the right side of the Spectrum Analyzer figure.

Trace Selection Panel

When you use the scope to view multiple signals, the Trace Selection panel appears if you have more than one signal displayed and you click any of the other Measurements panels. The Measurements panels display information about only the signal chosen in this panel. Choose the signal name for which you would like to display time domain measurements. See the following figure.



You can choose to hide or display the **Trace Selection** panel. In the Scope menu, select **Tools** > **Measurements** > **Trace Selection**.

Cursor Measurements Panel

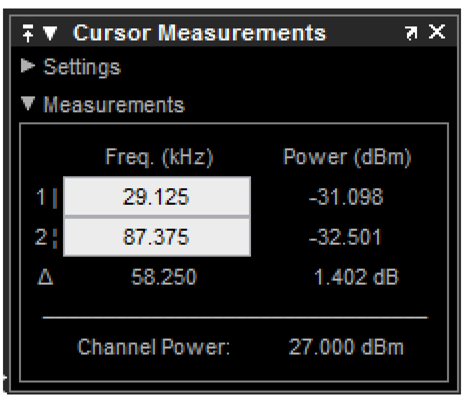
The **Cursor Measurements** panel displays screen cursors. The panel provides two types of cursors for measuring signals. Waveform cursors are vertical cursors that track along the signal. Screen cursors are both horizontal and vertical cursors that you can place anywhere in the display.

Note: If a data point in your signal has more than one value, the cursor measurement at that point is undefined and no cursor value is displayed.

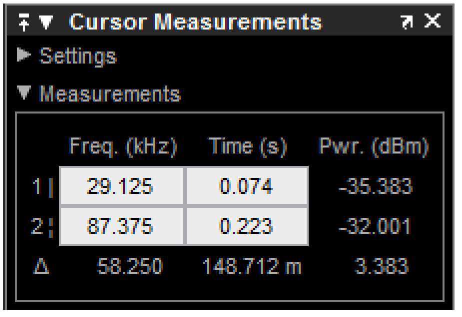
In the Scope menu, select **Tools** > **Measurements** > **Cursor Measurements**.

Alternatively, in the Scope toolbar, click the Cursor Measurements  button.

The **Cursor Measurements** panel appears as follows for the spectrum and dual view.



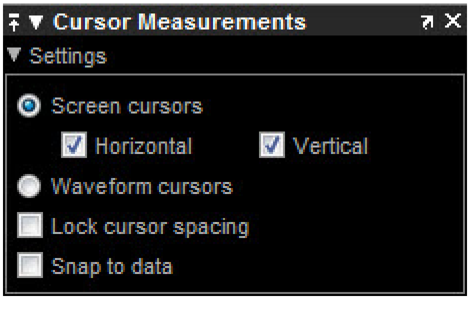
The **Cursor Measurements** panel appears as follows for the spectrogram view. You must pause the spectrogram display before you can use cursors.



The **Cursor Measurements** panel is separated into two panes, labeled **Settings** and **Measurements**. You can expand each pane to see the available options.

You can use the mouse or the left and right arrow keys to move vertical or waveform cursors and the up and down arrow keys for horizontal cursors.

The **Settings** pane enables you to modify the type of screen cursors used for calculating measurements. When more than one signal is displayed, you can assign cursors to each trace individually.



- **Screen Cursors** — Shows screen cursors (for spectrum and dual view only).
- **Horizontal** — Shows horizontal screen cursors (for spectrum and dual view only).
- **Vertical** — Shows vertical screen cursors (for spectrum and dual view only).
- **Waveform Cursors** — Shows cursors that attach to the input signals (for spectrum and dual view only).
- **Lock Cursor Spacing** — Locks the frequency difference between the two cursors.
- **Snap to Data** — Positions the cursors on signal data points.

Measurements Pane

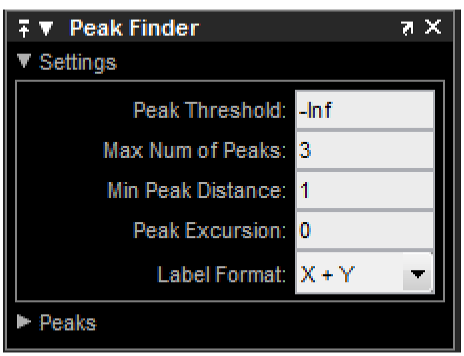
The **Measurements** pane displays the frequency (Hz) , time (s), and power (dBm) value measurements. Time is displayed only in spectrogram mode. **Channel Power** shows the total power between the cursors.

- **1** — Shows or enables you to modify the frequency, time (for spectrograms only), or both, at cursor number one.
- **2** — Shows or enables you to modify the frequency, time (for spectrograms only), or both, at cursor number two.
- **Δ** — Shows the absolute value of the difference in the frequency, time (for spectrograms only), or both, and power between cursor number one and cursor number two.
- **Channel Power** — Shows the total power in the channel defined by the cursors.

The letter after the value associated with a measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix.

Peak Finder Panel

The **Peak Finder** panel displays the maxima, showing the *x*-axis values at which they occur. Peaks are defined as a local maximum where lower values are present on both sides of a peak. Endpoints are not considered to be peaks. This panel allows you to modify the settings for peak threshold, maximum number of peaks, and peak excursion. You can choose to hide or display the **Peak Finder** panel. In the scope menu, select **Tools** > **Measurements** > **Peak Finder**. Alternatively, in the scope toolbar, select the Peak Finder  button.



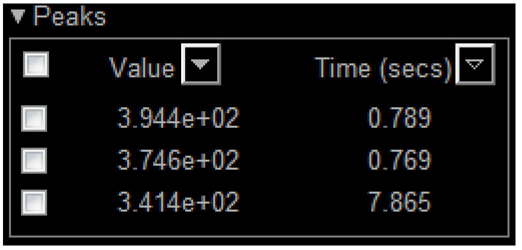
The **Peak finder** panel is separated into two panes, labeled **Settings** and **Peaks**. You can expand each pane to see the available options.

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input signal. For more information on the algorithms this pane uses, see the Signal Processing Toolbox `findpeaks` function reference.

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the `MINPEAKHEIGHT` parameter, which you can set when you run the `findpeaks` function.
- **Max Num of Peaks** — The maximum number of peaks to show. The value you enter must be a scalar integer from 1 through 99. This setting is equivalent to the `NPEAKS` parameter, which you can set when you run the `findpeaks` function.
- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the `MINPEAKDISTANCE` parameter, which you can set when you run the `findpeaks` function.
- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples The peak excursion setting is equivalent to the `THRESHOLD` parameter, which you can set when you run the `findpeaks` function.
- **Label Format** — The coordinates to display next to the calculated peak values on the plot. To see peak values, expand the **Peaks** pane and select the check boxes associated with individual peaks of interest. By default, both *x*-axis and *y*-axis values are displayed on the plot. Select which axes values you want to display next to each peak symbol on the display.

  - `X+Y` — Display both *x*-axis and *y*-axis values.
  - `X` — Display only *x*-axis values.
  - `Y` — Display only *y*-axis values.

The **Peaks** pane displays all of the largest calculated peak values. It also shows the coordinates at which the peaks occur, using the parameters you define in the **Settings** pane. You set the **Max Num of Peaks** parameter to specify the number of peaks shown in the list.



The numerical values displayed in the **Value** column are equivalent to the `pks` output argument returned when you run the `findpeaks` function. The numerical values displayed in the second column are similar to the `locs` output argument returned when you run the `findpeaks` function.

The Peak Finder displays the peak values in the **Peaks** pane. By default, the **Peak Finder** panel displays the largest calculated peak values in the **Peaks** pane in decreasing order of peak height. Use the sort descending button ( ) to rearrange the category and order by which Peak Finder displays peak values. Click this button again to sort the peaks in ascending order instead. When you do so, the arrow changes direction to become the sort ascending button ( ). A filled sort button indicates that the peak values are currently sorted in the direction of the button arrow. If the sort button is not filled ( ), then the peak values are sorted in the opposite direction of the button arrow. The **Max Num of Peaks** parameter still controls the number of peaks listed.

Use the check boxes to control which peak values are shown on the display. By default, all check boxes are cleared and the **Peak Finder** panel hides all the peak values. To show all the peak values on the display, select the check box in the top-left corner of the **Peaks** pane. To hide all the peak values on the display, clear this check box. To show an individual peak, select the check box directly to the left of its **Value** listing. To hide an individual peak, clear the check box directly to the left of its **Value** listing.

The Peaks are valid for any units of the input signal. The letter after the value associated with each measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Channel Measurements Panel

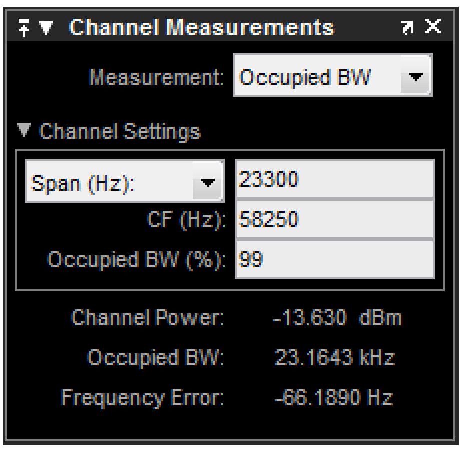
The **Channel Measurements** panel displays occupied bandwidth or adjacent channel power ratio (ACPR) measurements. You can choose to hide or display this pane in the Scope menu by selecting **Tools** > **Measurements** > **Channel Measurements**.

Alternatively, in the Scope toolbar, click the Cursor Measurements  button.

In addition to the measurements, the **Channel Measurements** panel has an expandable **Channel Settings** pane.

- **Measurement** — The type of measurement data to display. Available options are `Occupied BW` or `ACPR`. See "Algorithms" on page 2-1592 for information on how Occupied BW is calculated. ACPR is the adjacent channel power ratio, which is the ratio of the main channel power to the adjacent channel power.

When you select `Occupied BW` as the **Measurement**, the following fields appear.



- **Channel Settings** — Enables you to modify the parameters for calculating the channel measurements.

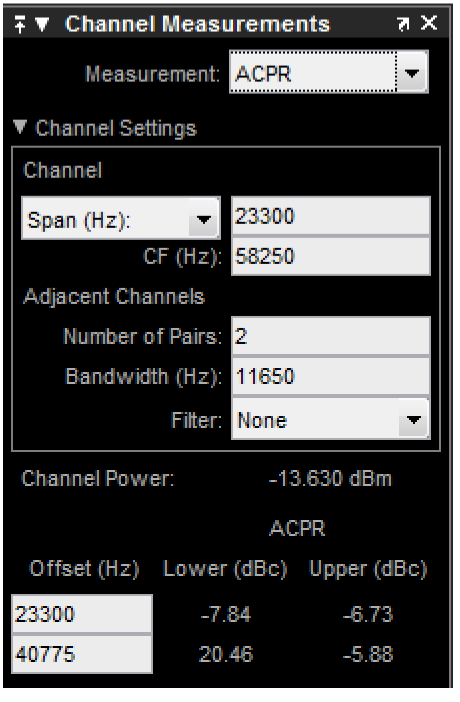
  #### Channel Settings for Occupied BW

  - Select the frequency span of the channel, `Span(Hz)`, and specify the center frequency **CF (Hz)** of the channel. Alternatively, select the starting frequency, FStart(Hz), and specify the starting frequency and ending frequency (**FStop (Hz)**) values of the channel.

- **CF (Hz)** — The center frequency of the channel.
- **Occupied BW (%)** — The percentage of the total integrated power of the spectrum centered on the selected channel frequency over which to compute the occupied bandwidth.

- **Channel Power** — The total power in the channel.
- **Occupied BW** — The bandwidth containing the specified **Occupied BW (%)** of the total power of the spectrum. This setting is available only if you select Occupied BW as the **Measurement** type.
- **Frequency Error** — The difference between the center of the occupied band and the center frequency (**CF**) of the channel. This setting is available only if you select Occupied BW as the **Measurement** type.

When you select ACPR as the **Measurement**, the following fields appear.



- **Channel Settings** — Enables you to modify the parameters for calculating the channel measurements.

  Channel Settings for ACPR

  - Select the frequency span of the channel, `Span (Hz)`, and specify the center frequency **CF (Hz)** of the channel. Alternatively, select the starting frequency, `FStart(Hz)`, and specify the starting frequency and ending frequency (**FStop (Hz)**) values of the channel.
  - **CF (Hz)** — The center frequency of the channel.
  - **Number of Pairs** — The number of pairs of adjacent channels.
  - **Bandwidth (Hz)** — The bandwidth of the adjacent channels.
  - **Filter** — The filter to use for both main and adjacent channels. Available filters are `None`, `Gaussian`, and `RRC` (root-raised cosine).
- **Channel Power** — The total power in the channel.
- **Offset (Hz)** — The center frequency of the adjacent channel with respect to the center frequency of the main channel. This setting is available only if you select `ACPR` as the **Measurement** type.
- **Lower (dBc)** — The power ratio of the lower sideband to the main channel. This setting is available only if you select `ACPR` as the **Measurement** type.
- **Upper (dBc)** — The power ratio of the upper sideband to the main channel. This setting is available only if you select `ACPR` as the **Measurement** type.

Distortion Measurements Panel

The **Distortion Measurements** panel displays harmonic distortion and intermodulation distortion measurements. You can choose to hide or display this panel in the Scope menu by selecting **Tools** > **Measurements** > **Distortion Measurements**. Alternatively, in the Scope toolbar, click the Distortion Measurements  button.

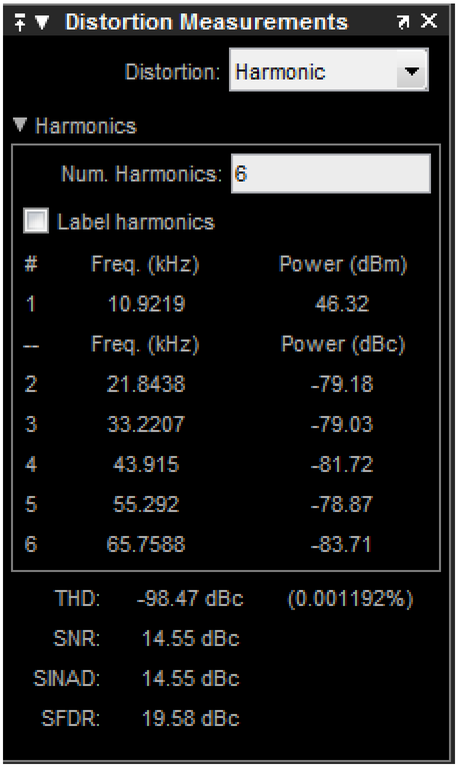
The **Distortion Measurements** panel has an expandable **Harmonics** pane, which shows measurement results for the specified number of harmonics.

Note: For an accurate measurement, ensure that the fundamental signal (for harmonics) or primary tones (for intermodulation) is larger than any spurious or harmonic content. To do so, you may need to adjust the resolution bandwidth (RBW) of the spectrum analyzer. Make sure that the bandwidth is low enough to isolate the signal and harmonics from spurious and noise content. In general, you should set the RBW so that there is at least a 10dB separation between the peaks of the sinusoids and the noise floor. You may also need to select a different spectral window to obtain a valid measurement.

- **Distortion** — The type of distortion measurements to display. Available options are `Harmonic` or `Intermodulation`. Select `Harmonic` if your system input is a single sinusoid. Select `Intermodulation` if your system input is two equal amplitude sinusoids. Intermodulation can help you determine distortion when only a small portion of the available bandwidth will be used.

  See “Algorithms” on page 2-1592 for information on how distortion measurements are calculated.

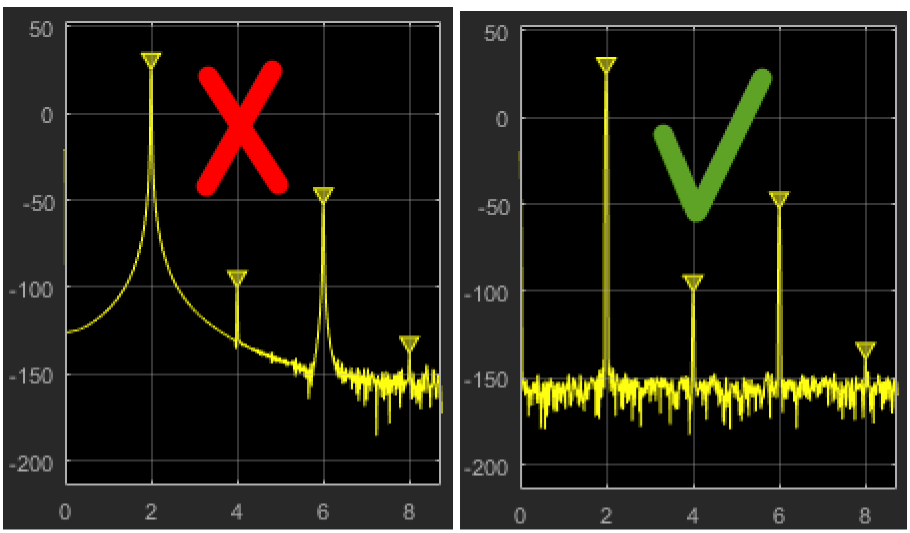
When you select `Harmonic` as the **Distortion**, the following fields appear.



The harmonic distortion measurement automatically locates the largest sinusoidal component (fundamental signal frequency). It then computes the harmonic frequencies and power in each harmonic in your signal. Any DC component is ignored. Any harmonics that are outside the spectrum analyzer's frequency span are not included in the measurements. Adjust your frequency span so that it includes all the desired harmonics.

Note: To best view the harmonics, make sure that your fundamental frequency is set high enough to resolve the harmonics. However, this frequency should not be so high that aliasing occurs. For the best display of harmonic distortion, your plot should not show skirts, which indicate frequency leakage. Additionally, the noise floor should be visible.

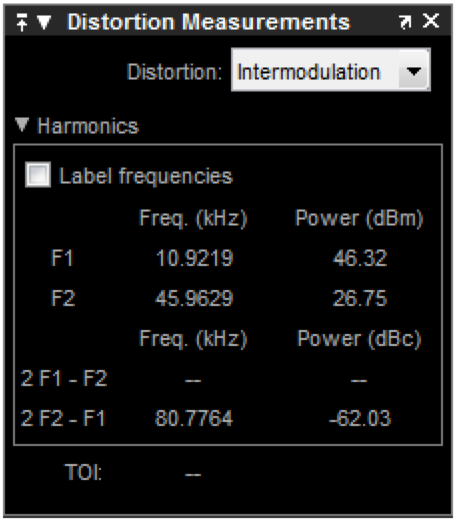
For a better display, try a Kaiser window with a large sidelobe attenuation (e.g. between 100–300 db).



- **Num. Harmonics** — Number of harmonics to display, including the fundamental frequency. Valid values of **Num. Harmonics** are from 2 to 99. The default value is 6.
- **Label Harmonics** — Select **Label Harmonics** to add numerical labels to each harmonic in the spectrum display.
- **1** — The fundamental frequency, in hertz, and its power, in decibels of the measured power referenced to one milliwatt (dBm).

- **2, 3, ...** — The harmonics frequencies, in hertz, and their power in decibels relative to the carrier (dBc). If the harmonics are at the same level or exceed the fundamental frequency, reduce the input power.
- **THD** — The total harmonic distortion. This value represents the ratio of the power in the harmonics, *D*, to the power in the fundamental frequency, *S*. If the noise power is too high in relation to the harmonics, the THD value is not accurate. In this case, lower the resolution bandwidth or select a different spectral window. $THD = 10\log_{10}(D/S)$.
- **SNR** — Signal-to-noise ratio (SNR). This value represents the ratio of power in the fundamental frequency, *S*, to the power of all nonharmonic content, *N*, including spurious signals, in decibels relative to the carrier (dBc). $SNR = 10\log_{10}(S/N)$. If you see – – as the reported SNR, your signal's total non-harmonic content is less than 30% of the total signal.
- **SINAD** — Signal-to-noise-and-distortion. This value represents the ratio of the power in the fundamental frequency, *S* to all other content (including noise, *N*, and harmonic distortion, *D*), in decibels relative to the carrier (dBc). $SINAD = 10\log_{10}(S/(N+D)$.
- **SFDR** — Spurious free dynamic range (SFDR). This value represents the ratio of the power in the fundamental frequency, *S*, to power of the largest spurious signal, *R*, regardless of where it falls in the frequency spectrum. The worst spurious signal may or may not be a harmonic of the original signal. SFDR represents the smallest value of a signal that can be distinguished from a large interfering signal. SFDR includes harmonics. $SFDR = 10\log_{10}(S/R)$.

When you select `Intermodulation` as the **Distortion**, the following fields appear.



The intermodulation distortion measurement automatically locates the fundamental, first-order frequencies (F1 and F2). It then computes the frequencies of the third-order intermodulation products (2*F1-F2 and 2*F2-F1).

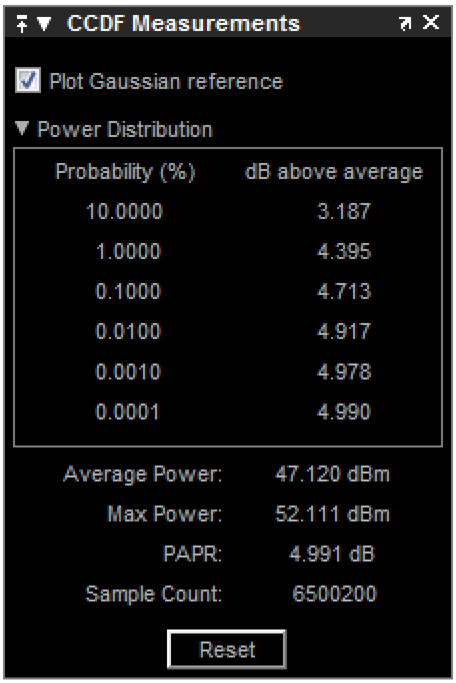
- **Label frequencies** — Select **Label frequencies** to add numerical labels to the first-order intermodulation product and third-order frequencies in the spectrum analyzer display.
- **F1** — Lower fundamental first-order frequency
- **F2** — Upper fundamental first-order frequency
- **2F1 - F2** — Lower intermodulation product from third-order harmonics
- **2F2 - F1** — Upper intermodulation product from third-order harmonics
- **TOI** — Third-order intercept point. If the noise power is too high in relation to the harmonics, the TOI value will not be accurate. In this case, you should lower the resolution bandwidth or select a different spectral window. If the TOI has the same amplitude as the input two-tone signal, reduce the power of that input signal.

CCDF Measurements Panel

The **CCDF Measurements** panel displays complimentary cumulative distribution function measurements. CCDF measurements in this scope show the probability of a signal's instantaneous power being a specified level above the signal's average power. These measurements are useful indicators of a signal's dynamic range.

To compute the CCDF measurements, each input sample is quantized to 0.01 dB increments. Using a histogram 100 dB wide (10,000 points at 0.01 dB increments), the largest peak encountered is placed in the last bin of the histogram. If a new peak is encountered, the histogram shifts to make room for that new peak.

You can choose to hide or display this panel in the Scope menu by selecting **Tools** > **Measurements** > **CCDF Measurements**. Alternatively, in the Scope toolbar, click the CCDF Measurements button.



- **Plot Gaussian reference** — Select **Plot Gaussian reference** to show the Gaussian white noise reference signal on the plot.
- **Probability (%)** — The percentage of the signal that contains the power level above the value listed in the **dB above average** column
- **dB above average** — The expected minimum power level at the associated **Probability (%)**.

- **Average Power** — The average power level of the signal since the start of simulation or from the last reset.

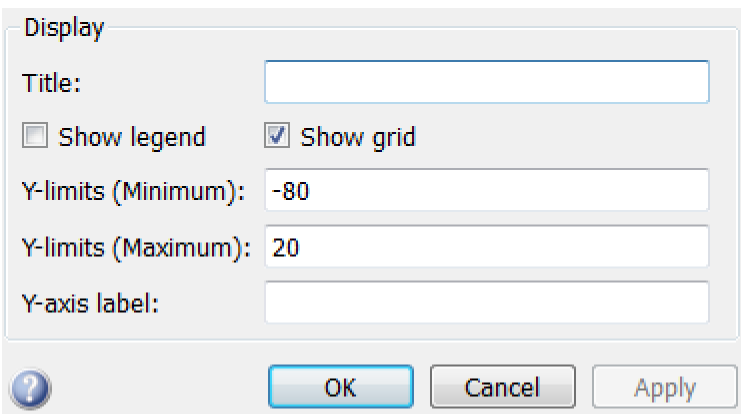
  Max Power — The maximum power level of the signal since the start of simulation or from the last reset.
- **PAPR** — The ratio of the peak power to the average power of the signal. PAPR should be less that 100 dB to obtain accurate CCDF measurements. If PAPR is above 100 dB, only the highest 100 dB power levels are plotted in the display and shown in the distribution table.
- **Sample Count** — The total number of samples used to compute the CCDF.
- **Reset** — Clear all current CCDF measurements and restart.

Visuals — Spectrum Properties

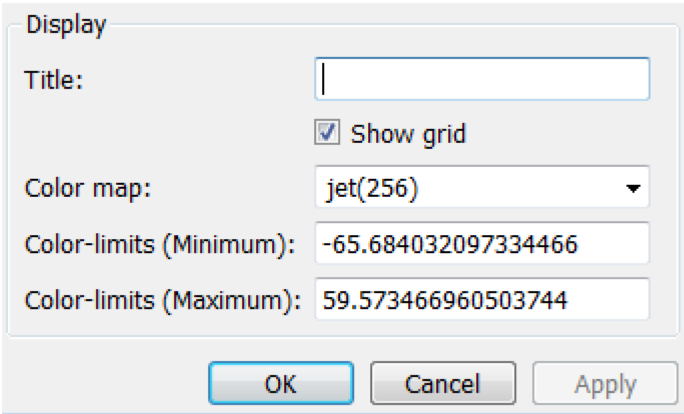
The Visuals—Spectrum Properties dialog box controls the visual configuration settings of the Spectrum Analyzer display. From the Spectrum Analyzer menu, select **View** > **Configuration Properties** to open this dialog box. Alternatively, in the Spectrum Analyzer toolbar, click the Configuration Properties  button.

Display Pane

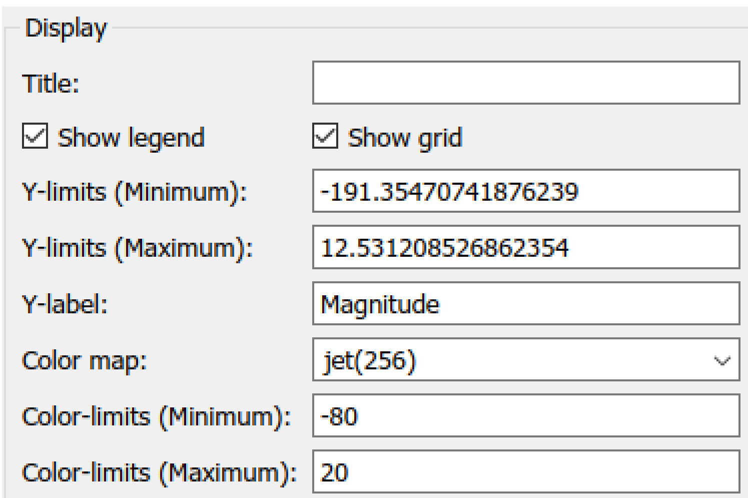
When the Spectrum **View** is `Spectrum`, the **Display** pane of the Visuals—Spectrum Properties dialog box appears as follows:



When the Spectrum **View** is Spectrogram the **Display** pane of the Visuals—Spectrum Properties dialog box appears as follows:



When the Spectrum **View** is Spectrum or spectroogram the **Display** pane of the Visuals—Spectrum Properties dialog box appears as follows:



Title

Specify the display title as a character vector. Enter %<SignalLabel> to use the signal labels in the Simulink Model as the axes titles. This property is Tunable (Simulink).

By default, the display has no title.

Show legend

Select this check box to show the legend in the display. The channel legend displays a name for each channel of each input signal. When the legend appears, you can place it anywhere inside of the scope window. To turn off the legend, clear the **Show legend** check box.

You can edit the name of any channel in the legend by double-clicking the current name and entering a new channel name. By default, if the signal has multiple channels, the scope uses an index number to identify each channel of that signal. To change the appearance of any channel of any input signal in the scope window, from the scope menu, select **View** > **Style**.

The legend lets you modify what signals are shown. To show only one signal, click the signal name. To toggle a signal on/off, right-click the signal name.

This parameter applies only when the **View Type** is `Spectrum` or `Spectrum and spectrogram`. Tunable (Simulink)

Show grid

When you select this check box, a grid appears in the display of the scope figure. To hide the grid, clear this check box. Tunable (Simulink)

Y-limits (Minimum)

Specify the minimum value of the *y*-axis. Tunable (Simulink)

Y-limits (Maximum)

Specify the maximum value of the *y*-axis. Tunable (Simulink)

Y-axis label

Specify the text for the scope to display to the left of the *y*-axis. Regardless of this property, Spectrum Analyzer always displays power units after this text as one of `'dBm'`, `'dBW'`, `'Watts'`, `'dBm/Hz'`, `'dBW/Hz'`, `'Watts/Hz'`, `Vrms`, or `dBV`Tunable (Simulink).

Color map

Select the color map for the spectrogram, or enter a 3-column matrix expression for the color map. See `colormap` for information. Tunable (Simulink).

Color-limits (Minimum)

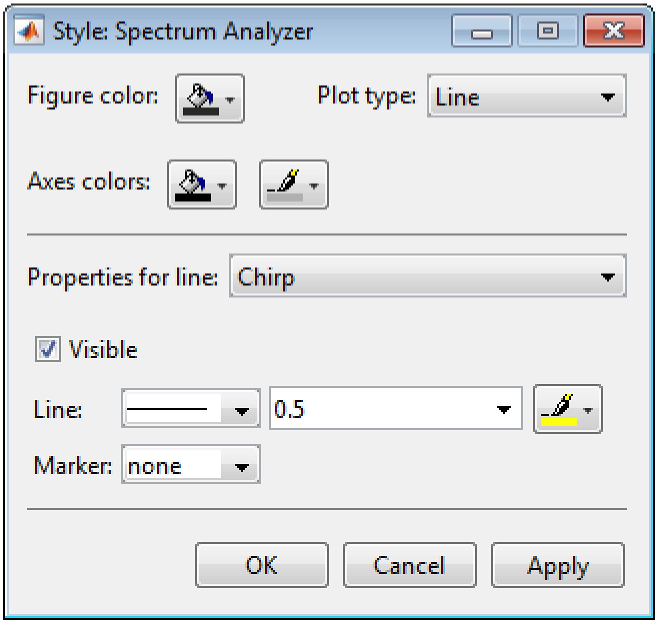
Set the signal power for the minimum color value of the spectrogram. Tunable (Simulink).

Color-limits (Maximum)

Set the signal power for the maximum color value of the spectrogram. Tunable (Simulink).

Style Dialog Box

In the **Style** dialog box, you can customize the style of spectrum display. This dialog box is not available for the spectrogram view. You are able to change the color of the figure, the background and foreground colors of the axes, and properties of the lines. From the Spectrum Analyzer menu, select **View** > **Style** to open this dialog box.



Properties

The **Style** dialog box allows you to modify the following properties of the Spectrum Analyzer figure:

Figure color

Specify the color that you want to apply to the background of the scope figure. By default, the figure color is gray.

Plot type

Specify whether to display a `Line` or `Stem` plot.

Axes colors

Specify the color that you want to apply to the background of the axes.

Properties for line

Specify the channel for which you want to modify the visibility, line properties, and marker properties.

Visible

Specify whether the selected channel should be visible. If you clear this check box, the line disappears.

Line

Specify the line style, line width, and line color for the selected channel.

Marker

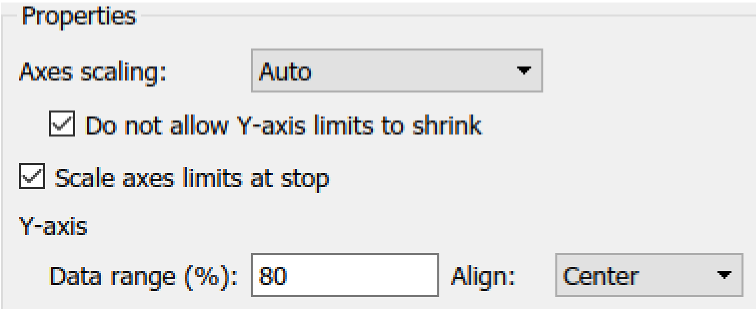
Specify marks for the selected channel to show at its data points. This parameter is similar to the `Marker` property for plot objects. You can choose any of the marker symbols from the dropdown.

Tools — Axes Scaling Properties

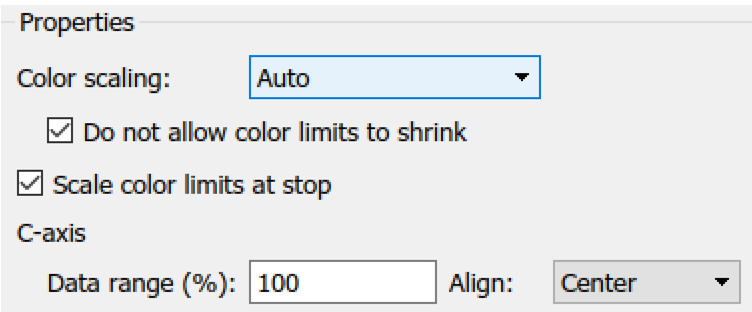
The Tools — Axes Scaling Properties dialog box allows you to automatically zoom in on and zoom out of your data. You can also scale the axes and color of the Spectrum Analyzer. In the Spectrum Analyzer menu, select **Tools** > **Scaling Properties** to open this dialog box.

Properties

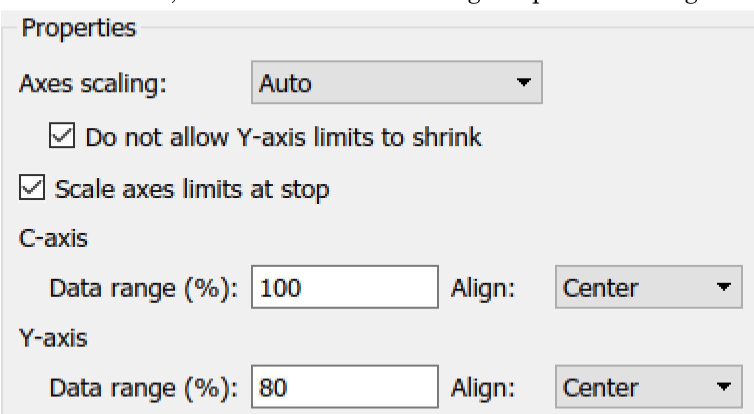
For the spectrum view, the Tools—Axes Scaling Properties dialog box appears as:



For spectrogram view, the Tools—Axes Scaling Properties dialog box appears as:



For dual view, the Tools—Axes Scaling Properties dialog box appears as:



Note: The X-axis scaling options only apply when using CCDF measurements.

Axes scaling/Color scaling

Specify when the scope automatically scales the axes. If the spectrogram is displayed, specify when the scope automatically scales the color. You can select one of the following options:

- `Manual` — When you select this option, the scope does not automatically scale the axes or color. You can manually scale the axes or color in any of the following ways:
  - Select **Tools** > **Scaling Properties**.
  - Press one of the **Scale Axis Limits** toolbar buttons.
  - When the scope figure is the active window, press **Ctrl** and **A** simultaneously.
- `Auto` — When you select this option, the scope scales the axes or color as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** or **Do not allow color limits to shrink** .
- `After N Updates` — Selecting this option causes the scope to scale the axes or color after a specified number of updates. This option is useful and more efficient when your scope display starts with one axis scale, but quickly reaches a different steady state axis scale. Selecting this option shows the **Number of updates** edit box.

By default, this parameter is set to `Auto`, and the scope does not shrink the *y*-axis limits when scaling the axes or color. Tunable (Simulink).

Do not allow Y-axis limits to shrink / Do not allow color limits to shrink

When you select this property, the *y*-axis are allowed to grow during axes scaling operations. If the spectrogram is displayed, selecting this property allows the color limits to grow during axis scaling. If you clear this check box, the *y*-axis or color limits can shrink during axes scaling operations.

This property appears only when you select `Auto` for the **Axis scaling** or **Color scaling** property. When you set the **Axes scaling** or **Color scaling** property to `Manual` or `After N Updates`, the *y*-axis or color limits can shrink. Tunable (Simulink).

Number of updates

Specify as a positive integer the number of updates after which to scale the axes. If the spectrogram is displayed, this property specifies the number of updates after which to scale the color. This property appears only when you select `After N Updates` for the **Axes scaling** or **Color scaling** property. Tunable (Simulink).

Scale axes limits at stop/Scale color limits at stop

Select this check box to scale the axes when the simulation stops. If the spectrogram is displayed, select this check box to scale the color when the simulation stops. The *y*-axis is always scaled. The *x*-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

Y-axis Data range (%) / Color-limits Data range

Set the percentage of the *y*-axis that the scope uses to display the data when scaling the axes. If the spectrogram is displayed, set the percentage of the power values range within the colormap. Valid values are from 1 through 100. For example, if you set this property to `100`, the Scope scales the *y*-axis limits such that your data uses the entire *y*-axis range. If you then set this property to `30`, the scope increases the *y*-axis range or color such that your data uses only 30% of the *y*-axis range or color. Tunable (Simulink).

Y-axis Align / Color-limits Align

Specify where the scope aligns your data along the *y*-axis when it scales the axes. If the spectrogram is displayed, specify where the scope aligns your data along the *y*-axis when it scales the color. You can select `Top`, `Center`, or `Bottom`. Tunable (Simulink).

Autoscale X-axis limits

Check this box to allow the scope to scale the *x*-axis limits when it scales the axes. If **Axes scaling** is set to `Auto`, checking **Autoscale X-axis limits** only scales the data currently within the axes, not the entire signal in the data buffer. If **Autoscale X-axis limits** is on and the resulting axis is greater than the span of the scope, trigger position markers will not be displayed. Triggers are controlled using the Trigger Measurements panel. Tunable (Simulink).

X-axis Data range (%)

Set the percentage of the *x*-axis that the scope uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the scope scales the *x*-axis limits such that your data uses the entire*x*-axis range. If you then set this property to `30`, the scope increases the *x*-axis range such that your data uses only 30% of the *x*-axis range. Use the *x*-axis **Align** property to specify data placement along the *x*-axis.

This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

X-axis Align

Specify how the scope aligns your data along the *x*-axis: `Left`, `Center`, or `Right`. This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

Algorithms

Spectrum Analyzer uses the `RBW` or the `Window Length` setting in the **Spectrum Settings** panel to determine the data window length. The value of the `FrequencyResolutionMethod` property determines whether RBW or window length is used. Then, it partitions the input signal into a number of windowed data segments. Finally, Spectrum Analyzer uses the modified periodogram method to compute spectral updates, averaging the windowed periodograms for each segment.

1 Spectrum Analyzer requires that a minimum number of samples have been provided before it computes a spectral estimate. This number of input samples required to compute one spectral update is shown as **Samples/update** in the **Main options** pane. This value is directly related to resolution bandwidth, *RBW*, by the following equation or to the window length, by the equation shown in step 1b.

$$N_{samples} = \frac{\left(1 - \frac{O_p}{100}\right) \times NENBW \times F_s}{RBW}$$

The normalized effective noise bandwidth, *NENBW*, is a factor that depends on the windowing method. Spectrum Analyzer shows NENBW in the **Window Options** pane of the **Spectrum Settings** panel. Overlap percentage, O_p, is the value of the **Overlap %** parameter in the **Window Options** pane of the **Spectrum Settings** panel. F_s is the sample rate of the input signal. Spectrum Analyzer shows sample rate in the **Main Options** pane of the **Spectrum Settings** panel.

a When in RBW mode, the window length required to compute one spectral update, N_{window}, is directly related to the resolution bandwidth and normalized effective noise bandwidth by the following equation.

$$N_{window} = \frac{NENBW \times F_s}{RBW}$$

When in WindowLength mode, the window length is used as specified.

b The number of input samples required to compute one spectral update, $N_{samples}$, is directly related to the window length and the amount of overlap by the following equation.

$$N_{samples} = \left(1 - \frac{O_p}{100}\right) N_{window}$$

When you increase the overlap percentage, fewer new input samples are needed to compute a new spectral update. For example, if the window length is 100, then the number of input samples required to compute one spectral update is given as shown in the following table.

O_p	$N_{samples}$
0%	100
50%	50
80%	20

c The normalized effective noise bandwidth, *NENBW*, is a window parameter determined by the window length, N_{window}, and the type of window used. If $w(n)$ denotes the vector of N_{window} window coefficients, then *NENBW* is given by the following equation.

$$NENBW = N_{window} \frac{\sum_{n=1}^{N_{window}} w^2(n)}{\left[\sum_{n=1}^{N_{window}} w(n)\right]^2}$$

d When in RBW mode, you can set the resolution bandwidth using the value of the **RBW** parameter on the **Main options** pane of the **Spectrum Settings** panel. You must specify a value to ensure that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two, as given in the following equation.

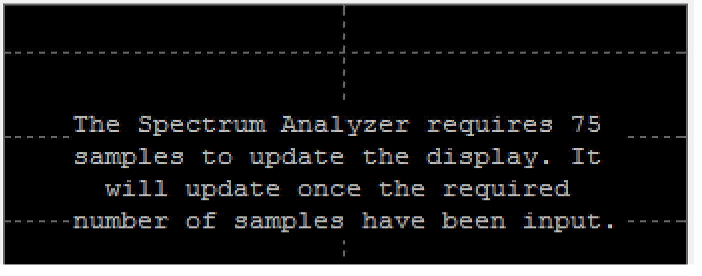
$$\frac{span}{RBW} > 2$$

By default, the **RBW** parameter on the **Main options** pane is set to `Auto`. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified frequency span. Thus, when you set **RBW** to `Auto`, it is calculated by the following equation. $RBW_{auto} = \frac{span}{1024}$

e When in window length mode, you specify N_{window} and the resulting RBW is

$$\frac{NENBW * Fs}{N_{window}}$$

In some cases, the number of samples provided in the input are not sufficient to achieve the resolution bandwidth that you specify. When this situation occurs, Spectrum Analyzer produces a message on the display, as shown in the following figure.



Spectrum Analyzer removes this message and displays a spectral estimate as soon as enough data has been input. Notice that this behavior differs from the Spectrum Scope block in versions R2012b and earlier. If the **Buffer input** check box was selected, the Spectrum Scope block computed a spectral update using the number of samples given by the **Buffer size** parameter. Otherwise, the Spectrum Scope block computed a spectral update using the number of samples in each frame.

2 Spectrum Analyzer calculates and plots the power spectrum, power spectrum density, and RMS computed by the modified *Periodogram* estimator. For more information about the Periodogram method, see `periodogram` in the Signal Processing Toolbox documentation.

Power Spectral Density — The power spectral density (PSD) is given by the following equation.

$$PSD(f) = \frac{1}{P}\sum_{p=1}^{P}\frac{\left|\sum_{n=1}^{N_{FFT}} x^{(p)}[n]e^{-j2\pi f(n-1)T}\right|^2}{F_s \times \sum_{n=1}^{N_{window}} w^2[n]}$$

In this equation, $x[n]$ is the discrete input signal. On every input signal frame, Spectrum Analyzer generates as many overlapping windows as possible, each window denoted as $x^{(p)}[n]$, and computes their periodograms. Spectrum Analyzer displays a running average of the P most current periodograms.

Power Spectrum — The power spectrum is the product of the power spectral density and the resolution bandwidth, as given by the following equation.

$$P_{spectrum}(f) = PSD(f) \times RBW = PSD(f) \times \frac{F_s \times NENBW}{N_{window}} = \frac{1}{P}\sum_{p=1}^{P}\frac{\left|\sum_{n=1}^{N_{FFT}} x^{(p)}[n]e^{-j2\pi f(n-1)T}\right|^2}{\left[\sum_{n=1}^{N_{window}} w[n]\right]^2}$$

Spectrogram — You can plot any power as a spectrogram. Each line of the spectrogram is one periodogram. The time resolution of each line is 1/RBW, which is the minimum attainable resolution. Achieving the resolution you want may require combining several periodograms may be combined. You then use interpolation to calculate noninteger values of 1/RBW. In the spectrogram display, time scrolls from bottom to top, so the most recent data is shown at the bottom of the display. The offset shows the time value at which the center of the most current spectrogram line occurred.

Note: The number of FFT points (N_{fft}) is independent of the window length (N_{window}). You can set them to different values provided that N_{fft} is greater than or equal to N_{window}.

The *Occupied BW* is calculated as follows.

- Calculate the total power in the measured frequency range.
- Determine the lower frequency value. Starting at the lowest frequency in the range and moving upward, the power distributed in each frequency is summed until this sum is $\frac{100 - Occupied\ BW\ \%}{2}$ of the total power.
- Determine the upper frequency value. Starting at the highest frequency in the range and moving downward, the power distributed in each frequency is summed until it reaches $\frac{100 - Occupied\ BW\ \%}{2}$ of the total power.
- The bandwidth between the lower and upper power frequency values is the occupied bandwidth.
- The frequency halfway between the lower and upper frequency values is the center frequency.

The *Distortion Measurements* are computed as follows.

1. Spectral content is estimated by finding peaks in the spectrum. When the algorithm detects a peak, it ignores all adjacent content that decreases monotonically from the peak. After recording the width of the peak, it clears all monotonically decreasing values (that is, it treats all of these values as if they belong to the peak). Using this method, all spectral content centered at DC (0 Hz) is removed from the spectrum and the amount of bandwidth cleared (W_0) is recorded.
2. The fundamental power (P_1) is determined from the remaining maximum value of the displayed spectrum. A local estimate (Fe_1) of the fundamental frequency is made by computing the central moment of the power in the vicinity of the peak. The bandwidth of the fundamental power content (W_1) is recorded. Then, the power associated from the fundamental is removed as in step 1.
3. The power and width of the second, and higher order harmonics (P_2, W_2, P_3, W_3, etc.) are determined in succession by examining the frequencies closest to the appropriate multiple of the local estimate (Fe_1). Any spectral content that decreases in a monotonically about the harmonic frequency is removed from the spectrum first before proceeding to the next harmonic.

4 Once the DC, fundamental, and harmonic content is removed from the spectrum, the power of the remaining spectrum is examined for its sum ($P_{remaining}$) peak value ($P_{maxspur}$), and its median value ($P_{estnoise}$).

5 The sum of all the removed bandwidth is computed as $W_{sum} = W_0+W_1+W_2+...+W_n$.

The sum of powers of the second and higher order harmonics are computed as $P_{harmonic} = P_2+P_3+P_4+...+P_n$.

6 The sum of the noise power is then estimated as $P_{noise} = (P_{remaining}$*dF + $P_{estnoise}*W_{sum})/RBW$, where dF is the absolute difference between frequency bins, and RBW is the resolution bandwidth of the window.

7 The metrics for SNR, THD, SINAD, and SFDR are then computed from the estimates.

$$THD = 10 \cdot \log_{10}\left(\frac{P_{harmonic}}{P_1}\right)$$

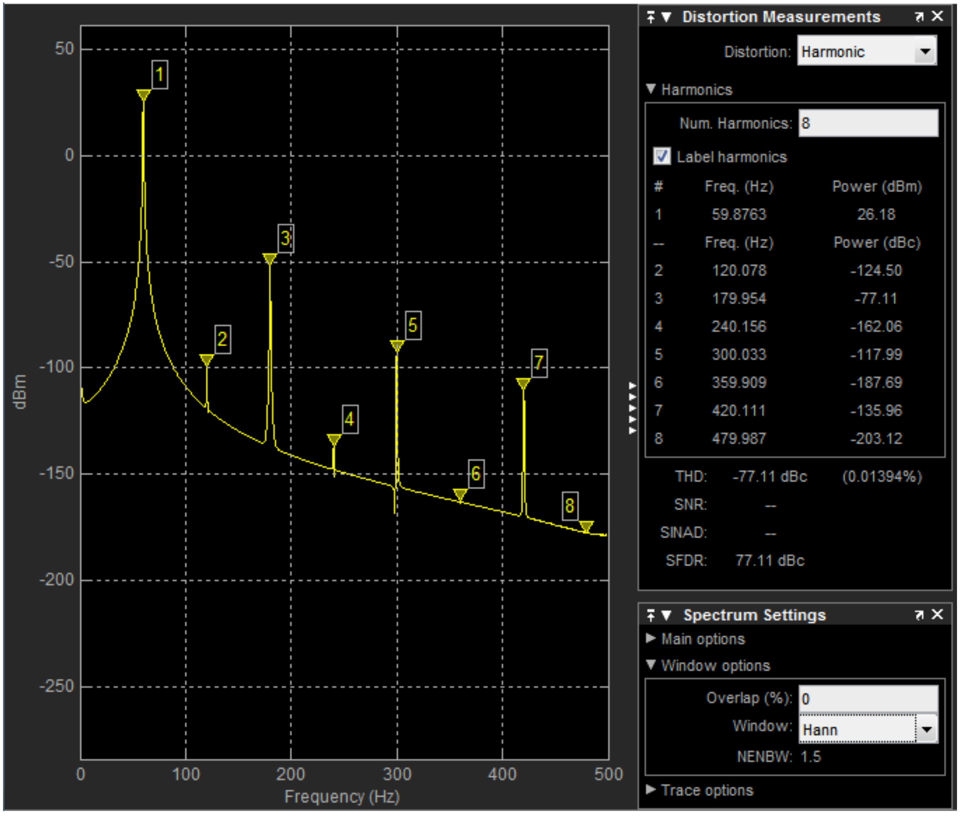
$$SINAD = 10 \cdot \log_{10}\left(\frac{P_1}{P_{harmonic} + P_{noise}}\right)$$

$$SNR = 10 \cdot \log_{10}\left(\frac{P_1}{P_{noise}}\right)$$

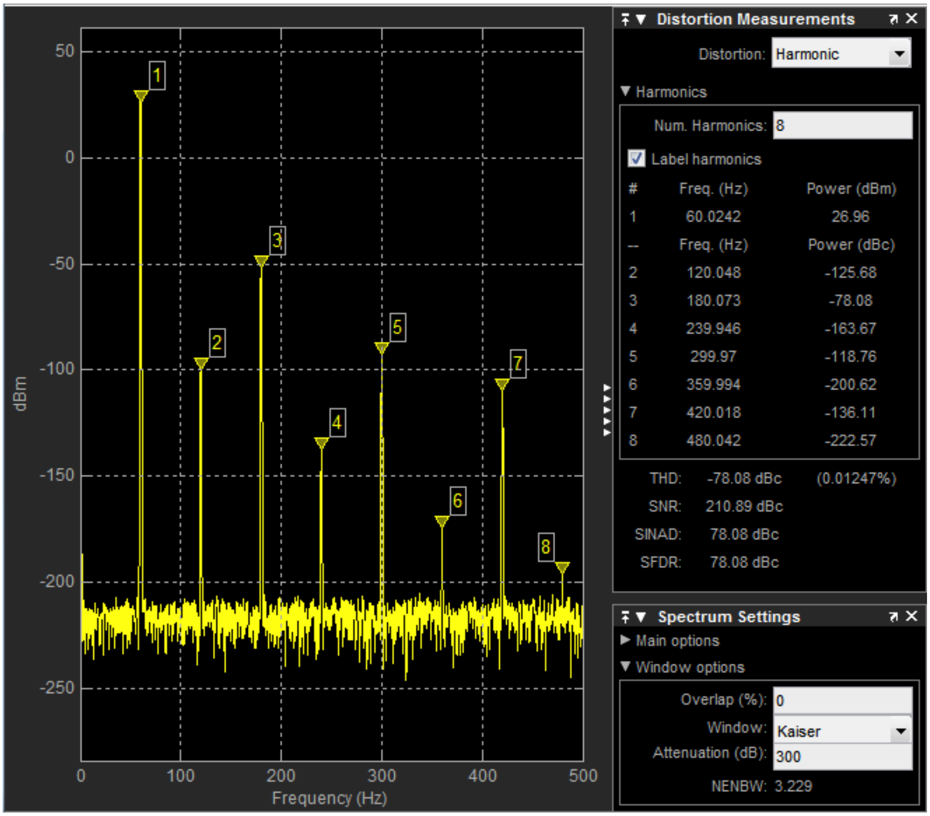
$$SFDR = 10 \cdot \log_{10}\left(\frac{P_1}{\max\left(P_{maxspur}, \max\left(P_2, P_3, ..., P_n\right)\right)}\right)$$

The following considerations apply to *Distortion Measurements*.

- The harmonic distortion measurements use the spectrum trace shown in the display as the input to the measurements. The default `Hann` window setting of the Spectrum Analyzer may exhibit leakage that can completely mask the noise floor of the measured signal.



The harmonic measurements attempt to correct for leakage by ignoring all frequency content that decreases monotonically away from the maximum of harmonic peaks. If the window leakage covers more than 70% of the frequency bandwidth in your spectrum, you may see a blank reading (–) reported for **SNR** and **SINAD**. Consider using a Kaiser window with a high attenuation (up to 330dB) to minimize spectral leakage if your application can tolerate the increased equivalent noise bandwidth (ENBW) of the Kaiser window.



- The DC component is ignored.
- After windowing, the width of each harmonic component masks the noise power in the neighborhood of the fundamental frequency and harmonics. To estimate the noise power in each region, Spectrum Analyzer computes the median noise level in the nonharmonic areas of the spectrum. It then extrapolates that value into each region.
- *N*th order intermodulation products occur at

  $A*F1 + B*F2$

  where $F1$ and $F2$ are the sinusoid input frequencies and $|A| + |B| = N$. A and B are integer values.

- For intermodulation measurements, the third-order intercept (TOI) point is computed as follows, where P is power in decibels of the measured power referenced to one milliwatt (dBm).:
  - $TOI_{lower} = P_{F1} + (P_{F2} - P_{(2F1-F2)})/2$
  - $TOI_{upper} = P_{F2} + (P_{F1} - P_{(2F2-F1)})/2$
  - $TOI = + (TOI_{lower} + TOI_{upper})/2$

Differences from Spectrum Scope Block

All Simulink models containing Spectrum Scope blocks load with Spectrum Analyzer blocks in R2013a or later. Several options that were available on the Parameters dialog box of the Spectrum Scope block are no longer available or have changed. The parameters of Spectrum Scope map to Spectrum Analyzer parameters in the following manner.

R2012b Spectrum Scope Block Parameters dialog box Tab name	R2012b Spectrum Scope Parameter	R2013a Spectrum Analyzer Change	R2013a Spectrum Analyzer Equivalent Parameter
Scope Properties	**Buffer input** check box	R2013a Spectrum Analyzer does not require that input signals are buffered. Spectrum Analyzer determines the number of samples needed using the value of the **RBW** parameter. Regardless of whether the input is a frame-based or sample-based signal, Spectrum Analyzer calculates the spectrum once it has acquired the requisite number of samples.	For Spectrum Scope blocks in R2012b or earlier models, the equivalent R2013a Spectrum Analyzer **RBW** value is given by the equation: $$RBW = \frac{NENBW \times F_s}{N_{window}}$$ In the preceding equation, $NENBW$ is the window constant calculated for a window length of 1000, F_s is the sample rate of the block,

R2012b Spectrum Scope Block Parameters dialog box Tab name	R2012b Spectrum Scope Parameter	R2013a Spectrum Analyzer Change	R2013a Spectrum Analyzer Equivalent Parameter
			and N_{window} is the buffer length. If the input signal to the R2012b Spectrum Scope block was frame-based and the **Buffer input** check box was cleared, then the R2013a Spectrum Analyzer computes the **RBW** value with N_{window} set to the frame size of the input signal.
Scope Properties	**Buffer size** parameter	R2013a Spectrum Analyzer uses the **RBW** parameter to determine the requisite number of samples to calculate the spectrum, instead of using the buffer size or frame length.	For Spectrum Scope blocks in R2012b or earlier models, if the input signal was frame-based and the **Buffer input** check box was selected, then the R2013a Spectrum Analyzer computes the **RBW** value with N_{window} set to the value of the **Buffer size** parameter.

R2012b Spectrum Scope Block Parameters dialog box Tab name	R2012b Spectrum Scope Parameter	R2013a Spectrum Analyzer Change	R2013a Spectrum Analyzer Equivalent Parameter
Scope Properties	**Buffer Overlap** parameter	R2013a Spectrum Analyzer has an **Overlap %** parameter that is directly related to buffer overlap.	R2013a Spectrum Analyzer will compute its **Overlap %** using the equation: $$O_p = {O_l}/{N_{window}} \times 100$$ In the preceding equation, O_p is **Overlap %** parameter value, O_l is the R2012b Spectrum Scope **Buffer overlap** parameter value, and N_{window} is the buffer length.
Scope Properties	**Treat Mx1 and unoriented sample-based signals as**	R2013a Spectrum Analyzer defaults to treating Mx1 and unoriented sample-based signals as one channel.	Spectrum Scope blocks in R2012b or earlier models with **Treat Mx1 and unoriented sample-based signals as** set to `M Channels` will have the Spectrum Analyzer property `TreatMby1SignalAsOneChannel` set to `false`. This property is available only via the Scope Configuration object.
Scope Properties	**Window** parameter	R2013a Spectrum Analyzer does not have the `Bartlett`, `Blackman`, `Triang`, or `Hanning` settings.	Spectrum Scope blocks in R2012b or earlier models with a window parameter set to any of these values will have their **Window** parameter set to `Hann` in the R2013a Spectrum Analyzer.

R2012b Spectrum Scope Block Parameters dialog box Tab name	R2012b Spectrum Scope Parameter	R2013a Spectrum Analyzer Change	R2013a Spectrum Analyzer Equivalent Parameter
Scope Properties	**Window Sampling** parameter	R2013a Spectrum Analyzer does not have a `Periodic` option. All window sampling is now symmetric in the R2013a Spectrum Analyzer.	n/a
Display Properties	**Persistence** check box — this setting would execute the equivalent of the MATLAB `hold on` command, adding another line for each spectrum computation on the display.	This option is not available in the R2013a Spectrum Analyzer, which has replaced this feature with the trace options, **Normal Trace**, **Max Hold Trace**, and **Min Hold Trace**.	Spectrum Scope blocks in R2012b or earlier models with persistence enabled will have their **Max Hold Trace** check box selected in the R2013a Spectrum Analyzer.
Display Properties	**Compact Display** check box	There is no equivalent capability in the R2013a Spectrum Analyzer.	n/a
Axis Properties	**Inherit Sample time from input** check box	R2013a Spectrum Analyzer always uses the sample time of the input signal.	n/a

R2012b Spectrum Scope Block Parameters dialog box Tab name	R2012b Spectrum Scope Parameter	R2013a Spectrum Analyzer Change	R2013a Spectrum Analyzer Equivalent Parameter
Axis Properties	**Frequency display limits** parameter	R2013a Spectrum Analyzer determines the range of frequencies calculated based on the **Full Span**, **FStart (Hz)**, and **FStop (Hz)** parameters.	If this parameter was set to: • `Auto` — R2013a Spectrum Analyzer selects the **Full Span** check box on the Spectrum Settings panel, Main options pane. • `User-defined` — R2013a Spectrum Analyzer clears the **Full Span** check box on the Spectrum Settings panel Main options pane.
Axis Properties	**Minimum frequency (Hz)** parameter	R2013a Spectrum Analyzer determines the range of frequencies calculated based on the **Full Span**, **FStart (Hz)**, and **FStop (Hz)** parameters.	If the `User-defined` parameter was chosen, then this parameter maps to the R2013a Spectrum Analyzer **FStart (Hz)** parameter.
Axis Properties	**Maximum frequency (Hz)** parameter	R2013a Spectrum Analyzer determines the range of frequencies calculated based on the **Full Span**, **FStart (Hz)**, and **FStop (Hz)** parameters.	If the `User-defined` parameter was chosen, then this parameter maps to the R2013a Spectrum Analyzer **FStop (Hz)** parameter.

R2012b Spectrum Scope Block Parameters dialog box Tab name	R2012b Spectrum Scope Parameter	R2013a Spectrum Analyzer Change	R2013a Spectrum Analyzer Equivalent Parameter
Line Properties	**Line visibilities**, **Line styles**, **Line markers**, and **Line colors** parameters	There are no equivalent capabilities in the R2013a Spectrum Analyzer.	Once the simulation has started, you can modify the line styles, markers, and colors using the Style dialog box.

The R2012b Spectrum Scope allowed you to retain the axes limits over multiple simulations by selecting **Axes** > **Save Axes Settings**. There is no equivalent capability in the R2013a Spectrum Analyzer. However, you can automatically scale the axes to a specified range using the Tools — Axes Scaling Properties dialog box.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned)

Supported Simulation Modes

You can use the scope block in models running the following supported simulation modes.

Mode	Supported	Notes and Limitations
Normal	Yes	

Mode	Supported	Notes and Limitations
Accelerator	Yes	
Rapid Accelerator	Yes	You can use Rapid Accelerator mode as a method to increase the execution speed of your Simulink model. Rapid Accelerator mode creates an executable that includes the solver and model methods. This executable resides outside MATLAB and Simulink. Rapid Accelerator mode uses External mode to communicate with Simulink. For more information about Rapid Accelerator mode, see "Acceleration" (Simulink).
PIL	No	
SIL	No	
External	Yes	You can use External mode to tune block parameters in real time and view block outputs in many types of blocks and subsystems. External mode establishes communication between a host system, where the Simulink environment resides, and a target system, where the executable runs after code generation and the build process. For more information about External mode, see "Set Up and Use Host/Target Communication Channel" (Simulink Coder). The scope does not support data archiving. See "Set External Mode Data Archiving Parameters" (Simulink Desktop Real-Time).

For more information about these modes, see "How Acceleration Modes Work" (Simulink).

See Also

See Also

dsp.SpectrumAnalyzer | `Array Plot` | `sptool` | `Time Scope`

Topics

"Display Frequency-Domain Data in Spectrum Analyzer"

Spectrum Analyzer Measurements

Introduced in R2014b

Spectrum Estimator

Estimate power spectrum or power-density spectrum

Library

Estimation / Power Spectrum Estimation

`dspspect3`

Description

The Spectrum Estimator block outputs the power spectrum or power-density spectrum of a real or complex input signal, using the Welch method of averaged modified periodograms and the filter bank approach.

When you choose the filter bank approach, the block uses an analysis filter bank to estimate the power spectrum. The filter bank approach produces a spectral estimate with a higher resolution, a more accurate noise floor, and more precise peaks than the Welch method, with low or no spectral leakage. They come at the expense of increased computation and slower tracking.

When you choose the Welch method, the block computes the averaged modified periodograms to compute the spectral estimate. The block buffers the input data into overlapping segments. Use the block parameters to set the length of the data segments, the amount of data overlap between consecutive segments, and other features of the power spectrum.

For more information on the Welch method and the filter bank method, see “Algorithms” on page 2-1617.

Each column of the input signal is treated as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then it is interpreted as a single channel.

Parameters

Main Tab

Method

Specify the spectral estimation method.

- `Filter bank` (default) — An analysis filter bank splits the broadband input signal into multiple narrow subbands. The block computes the power in each narrow frequency band, and the computed value is the spectral estimate over the respective frequency band.
- `Welch` — The block uses the Welch averaged modified periodograms method to compute the power spectrum over the narrow subbands.

Number of taps per band

Specify the number of filter coefficients, or taps, for each frequency band. This value corresponds to the number of filter coefficients per polyphase branch. The total number of filter coefficients is equal to **Number of taps per band** times the FFT length.

This parameter applies when you set **Method** to `Filter bank`. The default is 12.

Spectrum type

Type of spectrum to compute. You can set this parameter to:

- `Power` (default) — Compute the power spectrum.
- `Power density` — Compute the power spectral density.

This parameter is nontunable.

Frequency resolution method

Frequency resolution method. You can set this parameter to:

- `Auto` (default) — The Spectrum Estimator block computes the resolution bandwidth (RBW) so that the frequency span fits 1024 RBW intervals.

  - Welch method — The window length, *winLen*, is calculated using $winLen = NENBW \times Fs / RBW$. *NENBW* is the equivalent noise bandwidth of the window and *Fs* is the sample rate.
  - Filter bank method — The FFT length is the ceiling of the ratio of **Sample rate (Hz)** to the computed resolution bandwidth.
- `RBW` — Specify the resolution bandwidth, which is used to determine the window length (Welch method) or the FFT length (filter bank method). When the block uses the Welch method, the behavior is equivalent to that of the `Spectrum Analyzer` block. The window length is calculated using $winLen = NENBW \times Fs / RBW$. *NENBW* is the equivalent noise bandwidth of the window and *Fs* is the sample rate. The FFT length is equal to the ceiling of the ratio of **Sample rate (Hz)** to **RBW (Hz)**.
- `Window length` — Specify the window or segment length to use in the Welch algorithm. This option appears when you set **Method** to `Welch`.
- `Number of frequency bands` — Specify the number of polyphase branches of the analysis filter bank. This value corresponds to the FFT length that the filter bank uses. This option appears when you set **Method** to `Filter bank`.

This parameter is nontunable.

RBW (Hz)

Resolution bandwidth, specified as a positive scalar in Hz. The default is `5`. This parameter applies when you set **Frequency resolution method** to `RBW`. The ceiling of the ratio of the frequency span to RBW must be greater than `2`.

This parameter is nontunable.

Number of bands source

Source of the number of frequency bands. This parameter applies when you set **Method** to `Filter bank` and **Frequency resolution method** to `Number of frequency bands`. You can set this parameter to:

- `Same as input frame length` (default) — The FFT length is set to the frame size of the input.
- `Specify on dialog` — The FFT length is the value you specify in **Number of bands**.

This parameter is nontunable.

Number of bands

Number of frequency bands, or the FFT length the filter bank uses to compute the power spectral estimate, specified as a positive scalar. The default is `1024`. This parameter applies when you set **Method** to `Filter bank`, **Frequency resolution method** to `Number of frequency bands`, and **Number of bands source** to `Specify on dialog`. This parameter is nontunable.

Window length source

Source of the window length value. This parameter applies when you set **Method** to `Welch` and **Frequency resolution method** to `Window length`. You can set this parameter to:

- `Same as input frame length` (default) — Window length is set to the frame size of the input. Specify this option to obtain behavior equivalent to that of the `Periodogram` block.
- `Specify on dialog` — Window length is the value you specify in the **Window length** parameter.

This parameter is nontunable.

Window length

Length of the window used to compute the spectrum estimate, specified as a positive integer scalar greater than `2`. The default is `1024`. This parameter applies when you set **Method** to `Welch`, **Frequency resolution method** to `Window length`, and **Window length source** to `Specify on dialog`. This parameter is nontunable.

FFT length source

Source of the FFT length value. This parameter applies when you set **Method** to `Welch` and **Frequency resolution method** to `Window length`. You can set this parameter to:

- `Auto` (default) — The block sets the FFT length to the frame size of the input.
- `Property` — The block sets the FFT length to the value you specify in **FFT length**.

This parameter is nontunable.

FFT length

Length of the FFT used to compute the spectrum estimates, specified as a positive integer scalar. This parameter applies when you set **Method** to `Welch`, **Frequency resolution method** to `Window length`, and **FFT length source** to `Property`. The default is `1024`. This parameter is nontunable.

Inherit sample rate from input

When you select this check box, the block sample rate is computed as N/T_s, where N is the frame size of the input signal and T_s is the sample time of the input signal.

This check box applies when you do one of the following:

- Set **Method** to `Welch` and **Frequency resolution method** to `Window length`.
- Set **Method** to `Filter bank` and **Frequency resolution method** to `Number of frequency bands`.

When you clear this check box, the block sample rate is the value you specify in **Sample rate (Hz)**. By default, this check box is selected. This parameter is nontunable.

Sample rate (Hz)

Sample rate of the input signal, specified as a positive scalar. The default is `44100`. This parameter applies when you do one of the following:

- Set **Frequency resolution method** to `Auto` or `RBW`.
- Set **Method** to `Welch`, **Frequency resolution method** to `Window length`, and clear the **Inherit sample rate from input** check box.
- Set **Method** to `Filter bank`, **Frequency resolution method** to `Number of frequency bands`, and clear the **Inherit sample rate from input** check box.

This parameter is nontunable.

Window function

Window function the Welch algorithm uses, specified as one of `Chebyshev | Flat Top | Hamming | Hann | Kaiser | Rectangular`. This parameter appears when you set **Method** to `Welch`. The default is `Hann`. This parameter is nontunable.

Sidelobe attenuation of window (dB)

Sidelobe attenuation of the window, specified as a real positive scalar greater than or equal to `45`, in dB. The default is `60`. This parameter appears when you set **Method** to `Welch` and **Window function** to `Chebyshev` or `Kaiser`. This parameter is nontunable.

Number of spectral averages

Number of spectral averages, specified as a positive integer scalar. The default is `1`. The spectrum estimator computes the current power spectrum estimate by averaging the last *N* power spectrum estimates, where *N* is the number of spectral averages defined in **Number of spectral averages**. This parameter is nontunable.

Advanced Tab

Window overlap (%)

Percentage of overlap between successive data windows, specified as a scalar from `0` and `100`. The default value is `0`. To enable this parameter, on the **Main Tab**, set **Method** to `Welch`. This parameter is nontunable.

Reference load (ohms)

Load used as a reference to compute the power values, specified as a real positive scalar expressed in ohms. The default value is `1`. This parameter is nontunable.

Frequency range

Frequency range of the spectrum estimator. You can set this parameter to:

- `One-sided` — The spectrum estimator computes the one-sided spectrum of a real input signal. When the FFT length, `NFFT`, is even, the spectrum estimate has length (`NFFT`/2) + `1` and is computed over the frequency range `[0 SampleRate/2]`. `SampleRate` is the sample rate of the input signal. When `NFFT` is odd, the spectrum estimate has length (`NFFT` + 1)/2 and is computed over the frequency range [`0 SampleRate/2`).
- `Two-sided` — The spectrum estimator computes the two-sided spectrum of a complex or real input signal. The length of the spectrum estimate is equal to the FFT length. The spectrum estimate is computed over the frequency range [`0 SampleRate`), where `SampleRate` is the sample rate of the input signal.
- `Centered` (default) — The spectrum estimator computes the centered two-sided spectrum of a complex or real input signal. The length of the spectrum estimate is equal to the FFT length. The spectrum estimate is computed over the frequency range (`-SampleRate/2 SampleRate/2`] when the FFT length is even and `(-SampleRate/2 SampleRate/2)` when the FFT length is odd.

This parameter is nontunable.

Power units

Units used to measure power. You can set this parameter to:

- `'Watts'` (default) — The spectrum estimator measures power in watts.
- `'dBw'` — The spectrum estimator measures power in decibel-watts.

- 'dBm' — The spectrum estimator measures power in decibel-milliwatts.

This parameter is nontunable.

Output max-hold spectrum

When you select this check box, the block computes the max-hold spectrum of the input signal by keeping, at each frequency bin, the maximum value of all the power spectrum estimates. By default, this check box is not selected. This parameter is nontunable.

Output min-hold spectrum

When you select this check box, the block computes the min-hold spectrum of the input signal by keeping, at each frequency bin, the minimum value of all the power spectrum estimates. By default, this check box is not selected. This parameter is nontunable.

Output frequency vector

When you select this check box, the block outputs the frequency vector. By default, this check box is not selected. This parameter is nontunable.

Output effective RBW

When you select this check box, the block computes the effective resolution bandwidth. By default, this check box is not selected. This parameter is nontunable.

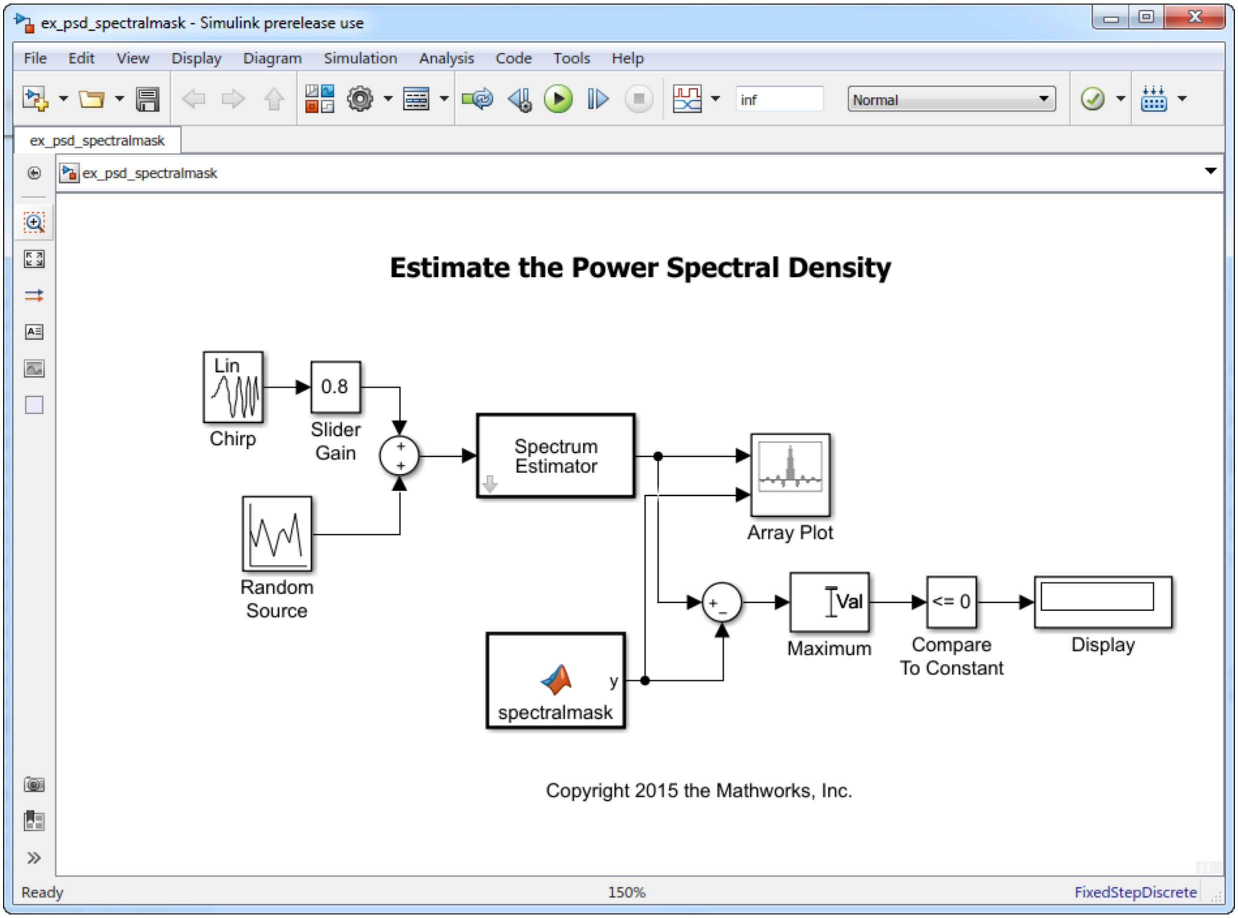
Simulate using

Type of simulation to run. You can set this parameter to:

- `Code generation` (default) — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Example

Estimate the Power Spectral Density (PSD) of a chirp signal using the Spectrum Estimator block. Compare the PSD data with a Bluetooth® spectral mask and determine if the PSD data complies with the mask.



To view the complete model, enter `ex_psd_spectralmask` in the MATLAB command prompt.

Input Signal

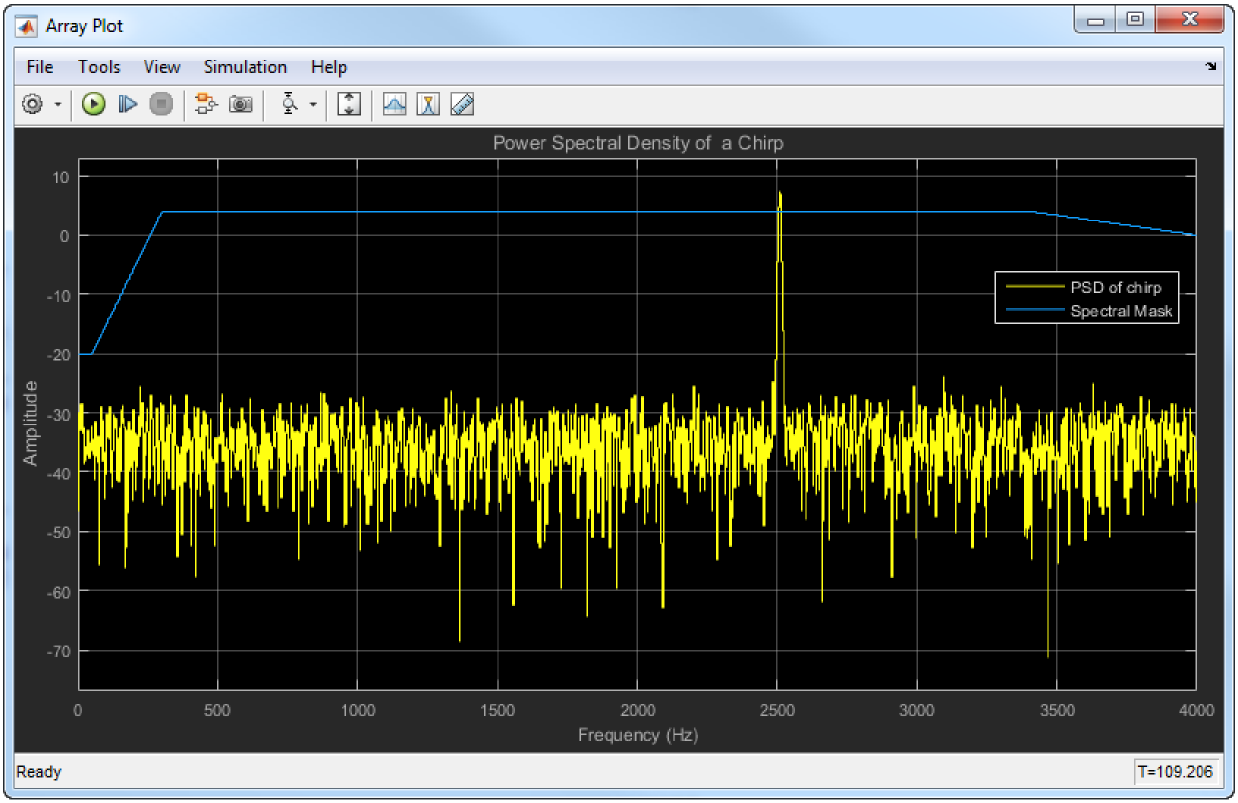
The input to the Spectrum Estimator block is a chirp signal embedded in Gaussian noise with zero mean and a variance of `0.01`. The chirp signal is amplified with a gain factor in the range [`0 1`].

Spectral Mask

The Spectral mask is created using the MATLAB Function block. The mask is based on the Bluetooth standard described in [5].

Live Processing

The Spectrum Estimator block estimates the PSD of the chirp. In this example, the PSD data is compared with the spectral mask. The Display block shows a 1 or 0, depending on whether the spectral data is within the mask or not. During simulation, you can change the power in the input signal by moving the slider in the Slider Gain block. Simultaneously, you can view this change in the Array Plot block.



Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Algorithms

Welch's Method of Averaged Modified Periodograms

When you choose the Welch method, the power spectrum estimate is averaged modified periodograms.

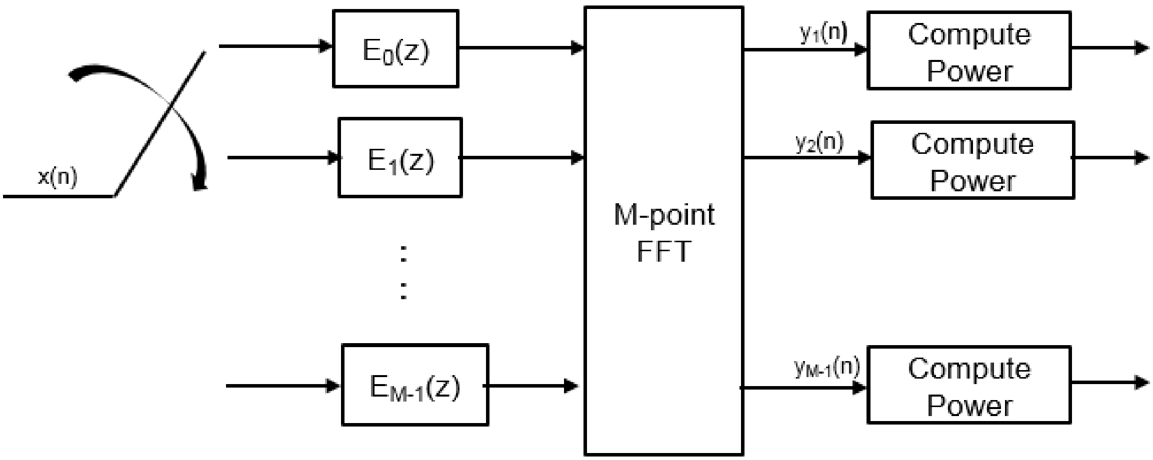
Let `x` be the input frame. Multiply `x` by the window and scale the result by the window power. Take the FFT of the signal, `Y`, and take the square magnitude using `Z = Y*conj(Y)`. Compute the current power spectrum estimate by averaging the last *N* `Z`s and scaling the answer by the sample rate. *N* is the number of spectral averages.

For further information refer to the "Algorithms" on page 2-1592 section in Spectrum Analyzer, which uses the same algorithm.

Filter Bank

The spectrum estimator uses an analysis filter bank to estimate the power spectrum. The number of taps per frequency band specifies the number of filter coefficients for each frequency band of the filter bank. The total number of filter coefficients is equal to number of taps per band times the FFT length.

The filter-bank-based spectrum estimator splits the broadband input signal, *x(n)*, into multiple narrow bands and computes the power in each band. Each value becomes the estimate of the power over that narrow frequency band. The power in all the narrow bands forms the spectral estimate vector.



For more details on the implementation, see the “Algorithm” on page 2-214 section in `dsp.Channelizer`.

References

[1] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996.

[2] Kay, Steven M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1999.

[3] Stoica, Petre, and Randolph L. Moses. *Spectral Analysis of Signals*. Englewood Cliffs, NJ: Prentice Hall, 2005.

[4] Welch, P. D. “The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms.” *IEEE Transactions on Audio and Electroacoustics*. Vol. 15, No. 2, June 1967, pp. 70–73.

[5] *Bluetooth Specification Version 4.2.* Bluetooth SIG. December 2014, p. 217. Specification of the Bluetooth System

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

When the FFT length is not a power of two, the executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

When the FFT length is a power of two, you can generate standalone C and C++ code from this block.

See Also

See Also

System Objects

dsp.SpectrumEstimator

Blocks

`Cross-Spectrum Estimator` | `Discrete Transfer Function Estimator` | `Periodogram` | `Spectrum Analyzer`

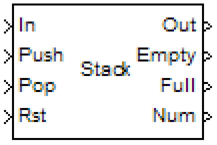
Topics

“Streaming Power Spectrum Estimation Using Welch's Method”

Introduced in R2015b

Stack

Store inputs into LIFO register



Library

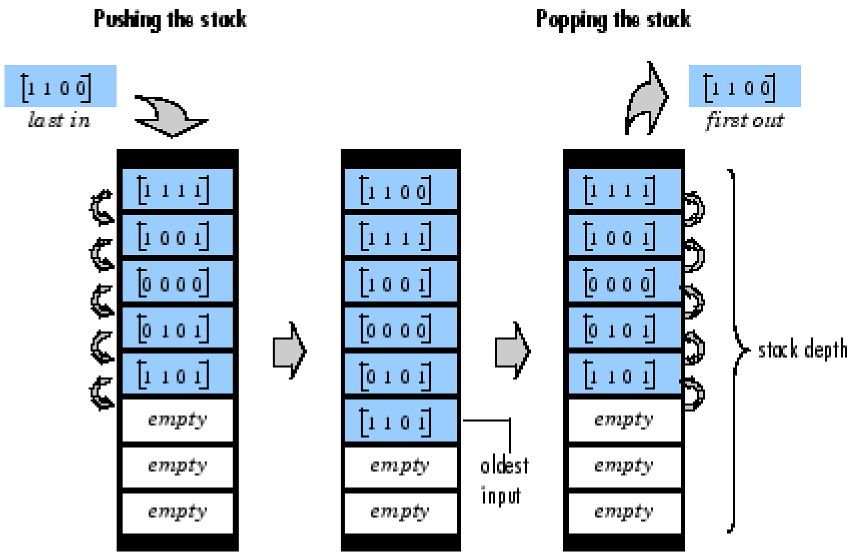
Signal Management / Buffers

`dspbuff3`

Description

The Stack block stores a sequence of input samples in a last in, first out (LIFO) register. The register capacity is set by the **Stack depth** parameter, and inputs can be scalars, vectors, or matrices.

The block *pushes* the input at the `In` port onto the top of the stack when a trigger event is received at the `Push` port. When a trigger event is received at the `Pop` port, the block *pops* the top element off the stack and holds the `Out` port at that value. The last input to be pushed onto the stack is always the first to be popped off.



A trigger event at the optional `Rst` port empties the stack contents. When you select **Clear output port on reset**, then a trigger event at the `Rst` port empties the stack *and* sets the value at the `Out` port to zero. This setting also applies when a disabled subsystem containing the Stack block is reenabled; the `Out` port value is only reset to zero in this case when you select **Clear output port on reset**.

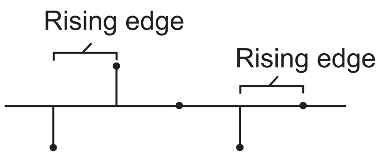
When two or more of the control input ports are triggered at the same time step, the operations are executed in the following order:

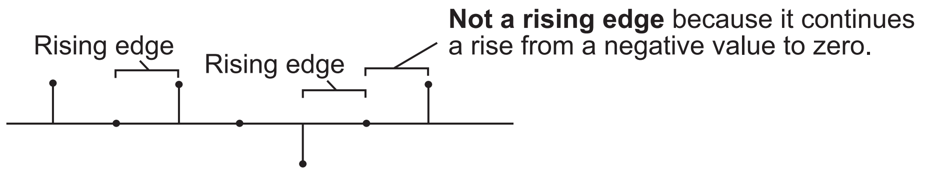
1. `Rst`
2. `Push`
3. `Pop`

The rate of the trigger signal must be the same as the rate of the data signal input. You specify the triggering event for the `Push`, `Pop`, and `Rst` ports in the **Trigger type** pop-up menu:

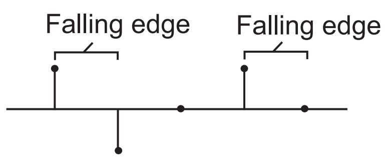
- `Rising edge` — Triggers execution of the block when the trigger input does one of the following:
  - Rises from a negative value to a positive value or zero

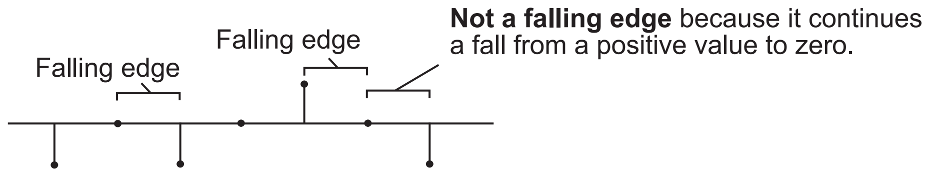
- Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)





- `Falling edge` — Triggers execution of the block when the trigger input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)





- `Either edge` — Triggers execution of the block when the trigger input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers execution of the block at each sample time that the trigger input is not zero.

Note: If your model contains any referenced models that use a Stack block with the **Push full stack** parameter set to `Dynamic reallocation`, you cannot simulate your top-level model in Simulink Accelerator mode.

The **Push full stack** parameter specifies the block's behavior when a trigger is received at the `Push` port but the register is full. The **Pop empty stack** parameter specifies the block's behavior when a trigger is received at the `Pop` port but the register is empty. The following options are available for both cases:

- `Ignore` — Ignore the trigger event, and continue the simulation.
- `Warning` — Ignore the trigger event, but display a warning message in the MATLAB command window.
- `Error` — Display an error dialog box and terminate the simulation.

Note The **Push full stack** and **Pop empty stack** parameters are diagnostic parameters. Like all diagnostic parameters on the Configuration Parameters dialog box, they are set to `Ignore` in the code generated for this block by Simulink Coder code generation software.

The **Push full stack** parameter additionally offers the **Dynamic reallocation** option, which dynamically resizes the register to accept as many additional inputs as memory permits. To find out how many elements are on the stack at a given time, enable the `Num` output port by selecting the **Show number of stack entries port** parameter.

Note: When `Dynamic reallocation` is selected, the **System target file** parameter on the **Code Generation** pane of the Model Configuration Parameters dialog box must be set to `grt_malloc.tlc — Generic Real-Time Target with dynamic memory allocation`.

Examples

Example 1

The table below illustrates the Stack block's operation for a **Stack depth** of 4, **Trigger type** of `Either edge`, and **Clear output port on reset** enabled. Because the block triggers on both rising and falling edges in this example, each transition from `1` to `0` or `0` to `1` in the `Push`, `Pop`, and `Rst` columns below represents a distinct trigger event. A `1` in the `Empty` column indicates an empty buffer, while a `1` in the `Full` column indicates a full buffer.

In	Push	Pop	Rst	Stack	Out	Empty	Full	Num
1	0	0	0	top bottom	0	1	0	0
2	1	0	0	top 2 bottom	0	0	0	1
3	0	0	0	top 3 2 bottom	0	0	0	2
4	1	0	0	top 4 3 2 bottom	0	0	0	3
5	0	0	0	top 5 4 3 2 bottom	0	0	1	4
6	0	1	0	top 4 3 2 bottom	5	0	0	3
7	0	0	0	top 3 2 bottom	4	0	0	2
8	0	1	0	top 2 bottom	3	0	0	1
9	0	0	0	top bottom	2	1	0	0
10	1	0	0	top 10 bottom	2	0	0	1
11	0	0	0	top 11 10 bottom	2	0	0	2
12	1	0	1	top 12 bottom	0	0	0	1

Note that at the last step shown, the `Push` and `Rst` ports are triggered simultaneously. The `Rst` trigger takes precedence, and the stack is first cleared and then pushed.

Example 2

The dspqdemo example provides an example of the related Queue block.

Parameters

Stack depth

The number of entries that the LIFO register can hold.

Trigger type

The type of event that triggers the block's execution. The rate of the trigger signal must be the same as the rate of the data signal input.

Push full stack

Response to a trigger received at the `Push` port when the register is full. Inputs to this port must have the same built-in data type as inputs to the `Pop` and `Rst` input ports.

When `Dynamic reallocation` is selected, the **System target file** parameter on the **Code Generation** pane of the Model Configuration Parameters dialog box must be set to `grt_malloc.tlc — Generic Real-Time Target with dynamic memory allocation`.

Pop empty stack

Response to a trigger received at the `Pop` port when the register is empty. Inputs to this port must have the same built-in data type as inputs to the `Push` and `Rst` input ports.

Show empty stack indicator port

Enable the `Empty` output port, which is high (`1`) when the stack is empty, and low (`0`) otherwise.

Show full stack indicator port

Enable the `Full` output port, which is high (`1`) when the stack is full, and low (`0`) otherwise. The `Full` port remains low when you select **Dynamic reallocation** from the **Push full stack** parameter.

Show number of stack entries port

Enable the `Num` output port, which tracks the number of entries currently on the stack. When inputs to the `In` port are double-precision values, the outputs from the `Num` port are double-precision values. Otherwise, the outputs from the `Num` port are 32-bit unsigned integer values.

Show reset port to clear internal stack buffer

Enable the `Rst` input port, which empties the stack when the trigger specified by the **Trigger type** is received. Inputs to this port must have the same built-in data type as inputs to the `Push` and `Pop` input ports.

Clear output port on reset

Reset the `Out` port to zero (in addition to clearing the stack) when a trigger is received at the `Rst` input port.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Push	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Inputs to this port must have the same built-in data type as inputs to the Pop and Rst input ports
Pop	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Inputs to this port must have the same built-in data type as inputs to the Push and Rst input ports.
Rst	• Double-precision floating point • Single-precision floating point • Boolean

Port	Supported Data Types
	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Inputs to this port must have the same built-in data type as inputs to the Push and Pop input ports.
Out	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Empty	• Double-precision floating point • Boolean
Full	• Double-precision floating point • Boolean
Num	• Double-precision floating point The block outputs a double-precision floating-point value at this port when the data type of the In port is double-precision floating-point. • 32-bit unsigned integers The block outputs a 32-bit unsigned integer value at this port when the data type of the In port is anything other than double-precision floating-point.

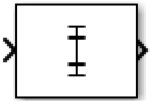
See Also

Buffer	DSP System Toolbox
Delay Line	DSP System Toolbox
Queue	DSP System Toolbox

Introduced before R2006a

Standard Deviation

Standard deviation of input or sequence of inputs
Library: DSP System Toolbox / Statistics



Description

The Standard Deviation block computes the standard deviation of each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the standard deviation of the entire input. You can specify the dimension using the **Find the standard deviation value over** parameter. The Standard Deviation block can also track the standard deviation in a sequence of inputs over a period of time. To track the standard deviation in a sequence of inputs, select the **Running standard deviation** parameter.

Note: The **Running** mode in the Standard Deviation block will be removed in a future release. To compute the running standard deviation in Simulink, use the `Moving Standard Deviation` block instead.

Ports

Input

`In` — Data input

vector | matrix | *N*-D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs.

This port is unnamed until you select the **Running standard deviation** parameter and set the **Reset port** parameter to any option other than `None`.

Data Types: `single` | `double`

Complex Number Support: Yes

`Rst` — Reset port
scalar

Specify the reset event that causes the block to reset the running standard deviation. The sample time of the **Rst** input must be a positive integer multiple of the input sample time.

Dependencies

To enable this port, select the **Running standard deviation** parameter and set the **Reset port** parameter to any option other than `None`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Port_1` — Standard deviation along the specified dimension
scalar | vector | matrix | *N*-D array

The data type of the output matches the data type of the input.

When you do not select the **Running standard deviation** parameter, the block computes the standard deviation in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the standard deviation of the entire input at each individual sample time. Each element in the output array `y` is the standard deviation of the corresponding column, row, or entire input. The output array `y` depends on the setting of the **Find the standard deviation value over** parameter. Consider a three-dimensional input signal of size *M*-by-*N*-by-*P*. When you set **Find the standard deviation value over** to:

- `Entire input` — The output at each sample time is a scalar that contains the standard deviation of the *M*-by-*N*-by-*P* input matrix.
- `Each row` — The output at each sample time consists of an *M*-by-1-by-*P* array, where each element contains the standard deviation of each vector over the second dimension of the input. For an *M*-by-*N* matrix input, the output at each sample time is an *M*-by-1 column vector.

- `Each column` — The output at each sample time consists of a 1-by-N-by-P array, where each element contains the standard deviation of each vector over the first dimension of the input. For an M-by-N matrix input, the output at each sample time is a 1-by-N row vector.

  In this mode, the block treats length-M unoriented vector inputs as M-by-1 column vectors.
- `Specified dimension` — The output at each sample time depends on the value of the **Dimension** parameter. If you set the **Dimension** to `1`, the output is the same as when you select `Each column`. If you set the **Dimension** to `2`, the output is the same as when you select `Each row`. If you set the **Dimension** to `3`, the output at each sample time is an M-by-N matrix containing the standard deviation of each vector over the third dimension of the input.

When you select **Running standard deviation**, the block tracks the standard deviation of each channel in a time sequence of inputs. In this mode, you must also specify a value for the **Input processing** parameter.

- `Elements as channels (sample based)` — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size M-by-N-by-P, the block outputs an M-by-N-by-P array. Each element y_{ijk} of the output contains the standard deviation of the element u_{ijk} for all inputs since the last reset.

  When a reset event occurs, the running standard deviation y_{ijk} in the current frame is reset to the element u_{ijk}.
- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel. This option does not support input signals with more than two dimensions. For a two-dimensional input signal of size M-by-N, the block outputs an M-by-N matrix. Each element y_{ij} of the output contains the standard deviation of the elements in the jth column of all inputs since the last reset, up to and including the element u_{ij} of the current input.

  When a reset event occurs, the running standard deviation for each channel becomes the standard deviation of all the samples in the current input frame, up to and including the current input sample.

Data Types: `single` | `double`

Parameters

Main Tab

`Running standard deviation` — Option to select running standard deviation

off (default) | on

When you select the **Running standard deviation** parameter, the block tracks the standard deviation value of each channel in a time sequence of inputs.

`Find the standard deviation value over` — Dimension over which the block computes the standard deviation

`Each column` (default) | `Entire input` | `Each row` | `Specified dimension`

- `Each column` — The block outputs the standard deviation over each column.
- `Each row` — The block outputs the standard deviation over each row.
- `Entire input` — The block outputs the standard deviation over the entire input.
- `Specified dimension` — The block outputs the standard deviation over the dimension, specified in the **Dimension** parameter.

Dependencies

To enable this parameter, clear the **Running standard deviation** parameter.

`Dimension` — Custom dimension

`1` (default) | scalar

Specify the dimension (one-based value) of the input signal over which the standard deviation is computed. The value of this parameter must be greater than 0 and less than the number of dimensions in the input signal.

Dependencies

To enable this parameter, set **Find the standard deviation value over** to `Specified dimension`.

`Input processing` — Method to process the input in running mode
`Columns as channels (frame based)` (default) | `Elements as channels (sample based)`

- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel. This option does not support input signals with more than two dimensions. For a two-dimensional input signal of size *M*-by-*N*, the block outputs an *M*-by-*N* matrix. Each element y_{ij} of the output contains the standard deviation of the elements in the *j*th column of all inputs since the last reset, up to and including the element u_{ij} of the current input.

  When a reset event occurs, the running standard deviation for each channel becomes the standard deviation of all the samples in the current input frame, up to and including the current input sample.
- `Elements as channels (sample based)` — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size *M*-by-*N*-by-*P*, the block outputs an *M*-by-*N*-by-*P* array. Each element y_{ijk} of the output contains the standard deviation of the element u_{ijk} for all inputs since the last reset.

  When a reset event occurs, the running standard deviation y_{ijk} in the current frame is reset to the element u_{ijk}.

  Variable-Size Inputs

  When your inputs are of variable size, and you select the **Running standard deviation** parameter, then:

  - If you set the **Input processing** parameter to `Elements as channels (sample based)`, the state is reset.
  - If you set the **Input processing** parameter to `Columns as channels (frame based)`, then:

    - When the input size difference is in the number of channels (number of columns), the state is reset.
    - When the input size difference is in the length of channels (number of rows), no reset occurs and the running operation is carried out as usual.

Dependencies

To enable this parameter, select the **Running standard deviation** parameter.

`Reset port` — Reset event

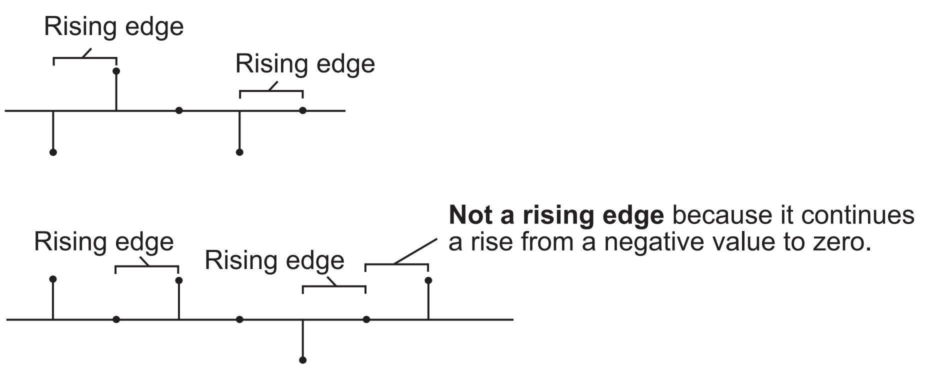
`None` (default) | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero sample`

The block resets the running standard deviation whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

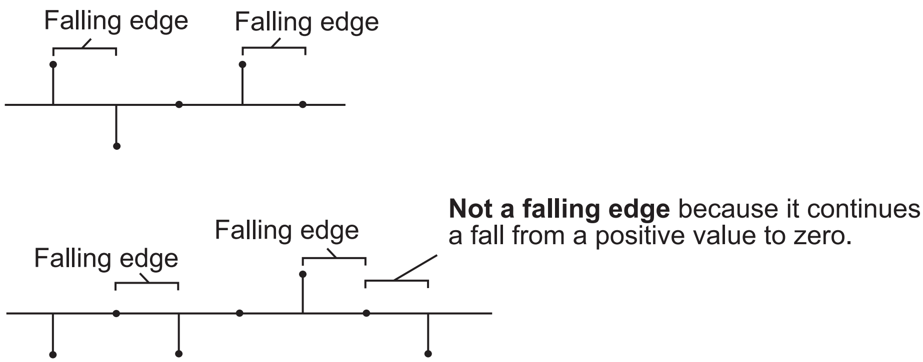
When a reset event occurs while the **Input processing** parameter is set to `Elements as channels (sample based)`, the running standard deviation for each channel is initialized to the value in the corresponding channel of the current input. Similarly, when the **Input processing** parameter is set to `Columns as channels (frame based)`, the running standard deviation for each channel becomes the standard deviation of all the samples in the current input frame, up to and including the current input sample.

Use this parameter to specify the reset event.

- `None` — Disables the **Rst** port.
- `Rising edge` — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to either a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.



- `Falling edge` — Triggers a reset operation when the **Rst** input does one of the following:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.



- `Either edge` — Triggers a reset operation when the **Rst** input is a `Rising edge` or `Falling edge`.
- `Non-zero sample` — Triggers a reset operation at each sample time, when the **Rst** input is not zero.

Note: When running simulations in the Simulink multitasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

Dependencies

To enable this parameter, select the **Running standard deviation** parameter.

Definitions

Standard Deviation

The standard deviation of a discrete-time signal is the square root of the variance of the signal.

Standard deviation gives a measure of deviation of the signal from its mean value.

For purely real or imaginary input, u, of size M-by-N, the standard deviation is given by the following equation:

$$y = \sigma = \sqrt{\frac{\sum_{i=1}^{M}\sum_{j=1}^{N}\left|u_{ij}\right|^2 - \frac{\left|\sum_{i=1}^{M}\sum_{j=1}^{N}u_{ij}\right|^2}{M*N}}{M*N-1}}$$

- u_{ij} is the input data element at indices i, j.
- M is the length of the jth column.
- N is the number of columns.

For complex inputs, the standard deviation is given by the following equation:

$$\sigma = \sqrt{\sigma_{\mathrm{Re}}{}^2 + \sigma_{\mathrm{Im}}{}^2}$$

- $\sigma_{Re}{}^2$ is the variance of the real part of the complex input.
- $\sigma_{Im}{}^2$ is the variance of the imaginary part of the complex input.

Algorithms

Standard Deviation

When you clear the **Running standard deviation** parameter in the block and specify a dimension, the block produces results identical to the MATLAB `std` function, when it is called as `y = std(u,0,D)`.

- `u` is the data input.
- `D` is the dimension.
- `y` is the standard deviation along the specified dimension.

The standard deviation along the entire input is identical to calling the `std` function as `y = std(u(:))`.

For a complex input signal, the standard deviation is the square root of the sum of the variances of the real and imaginary parts.

$$\sigma = \sqrt{\sigma_{\mathrm{Re}}{}^2 + \sigma_{\mathrm{Im}}{}^2}$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Functions
`std`

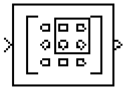
System Objects
dsp.StandardDeviation | dsp.MovingStandardDeviation

Blocks
`Moving Standard Deviation`

Introduced before R2006a

Submatrix

Select subset of elements (submatrix) from matrix input



Library

- Math Functions / Matrices and Linear Algebra / Matrix Operations

  `dspmtrx3`
- Signal Management / Indexing

  `dspindex`

Description

The Submatrix block extracts a contiguous submatrix from the *M*-by-*N* input matrix `u`. The block treats length-*M* unoriented vector input as an *M*-by-`1` matrix. The **Row span** parameter provides three options for specifying the range of rows in `u` to be retained in submatrix output `y`:

- `All rows`

  Specifies that `y` contains all *M* rows of `u`.
- `One row`

  Specifies that `y` contains only one row from `u`. The **Row** parameter (described below) is enabled to allow selection of the desired row.
- `Range of rows`

  Specifies that `y` contains one or more rows from `u`. The **Starting row** and **Ending row** parameters (described below) are enabled to allow selection of the desired range of rows.

The **Column span** parameter contains a corresponding set of three options for specifying the range of columns in `u` to be retained in submatrix `y`: `All columns`, `One column`, or `Range of columns`. The `One column` option enables the **Column** parameter, and `Range of columns` options enable the **Starting column** and **Ending column** parameters.

Range Specification Options

When you select `One row` or `Range of rows` from the **Row span** parameter, you specify the desired row or range of rows in the **Row** parameter, or the **Starting row** and **Ending row** parameters. Similarly, when you select `One column` or `Range of columns` from the **Column span** parameter, you specify the desired column or range of columns in the **Column** parameter, or the **Starting column** and **Ending column** parameters.

The **Row**, **Column**, **Starting row** or **Starting column** can be specified in six ways:

- `First`

  For rows, this specifies that the first row of `u` should be used as the first row of `y`. When all columns are to be included, this is equivalent to `y(1,:) = u(1,:)`.

  For columns, this specifies that the first column of `u` should be used as the first column of `y`. When all rows are to be included, this is equivalent to `y(:,1) = u(:,1)`.
- `Index`

  For rows, this specifies that the row of `u`, `firstrow`, forward-indexed by the **Row index** parameter or the **Starting row index** parameter, should be used as the first row of `y`. When all columns are to be included, this is equivalent to `y(1,:) = u(firstrow,:)`.

  For columns, this specifies that the column of `u`, forward-indexed by the **Column index** parameter or the **Starting column index** parameter, `firstcol`, should be used as the first column of `y`. When all rows are to be included, this is equivalent to `y(:,1) = u(:,firstcol)`.
- `Offset from last`

  For rows, this specifies that the row of `u` offset from row *M* by the **Row offset** or **Starting row offset** parameter, `firstrow`, should be used as the first row of `y`. When all columns are to be included, this is equivalent to `y(1,:) = u(M-firstrow,:)`.

For columns, this specifies that the column of `u` offset from column *N* by the **Column offset** or **Starting column offset** parameter, `firstcol`, should be used as the first column of `y`. When all rows are to be included, this is equivalent to `y(:,1) = u(:,N-firstcol)`.

- `Last`

  For rows, this specifies that the last row of `u` should be used as the only row of `y`. When all columns are to be included, this is equivalent to `y = u(M,:)`.

  For columns, this specifies that the last column of `u` should be used as the only column of `y`. When all rows are to be included, this is equivalent to `y = u(:,N)`.

- `Offset from middle`

  When you select this option, the block selects the first row or column of the output `y` by adding the specified offset to the middle row or column of the input `u`. When the number, *X*, of input rows or columns is even, the block defines the middle one as *X*/2 +1. When the number of input rows or columns is odd, the block defines the middle one as **`ceil`**(*X*/2).

  When all columns are to be included, the following code defines the starting row: `y(1,:) = u(MiddleRow+Offset,:)`, where `Offset` is the value of the **Row offset** or **Starting row offset** parameter. When all rows are to be included, the following code defines the starting column: `y(1,:) = u(:,MiddleColumn+Offset)`, where `Offset` is the value of the **Column offset** or **Starting column offset** parameter.

- `Middle`

  When you select this option, the block uses the middle row or column of the input `u` as the first row or column of the output `y`. When the number, *X*, of input rows or columns is even, the block defines the middle one as *X*/2 +1. When the number of input rows or columns is odd, the block defines the middle one as **`ceil`**(*X*/2).

  When all columns are to be included, the following code defines the starting row: `y = u(MiddleRow,:)`. When all rows are to be included, the following code defines the starting column: `y = u(:,MiddleColumn)`.

The **Ending row** or **Ending column** can similarly be specified in five ways:

- `Index`

For rows, this specifies that the row of `u` forward-indexed by the **Ending row index** parameter, `lastrow`, should be used as the last row of `y`. When all columns are to be included, this is equivalent to `y(end,:) = u(lastrow,:)`.

For columns, this specifies that the column of `u` forward-indexed by the **Ending column index** parameter, `lastcol`, should be used as the last column of `y`. When all rows are to be included, this is equivalent to `y(:,end) = u(:,lastcol)`.

- `Offset from last`

  For rows, this specifies that the row of `u` offset from row *M* by the **Ending row offset** parameter, `lastrow`, should be used as the last row of `y`. When all columns are to be included, this is equivalent to `y(end,:) = u(M-lastrow,:)`.

  For columns, this specifies that the column of `u` offset from column *N* by the **Ending column offset** parameter, `lastcol`, should be used as the last column of `y`. When all rows are to be included, this is equivalent to `y(:,end) = u(:,N-lastcol)`.

- `Last`

  For rows, this specifies that the last row of `u` should be used as the last row of `y`. When all columns are to be included, this is equivalent to `y(end,:) = u(M,:)`.

  For columns, this specifies that the last column of `u` should be used as the last column of `y`. When all rows are to be included, this is equivalent to `y(:,end) = u(:,N)`.

- `Offset from middle`

  When you select this option, the block selects the last row or column of the output `y` by adding the specified offset to the middle row or column of the input `u`. When the number, *X*, of input rows or columns is even, the block defines the middle one as *X*/2 +1. When the number of input rows or columns is odd, the block defines the middle one as **`ceil`**(*X*/2).

  When all columns are to be included, the following code defines the ending row: `y(end,:) = u(MiddleRow+Offset,:)`, where `Offset` is the value of the **Ending row offset** parameter. When all rows are to be included, the following code defines the ending column: `y(:,end) = u(:,MiddleColumn+Offset)`, where `Offset` is the value of the **Ending column offset** parameter.

- `Middle`

When you select this option, the block uses the middle row or column of the input `u` as the last row or column of the output `y`. When the number, X, of input rows or columns is even, the block defines the middle one as $X/2$ +1. When the number of input rows or columns is odd, the block defines the middle one as **`ceil`**($X/2$).

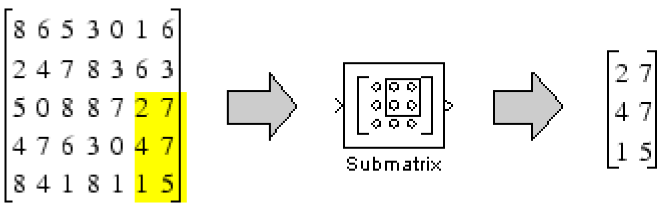
When all columns are to be included, the following code defines the ending row: `y(end,:) = u(MiddleRow,:)`. When all rows are to be included, the following code defines the ending column: `y(:,end) = u(:,MiddleColumn)`.

This block supports Simulink virtual buses.

Examples

The ex_submatrix_ref model uses a Submatrix block to extract a 3-by-2 submatrix from the lower-right corner of a 5-by-7 input matrix.

The following figure shows the operation of the Submatrix block with a 5-by-7 input matrix of random integer elements, `randi([0 9],5,7)`.



There are often several possible parameter combinations that you can use to select the *same* submatrix from the input. For example, in the case of a 5-by-7 input matrix, instead of specifying `Last` for **Ending column**, you could select the same submatrix by specifying

- **Ending column** = `Index`
- **Ending column index** = `7`

Parameters

Row span

The range of input rows to be retained in the output. Options are `All rows`, `One row`, or `Range of rows`.

Row/Starting row

The input row to be used as the first row of the output. **Row** is enabled when you select `One row` from **Row span**, and **Starting row** when you select `Range of rows` from **Row span**.

Row index/Starting row index

The index of the input row to be used as the first row of the output. **Row index** is enabled when you select `Index` from Row, and **Starting row index** when you select `Index` from **Starting row**.

Row offset/Starting row offset

The offset of the input row to be used as the first row of the output. **Row offset** is enabled when you select `Offset from middle` or `Offset from last` from **Row**, and Starting row offset is enabled when you select `Offset from middle` or `Offset from last` from **Starting row**.

Ending row

The input row to be used as the last row of the output. This parameter is enabled when you select `Range of rows` from **Row span** and you select any option but `Last` from **Starting row**.

Ending row index

The index of the input row to be used as the last row of the output. This parameter is enabled when you select `Index` from **Ending row**.

Ending row offset

The offset of the input row to be used as the last row of the output. This parameter is enabled when you select `Offset from middle` or `Offset from last` from **Ending row**.

Column span

The range of input columns to be retained in the output. Options are `All columns`, `One column`, or `Range of columns`.

Column/Starting column

The input column to be used as the first column of the output. **Column** is enabled when you select `One column` from **Column span**, and **Starting column** is enabled when you select `Range of columns` from **Column span**.

Column index/Starting column index

The index of the input column to be used as the first column of the output. **Column index** is enabled when you select `Index` from Column, and **Starting column index** is enabled when you select `Index` from **Starting column**.

Column offset/Starting column offset

The offset of the input column to be used as the first column of the output. **Column offset** is enabled when you select `Offset from middle` or `Offset from last` from Column. **Starting column offset** is enabled when you select `Offset from middle` or `Offset from last` from **Starting column**.

Ending column

The input column to be used as the last column of the output. This parameter is enabled when you select `Range of columns` from **Column span** and you select any option but `Last` from **Starting column**.

Ending column index

The index of the input column to be used as the last column of the output. This parameter is enabled when you select `Index` from **Ending column**.

Ending column offset

The offset of the input column to be used as the last column of the output. This parameter is enabled when you select `Offset from middle` or `Offset from last` from **Ending column**.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned)

Port	Supported Data Types
	• Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated

See Also

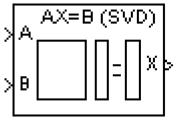
Reshape	Simulink
Selector	Simulink
Variable Selector	DSP System Toolbox
`reshape`	MATLAB

See "Split Multichannel Signals into Several Multichannel Signals" for related information.

Introduced before R2006a

SVD Solver

Solve $AX=B$ using singular value decomposition



Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

`dspsolvers`

Description

The SVD Solver block solves the linear system $AX=B$, which can be overdetermined, underdetermined, or exactly determined. The system is solved by applying singular value decomposition (SVD) factorization to the M-by-N matrix A, at the A port. The input to the B port is the right side M-by-L matrix, B. The block treats length-M unoriented vector input as an M-by-1 matrix.

The output at the X port is the N-by-L matrix, X. X is chosen to minimize the sum of the squares of the elements of B-AX (the residual). When B is a vector, this solution minimizes the vector 2-norm of the residual. When B is a matrix, this solution minimizes the matrix Frobenius norm of the residual. In this case, the columns of X are the solutions to the L corresponding systems $AX_k=B_k$, where B_k is the kth column of B, and X_k is the kth column of X.

X is known as the minimum-norm-residual solution to $AX=B$. The minimum-norm-residual solution is unique for overdetermined and exactly determined linear systems, but it is not unique for underdetermined linear systems. Thus when the SVD Solver block is applied to an underdetermined system, the output X is chosen such that the number of nonzero entries in X is minimized.

Parameters

Show error status port

Select to enable the E output port, which reports a failure to converge. The possible values you can receive on the port are:

- `0` — The singular value decomposition calculation converges.
- `1` — The singular value decomposition calculation does not converge.

If the singular value decomposition calculation fails to converge, the output at port X is an undefined matrix of the correct size.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point
B	• Double-precision floating point • Single-precision floating point
X	• Double-precision floating point • Single-precision floating point
E	• Boolean

See Also

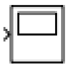
`Autocorrelation LPC`	DSP System Toolbox
`Cholesky Solver`	DSP System Toolbox
`LDL Solver`	DSP System Toolbox
`Levinson-Durbin`	DSP System Toolbox
`LU Inverse`	DSP System Toolbox
`Pseudoinverse`	DSP System Toolbox
`QR Solver`	DSP System Toolbox
`Singular Value Decomposition`	DSP System Toolbox

See “Linear System Solvers” for related information.

Introduced before R2006a

Time Scope

Display time-domain signals



Library

Sinks

Description

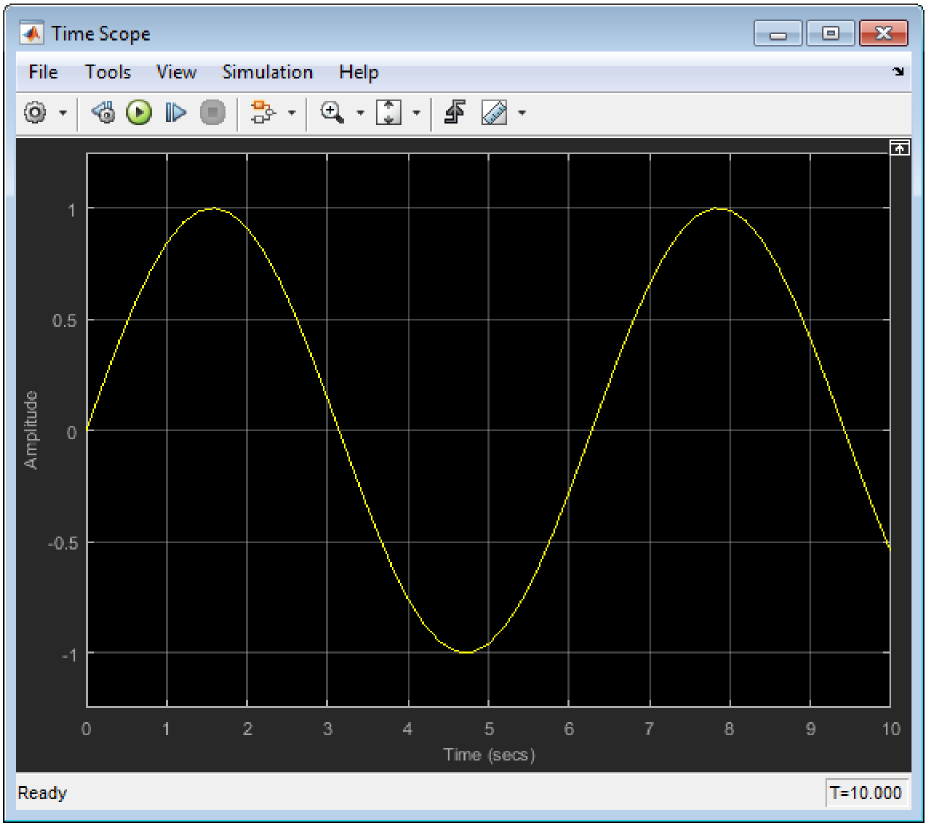
The Time Scope block displays signals in the time domain. The Time Scope block accepts input signals with the following characteristics:

- Continuous or discrete sample time
- Real- or complex-valued
- Fixed or variable size dimensions
- Floating- or fixed-point data type
- N-dimensional
- Simulink enumerations

To determine whether to use the DSP System Toolbox Time Scope or the Simulink Scope, see “Simulink Scope Versus DSP System Toolbox Time Scope” (Simulink).

For information on the simulation modes supported by the scope and limitations on these modes, see “Supported Simulation Modes” on page 2-1718. You can use the Time Scope block inside any subsystem or conditional subsystem. See “Conditional Subsystems” (Simulink) for more information.

To improve the performance of models that contain scopes, select **Analysis** > **Performance Tools** > **Performance Advisor** in the Simulink Editor or type `performanceadvisor('your_model_name')` at the command line. See “Automated Performance Optimization” (Simulink) for more information. A quick change you can make to improve performance is to close and comment out scope blocks using the right-click menu. Another performance improvement is to use **Scope Simulation** > **Reduce Updates to Improve Performance**.



The following sections describe the features of the Time Scope:

- "Signal Display" on page 2-1650
- "Measurements Panels" on page 2-1652
- "Configuration Properties Dialog Box" on page 2-1683
- "Style Dialog Box" on page 2-1698
- "Stepping Options" on page 2-1701
- "Tools—Axes Scaling Properties" on page 2-1702
- "Examples" on page 2-1705
- "Supported Data Types" on page 2-1718
- "Supported Simulation Modes" on page 2-1718

Note: For information about the Time Scope System object, see dsp.TimeScope.

For information on controlling the Time Scope from the command line, see “Control Scopes Programmatically” (Simulink).

Signal Display

Time Scope uses the simulation start time and stop time to determine the default time range. To define the length of simulation time for which the Time Scope displays data. see **Time span** and **Time display offset** parameters in the “Configuration Properties Dialog Box” on page 2-1683.

By default, the scope updates the displays periodically at a rate not exceeding 20 hertz. If you would like the scope to update on every simulation time step, you can disable the **Reduce Updates to Improve Performance** option. However, as a recommended practice, leave this option enabled because doing so can significantly improve the speed of the simulation.

In the Time Scope menu, clear the **Simulation** > **Reduce Updates to Improve Performance** check box. You can also use **Ctrl+R** to toggle this setting.

The scope shows the following status information.

Note: To prevent the scope from opening when you run your model, right-click the scope icon and select `Comment Out`. If the scope is already open, and you comment it out in the model, the scope displays, “No data can be shown because this scope is commented out.” Select `Uncomment` to turn the scope back on.

The scope shows the following plot labels and status information.

- *Minimum time-axis limit* — The Time Scope sets the minimum time-axis limit using the value of the **Time display offset** parameter on the **Time** tab of the Visuals—Time Domain Properties dialog box. If you specify a vector of values for the **Time display offset** parameter, the scope uses the smallest of those values to set the minimum time-axis limit.
- *Maximum time-axis limit* — The Time Scope sets the maximum time-axis limit by summing the value of **Time display offset** parameter with the value of the

Time span parameter. If you specify a vector of values for the **Time display offset** parameter, the scope sets the maximum time-axis limit by summing the largest of those values with the value of the **Time span** parameter.

You can choose to hide or display both the scope menus and the toolbar using the or button in the upper right corner of the scope. To hide or display the toolbar separately, use **View** > **Toolbar** in the scope menu.

You can choose to hide or display the entire *Status Bar* with **View** > **Status Bar** in the scope menu. The following items are in the Status Bar:

- *Simulation status* — Provides the status of the model simulation. The status can be one of the following conditions:

  - `Initializing`
  - `Ready`
  - `Running`
  - `Paused`

  The *Simulation status* is part of the *Status Bar* in the Time Scope window. You can choose to hide or display the entire *Status Bar*. From the Time Scope menu, select **View** > **Status Bar**.
- *Time offset* — The `Offset` value helps you determine the simulation times for which the scope is displaying data. The value is always in the range `0`≤ `Offset`≤ *Simulation time*. If the time offset is 0, the Scope does not display the `Offset` status field. Add the Time offset to the fixed time span values on the time-axis to get the overall simulation time.

  For example, if you set the **Time span** to `20` seconds, and you see an `Offset` of `0 (secs)` on the scope window. This value indicates that the scope is displaying data for the first `0` to `20` seconds of simulation time. If the **Offset** changes to `20 (secs)`, the scope displays data for simulation times from `20` seconds to `40` seconds. The scope continues to update the `Offset` value until the simulation is complete.
- *Simulation time* — When the model is running or simulation has been paused, the scope displays the current simulation time. This time is the amount of time that the Time Scope has spent processing the input. If the model simulation completes or is stopped, the scope displays the time at which the simulation stopped. You can choose to hide or display the entire *Status Bar*. From the Time Scope menu, select **View** > **Status Bar**.

Note: In some situations, the Time Scope block simulation time can be different from the Simulink simulation time. For multirate input signals, which have different sample times, and input signals originating from conditionally executed subsystems, such as Triggered and Enabled subsystems, separate Time Scope blocks can report different simulation times. The Time Scope block reports the simulation time as the time corresponding to the last point in the display.

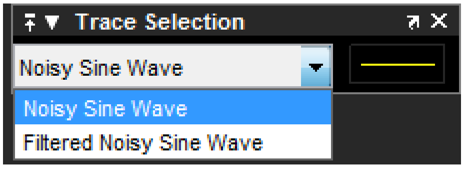
To show and adjust multiple displays in your scope, see "Display Multiple Signals in the Time Scope".

Measurements Panels

The Measurements panels are the panels that appear to the right side of the Time Scope GUI. Select the desired panels using **Tools** > **Measurements** in the scope toolbar or click the icons in the measurements dropdown.

Trace Selection Panel

When you use the scope to view multiple signals, the Trace Selection panel appears if you have more than one signal displayed and you click any of the other Measurements panels. The Measurements panels display information about only the signal chosen in this panel. Choose the signal name for which you would like to display time domain measurements. See the following figure.



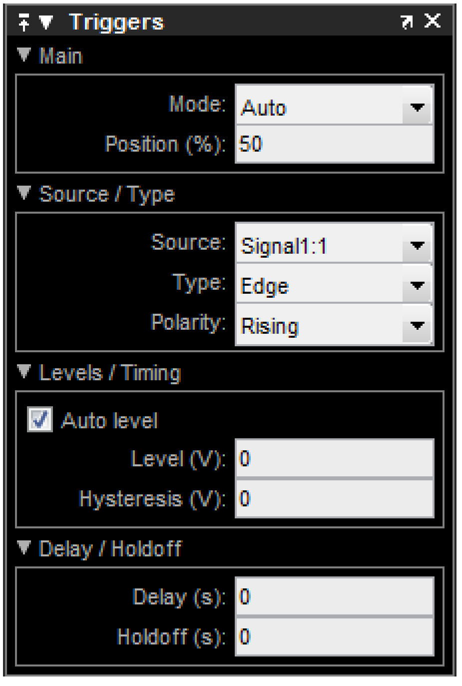
You can choose to hide or display the **Trace Selection** panel. In the Scope menu, select **Tools** > **Measurements** > **Trace Selection**.

Triggers Panel

The **Triggers** panel allows you to pause the display only when certain events occur. You can use the Triggers panel when you want to align or search for interesting events. You can configure triggers to both select and align specific regions of interest in the display area of the scope. Triggers work across multiple displays.

Note: If triggers are on and your model is running, you cannot pan the signal display. Panning, when triggers are on, is enabled only when your model is paused or stopped.

To open the Triggers panel, click the Triggers button ( ) or select **Tools** > **Triggers** in the scope menu.



When the **Triggers** panel is displayed, triangle pointers appear at the top and right side of the axes on each display. These markers indicate the time position ( ) and level ( ) at the event. The color of the markers corresponds to the color of the source signal.

Note: Triggers are governed by the **Time Span** setting of the scope. The time span is used for autoleveling and determines how much data to show in the display. The scope does not display an event until an entire time span is viewable in the display. To prevent data from being shown twice, the scope suppresses the alignment of recurring events until an entire time span has elapsed since the previous update.

The **Main** pane lets you choose how often the display updates and in what position the trigger indicator appears.

- `Mode` — Define how often the display updates.
  - `Auto` — The scope aligns and displays data from the latest trigger event. Data between trigger events is displayed. If no trigger event is found after an entire time span has elapsed, the scope displays that time span of data. Use this mode to see your data and have it align whenever a trigger event occurs.
  - `Normal` — The scope aligns and displays data only from the last trigger event in the time span and freezes the display. Data between trigger events is not displayed. Use this mode to search for infrequently occurring events in your data.
  - `Once` — The scope aligns and displays data on the first encountered trigger event and freezes the display. The scope ignores subsequent data until you press the **Rearm** button. If no trigger event is found, no data is shown in the display.
  - `Off` — Triggering is disabled. This setting is equivalent to hiding the **Triggers** panel.

  If mode is set to either `Normal` or `Once` and no trigger event occurs, the display remains blank. Set **Mode** to `Auto` to display all signal data, in addition to trigger events.
- **Position (%)** — Specify, as a percentage of the total time span within the active display, the horizontal position in which the trigger indicator appears. A position value of `0` corresponds to the minimum time-axis value at the far-left side of the display. A position value of `100` corresponds to the maximum time-axis value at the far-right side of the display. Drag the trigger position indicator to the left or right to adjust its position.

The **Source/Type** pane lets you choose the source of the trigger and the type of events on which to stop.

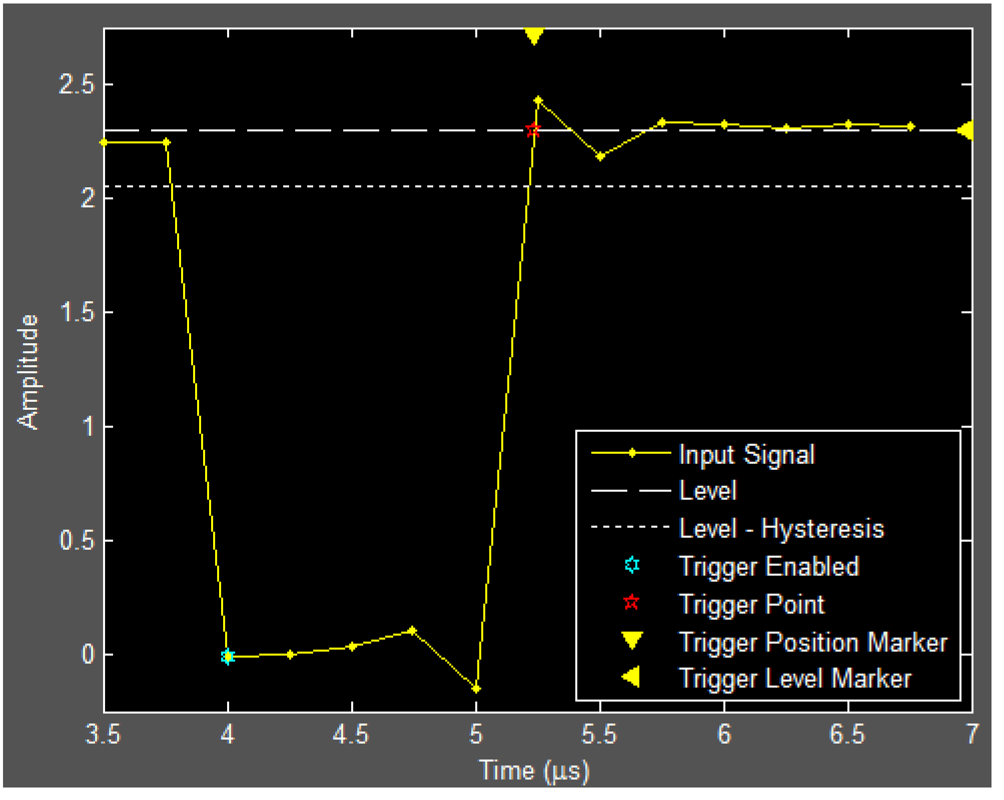
- **Source** — Assign the trigger source to a particular channel. If you are viewing a magnitude/phase plot, you can trigger off the magnitude or the phase. If you are not viewing the magnitude/phase plot, you can trigger off the real or imaginary data. If the input signal has multiple channels, the scope assigns an index number to identify each channel of that signal. For more information, see “Display Multiple Signals in the Time Scope”.

- **Type** — Select the type of trigger to use.

  - `Edge` — Trigger when the scope crosses a level threshold.

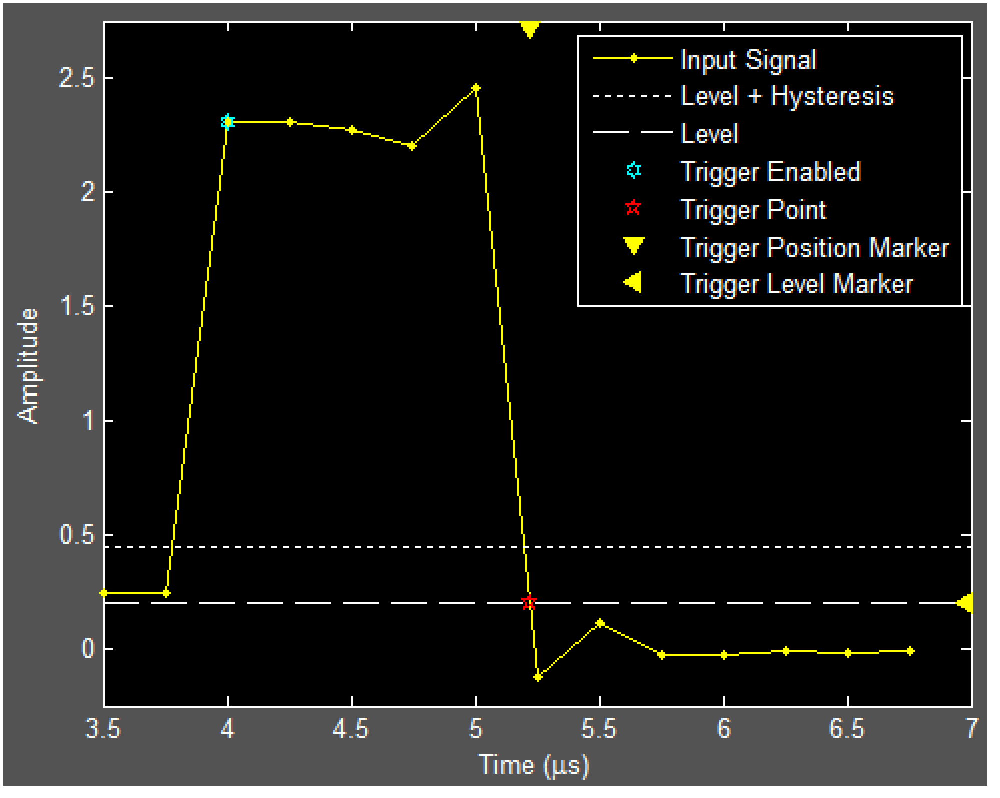
    For a rising edge, the scope enables the trigger event when the signal value becomes less than the level threshold minus hysteresis. The scope disables the trigger event when the signal becomes greater than the level threshold for the first time. The scope uses linear interpolation to generate a trigger event at the time when the signal crosses the level threshold.

Rising Edge Trigger Plot



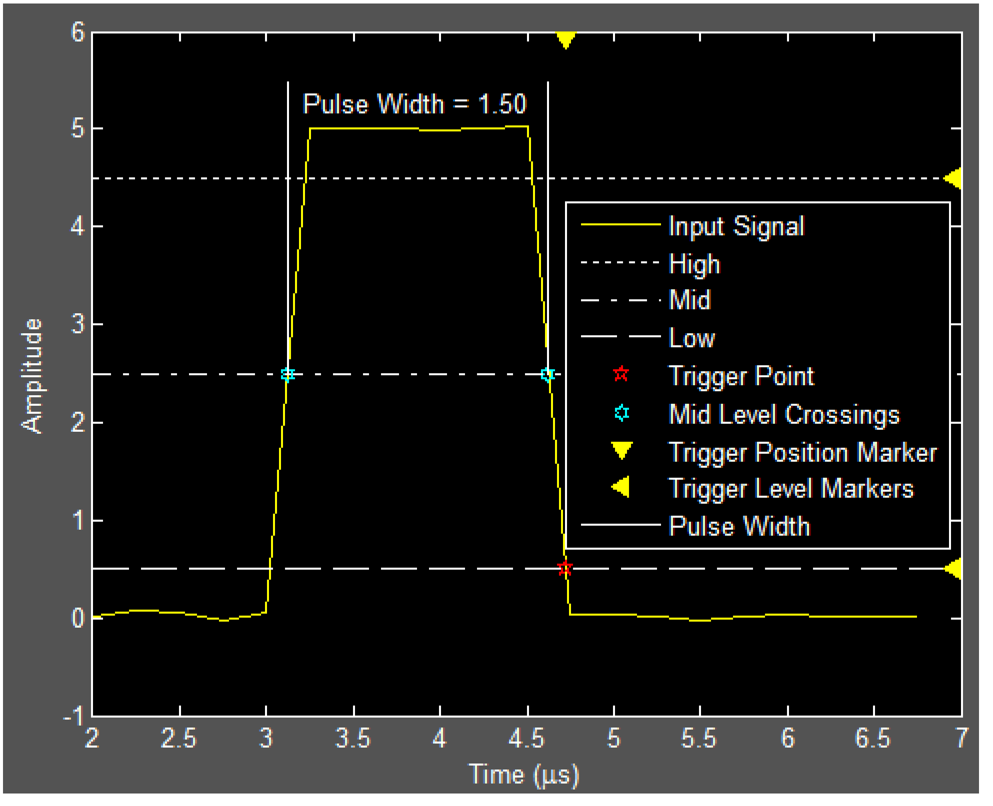
For a falling edge, the scope enables the trigger event when the signal value becomes greater than the level threshold plus hysteresis. The scope disables the trigger event when the signal becomes less than the level threshold for the first time. The scope uses linear interpolation to generate a trigger event at the time when the signal crosses the level threshold.

Falling Edge Trigger Plot



- `Pulse Width` — Trigger when the scope encounters a pulse whose width falls inside or outside specified time limits. You specify the range of valid time limits in the **Levels / Timing** pane. For a positive-polarity pulse, the scope encounters a trigger event when the signal crosses the low threshold for the second time. The scope measures the pulse width as the time between the first and second crossings of the middle threshold, located halfway between the high and low thresholds.

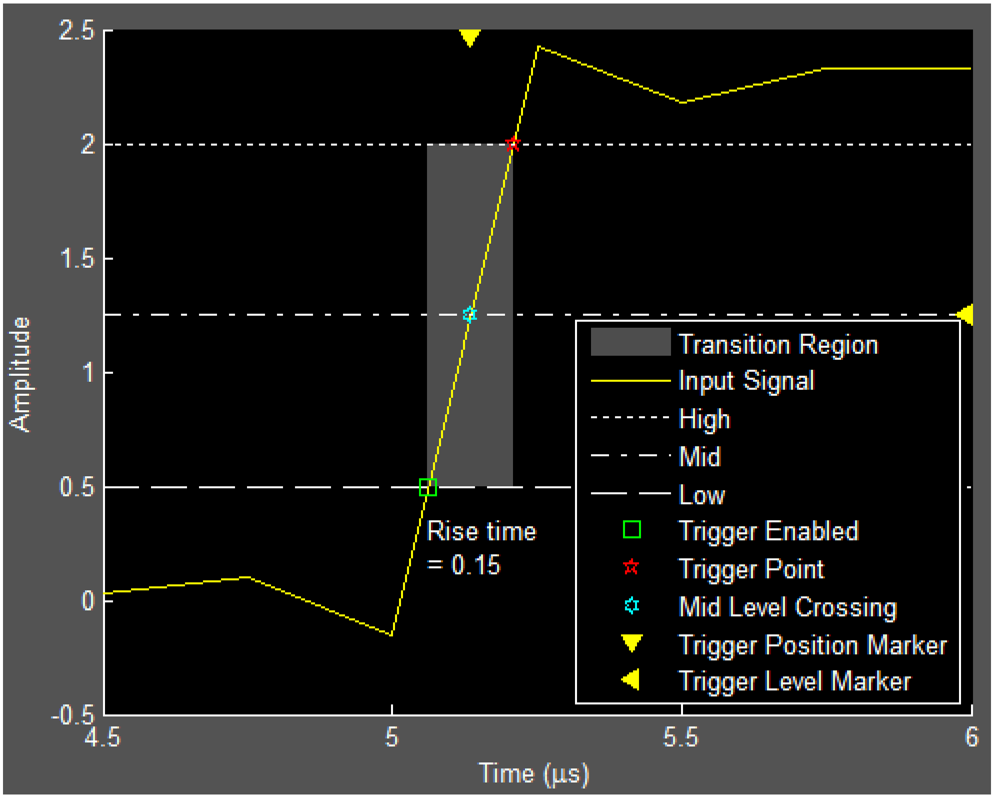
Pulse Width Trigger Plot



Note: A *Glitch*-type trigger looks for a pulse or spike with a duration less than a specified amount. You can implement a *Glitch* type trigger by using a `Pulse Width` type trigger and manually setting the **Max Width** parameter.

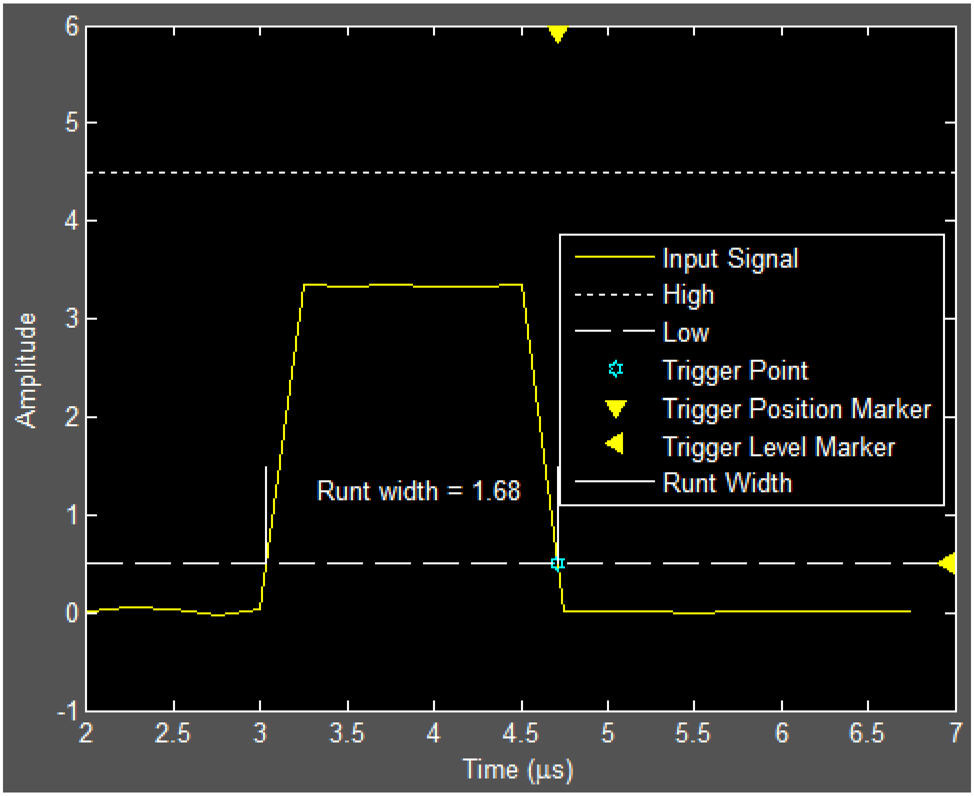
- `Transition` — Trigger on a rising or falling edge that crosses two levels, high and low, inside or outside a specified time interval. You specify the range of valid transition times in the **Levels / Timing** pane. For a rising transition, the scope encounters the trigger event when the signal crosses the high threshold. The transition time is when the signal crosses the middle threshold, located halfway between the high and low thresholds.

Transition Trigger Plot



- `Runt` — Trigger on a runt pulse, which crosses one threshold, high or low, but not both. For a positive-polarity runt pulse, the scope encounters a trigger event when the signal crosses the low threshold the second time, without ever crossing the high threshold. The scope measures the runt width as the time between the first and second crossings of the low threshold, as shown in the following figure. The runt width is the **Max Width** – **Min Width**. Any runt pulse width that is less than the minimum width or greater than the maximum width does not generate a trigger event.

Runt Trigger Plot

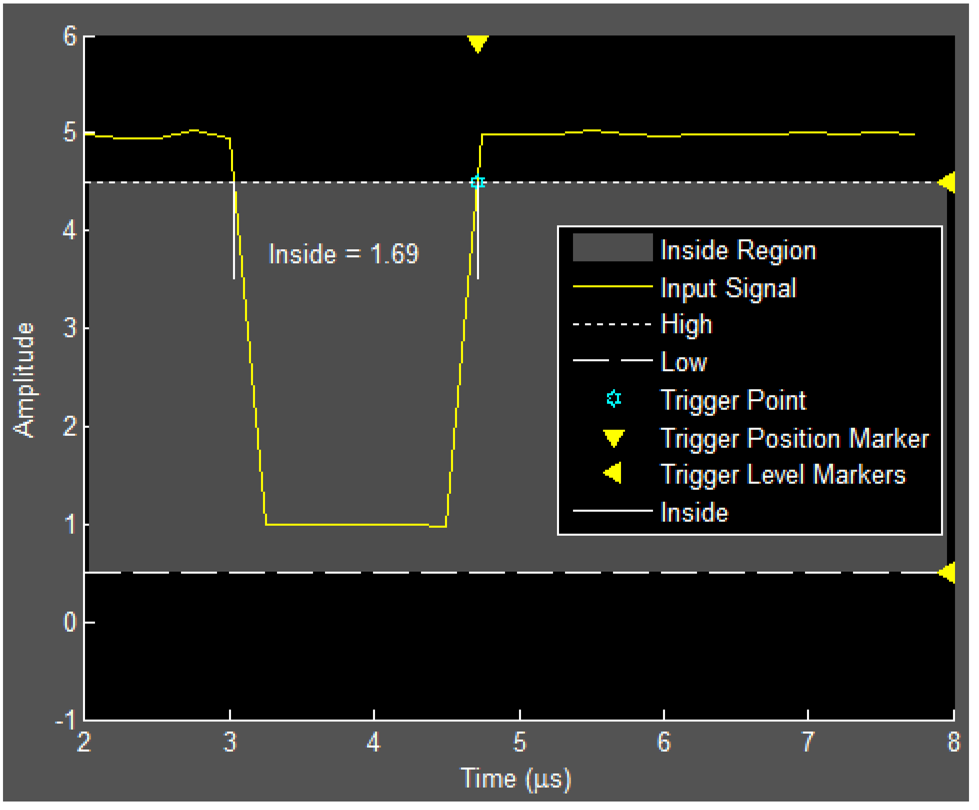


Note: You can also replicate a runt-type trigger by using a window-type trigger and setting **Polarity** to `Inside`.

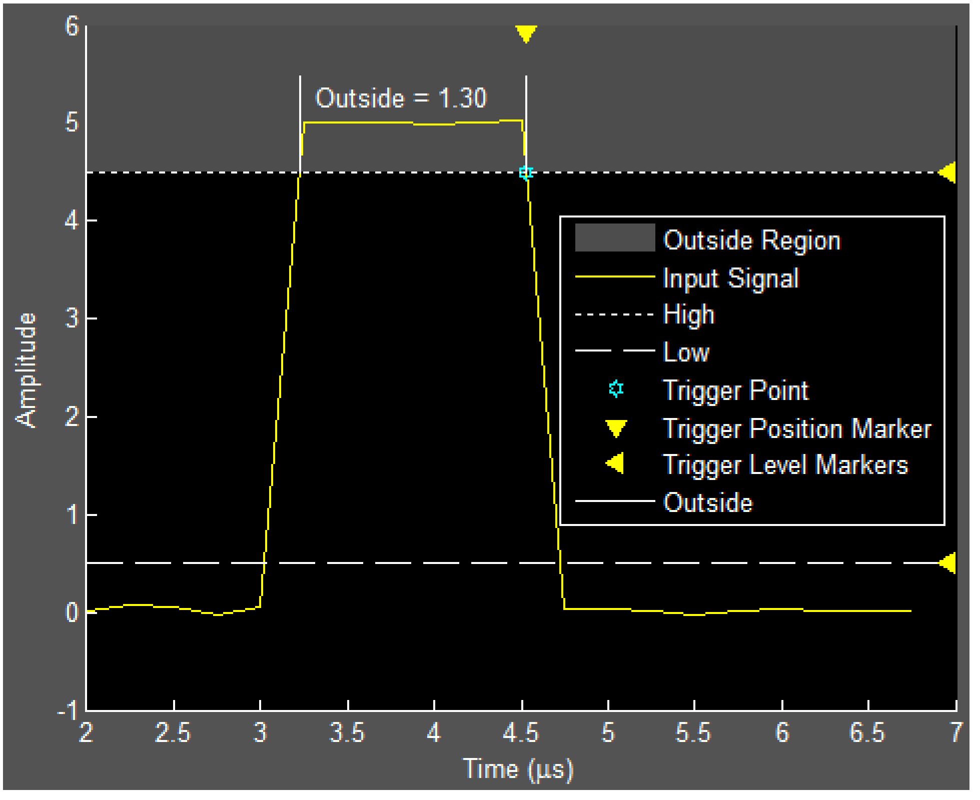
- `Window` — Trigger when the input signal stays within or outside the region defined by the high and low thresholds for a time.

  For an inside window, the scope encounters a trigger event when the signal enters and exits the inside region. For an outside window, the scope encounters a trigger event when the signal enters and exits the outside region.

Inside Window Trigger Plot



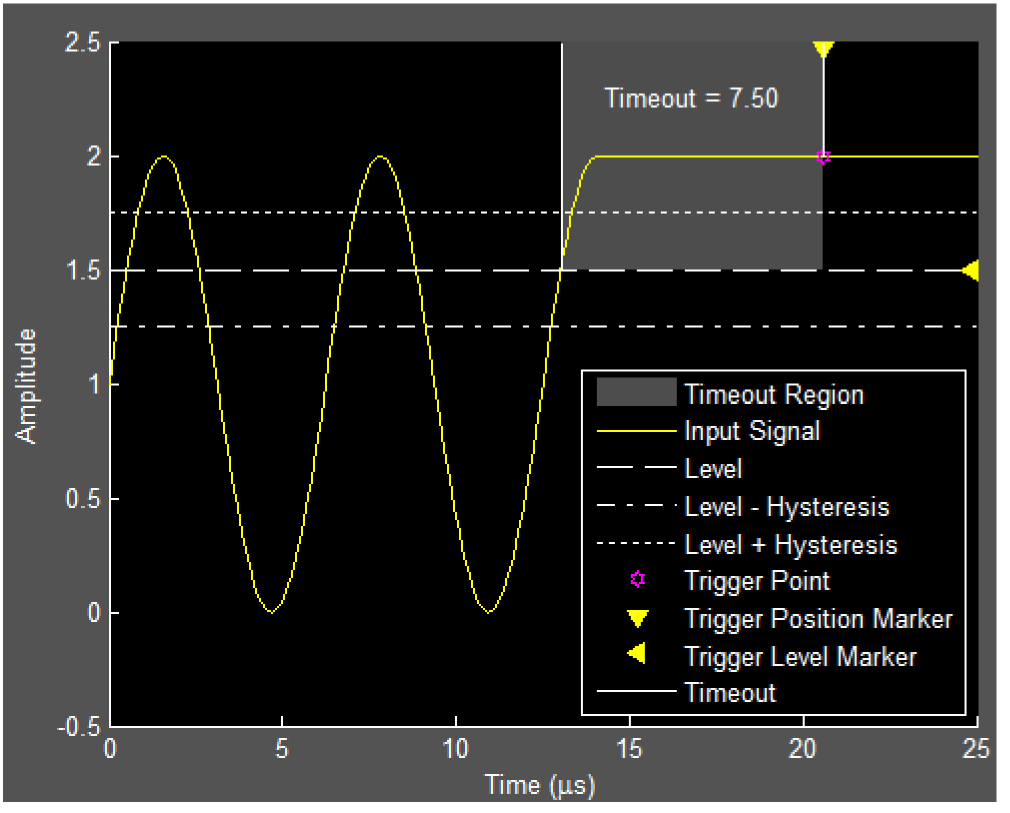
Outside Window Trigger Plot

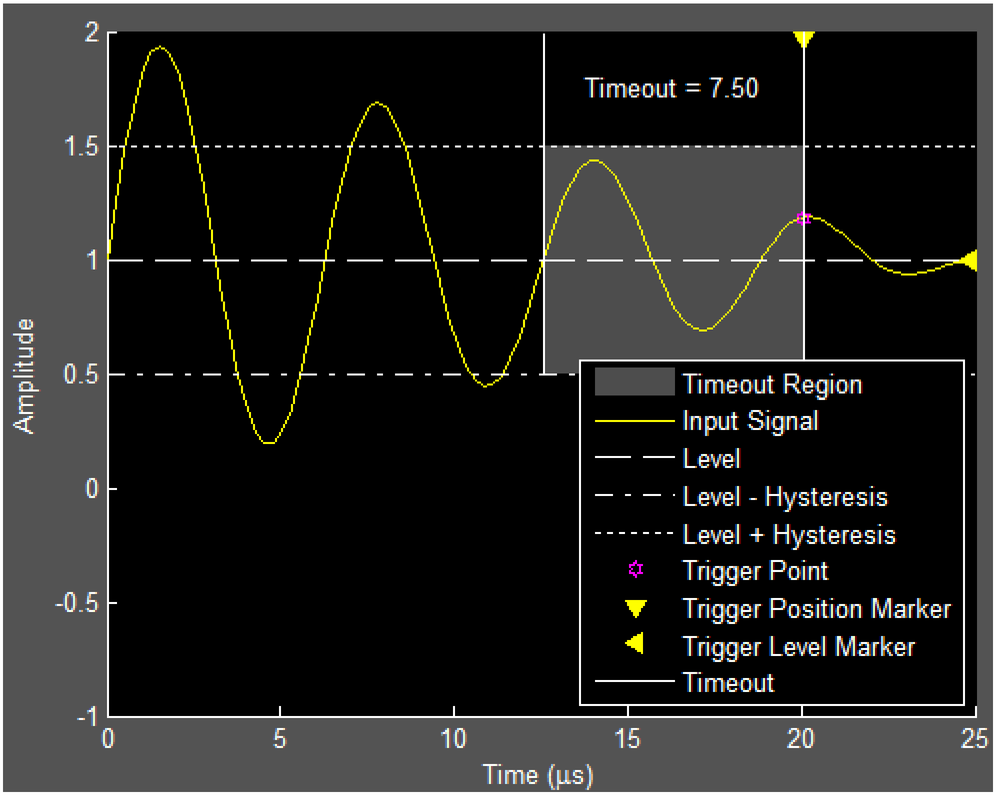


The scope encounters a trigger event when the signal crosses either the high or low threshold the second time.

- `Timeout` — Trigger when the input signal stays above or below a voltage threshold longer than a specified time. For a timeout trigger with polarity set to `Either` and a timeout duration of 7.50 seconds, the scope can encounter the trigger event 7.50 seconds after the signal crosses the level threshold the last time. Alternatively, the scope can encounter the trigger event when the signal stays within the boundaries defined by the hysteresis for 7.50 seconds after the signal crosses the level threshold.

Timeout Trigger Plots





- **Polarity** — Select the polarity of the trigger type. The option you choose for **Type** determines the options available for polarity.

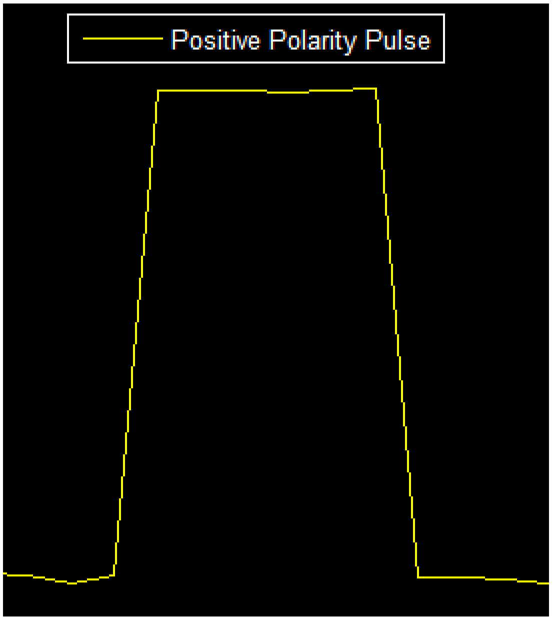
  For **Type** `Edge`:

  - `Rising` — Trigger on a *rising edge*, a transition from a low-state level to a high-state level.
  - `Falling` — Trigger on a *falling edge*, transition from a high-state level to a low-state level.
  - `Either` — Trigger on both rising edges and falling edges.

  For **Type** `Pulse Width`

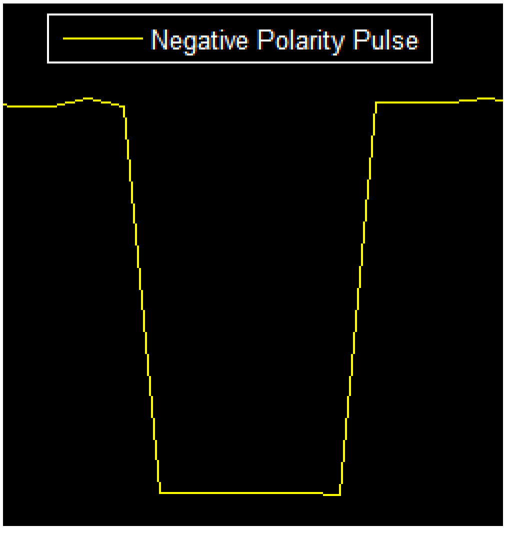
  - `Positive` — Trigger on a positive-polarity pulse.

Positive Polarity Plot



- `Negative` — Trigger on a negative-polarity pulse.

Negative Polarity Plot



For **Type** `Transition`

- `Rise Time` — Trigger based on how long the signal takes to transition from the low threshold to the high threshold.
- `Fall Time` — Trigger based on how long the signal takes to transition from the high threshold to the low threshold.
- `Either` — Trigger based on how long it takes to make either a rising or falling transition.

Type `Window`

- `Inside` — Trigger when the signal stays within the low and high levels for a specified time duration.
- `Outside` — Trigger when the signal stays outside of the low and high levels for a specified time duration.
- `Either` — Trigger on both inside and outside windows.

For **Type** `Timeout`

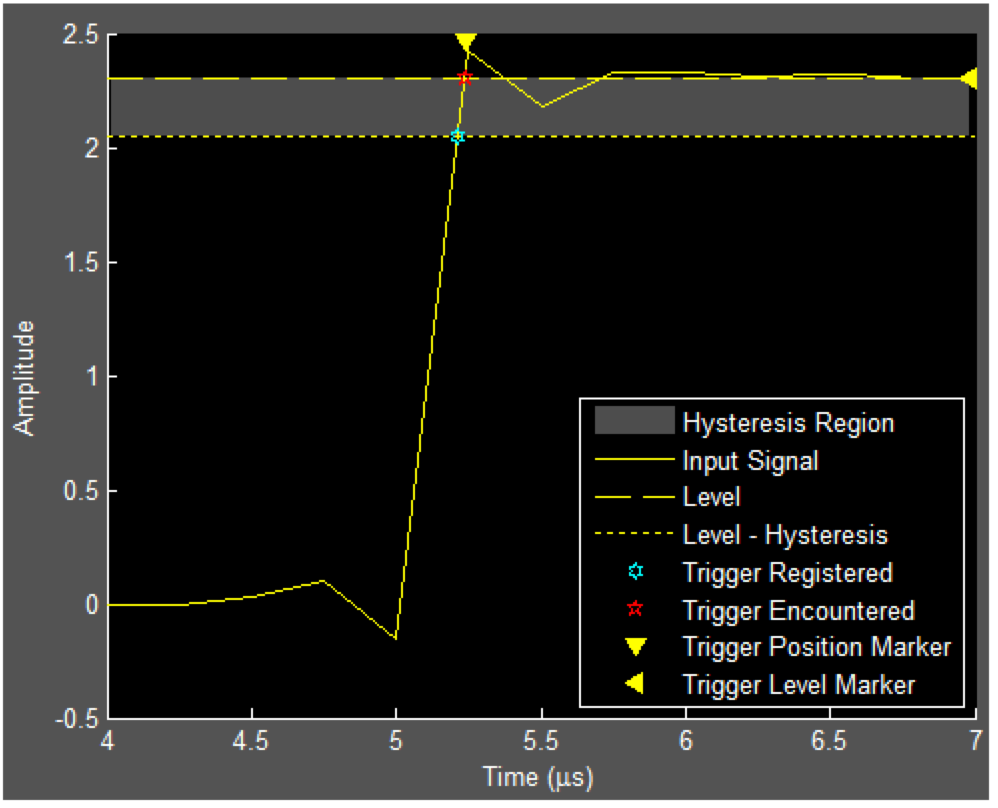
- `Rising` — Trigger when the signal does not cross the reference level from below.
- `Falling` — Trigger when the signal does not cross the reference level from above.
- `Either` — Trigger when the signal does not cross the reference level from either direction.

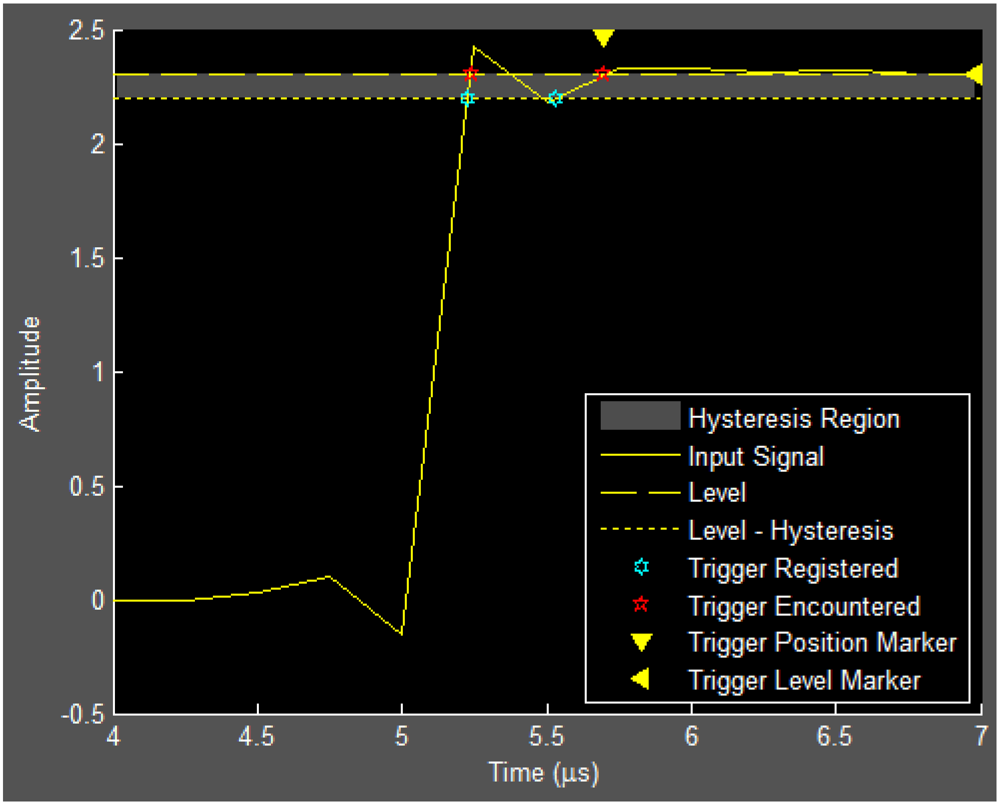
The **Levels / Timing** pane enables you to set the trigger level and hysteresis value. The option you choose for **Type** directly affects which level and timing parameters are available, as shown in the following table.

Trigger Type	Level Parameters	Auto-Level Setting	Timing Parameters
`Edge`	Level, Hysteresis	Level = 50%	n/a
`Pulse Width`, `Runt`	High, Low	High = 90%, Low = 10%	Min Width, Max Width
`Transition`, `Window`	High, Low	High = 90%, Low = 10%	Min Time, Max Time
`Timeout`	Level, Hysteresis	n/a	Timeout

- **Auto level** — Enable the Triggers panel to choose automatically the level parameters. If you set the trigger type to **Edge**, this option sets the **Level** parameter to 50% of the range of the source signal. If you set the trigger **Type** to `Timeout`, the Triggers panel does not show this option. Setting the trigger **Type** to other menu choices results in **High** and **Low** parameter adjustment. **Auto level** sets the **High** parameter to 90% of the range of the source signal and the **Low** parameter to 10% of the range of the source signal.
- **Level (V)** — Specify, in volts, the trigger level. This parameter is visible when you set **Type** to `Edge` or `Timeout`.
- **Hysteresis (V)** — Specify the hysteresis or noise reject value in volts. This parameter is visible when you set **Type** to `Edge` or `Timeout`. If the signal jitters inside this range and briefly crosses the trigger level, the scope does not register an event. For an edge trigger with rising polarity, the scope ignores any times that the signal crosses the trigger level within the hysteresis region. You can reduce the hysteresis region by decreasing the hysteresis value. If the signal then hysteresis level a second time and rises to the trigger level, a second trigger event occurs. See the following figures.

Hysteresis Plots





- **High (V)** — Specify, in volts, the value that denotes a positive polarity, or high-state level. This parameter is visible when you set **Type** to `Pulse Width`, `Transition`, `Runt`, or `Window`.
- **Low (V)** — Specify, in volts, the value that denotes a negative polarity, or low-state level. This parameter is visible when you set **Type** to `Pulse Width`, `Transition`, `Runt`, or `Window`.
- **Min Width (s)** — Specify, in seconds, the minimum pulse width. This parameter is visible when you set **Type** to `Pulse Width` or `Runt`.
- **Max Width (s)** — Specify, in seconds, the maximum pulse width. This parameter is visible when you set **Type** to `Pulse Width` or `Runt`.
- **Min Time (s)** — Specify, in seconds, the minimum duration. This parameter is visible when you set **Type** to `Transition` or `Window`.

- **Max Time (s)** — Specify, in seconds, the maximum duration. This parameter is visible when you set **Type** to `Transition` or `Window`.
- **Timeout (s)** — Specify, in seconds, the timeout duration. This parameter is visible when you set **Type** to `Timeout`.

The **Delay / Holdoff** pane enables you to offset the trigger position by a fixed delay or set the minimum possible time between trigger events.

- **Delay (s)** — Specify, in seconds, the fixed delay time by which to offset the trigger position. This parameter controls the amount of time the scope waits after a trigger event occurs before displaying a signal.
- **Holdoff (s)** — Specify, in seconds, the minimum possible time between trigger events. This amount of time is used to suppress data acquisition after a valid trigger event is encountered. A trigger holdoff prevents repeated occurrences of a trigger from occurring during the portion of a burst that is of interest.

Cursor Measurements Panel

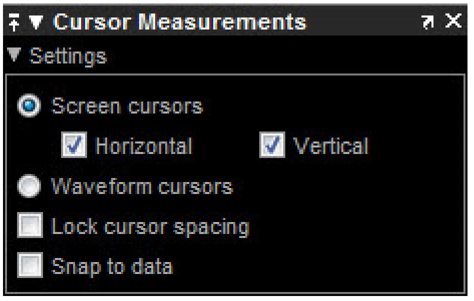
The **Cursor Measurements** panel displays screen cursors. The panel provides two types of cursors for measuring signals. Waveform cursors are vertical cursors that track along the signal. Screen cursors are both horizontal and vertical cursors that you can place anywhere in the display.

Note: If a data point in your signal has more than one value, the cursor measurement at that point is undefined and no cursor value is displayed.

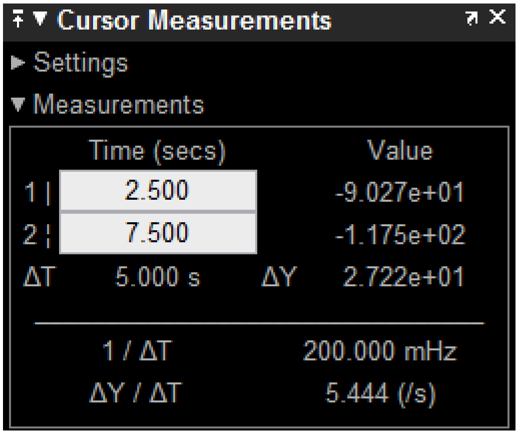
In the Scope menu, select **Tools** > **Measurements** > **Cursor Measurements**.

Alternatively, in the Scope toolbar, click the Cursor Measurements  button.

The **Settings** pane enables you to modify the type of screen cursors used for calculating measurements. When more than one signal is displayed, you can assign cursors to each trace individually.



- **Screen Cursors** — Shows screen cursors.
- **Horizontal** — Shows horizontal screen cursors.
- **Vertical** — Shows vertical screen cursors.
- **Waveform Cursors** — Shows cursors that attach to the input signals.
- **Lock Cursor Spacing** — Locks the frequency difference between the two cursors.
- **Snap to Data** — Positions the cursors on signal data points.



You can use the mouse or the left and right arrow keys to move vertical or waveform cursors and the up and down arrow keys for horizontal cursors.

The **Measurements** pane shows the time and value measurements.

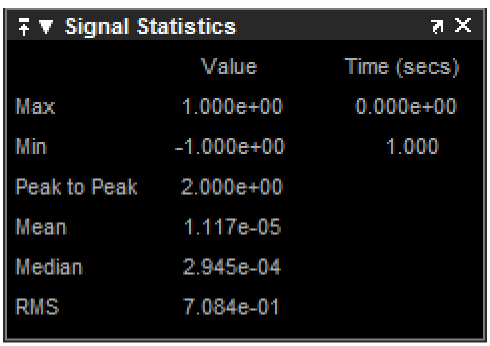
- **1 |** — Shows or enables you to modify the time or value at cursor number one, or both.

- **2 :**— Shows or enables you to modify the time or value at cursor number two, or both.
- **Δt**— Shows the absolute value of the difference in the times between cursor number one and cursor number two.
- **ΔV**— Shows the absolute value of the difference in signal amplitudes between cursor number one and cursor number two.
- **1/Δt**— Shows the rate, the reciprocal of the absolute value of the difference in the times between cursor number one and cursor number two.
- **ΔV/Δt**— Shows the scope, the ratio of the absolute value of the difference in signal amplitudes between cursors to the absolute value of the difference in the times between cursors.

Signal Statistics Panel

Note: The **Signal Statistics** panel requires a DSP System Toolbox or Simscape™ license.

The **Signal Statistics** panel displays the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal. It also shows the *x*-axis indices at which the maximum and minimum values occur. In the Scope menu, select **Tools** > **Measurements** > **Signal Statistics**. Alternatively, in the scope toolbar, click the Signal Statistics  button.



The statistics shown are:

- **Max** — The maximum or largest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `max` function reference.
- **Min** — The minimum or smallest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `min` function reference.
- **Peak to Peak** — The difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `peak2peak` function reference.
- **Mean** —The average or mean of all the values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `mean` function reference.
- **Median** — The median value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `median` function reference.
- **RMS** — Shows the difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `rms` function reference.

When you use the zoom options in the Scope, the Signal Statistics measurements automatically adjust to the time range shown in the display. For example, you can zoom in on one pulse to make the **Signal Statistics** panel display information about only that particular pulse.

The Signal Statistics measurements are valid for any units of the input signal. The letter after the value associated with each measurement represents the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

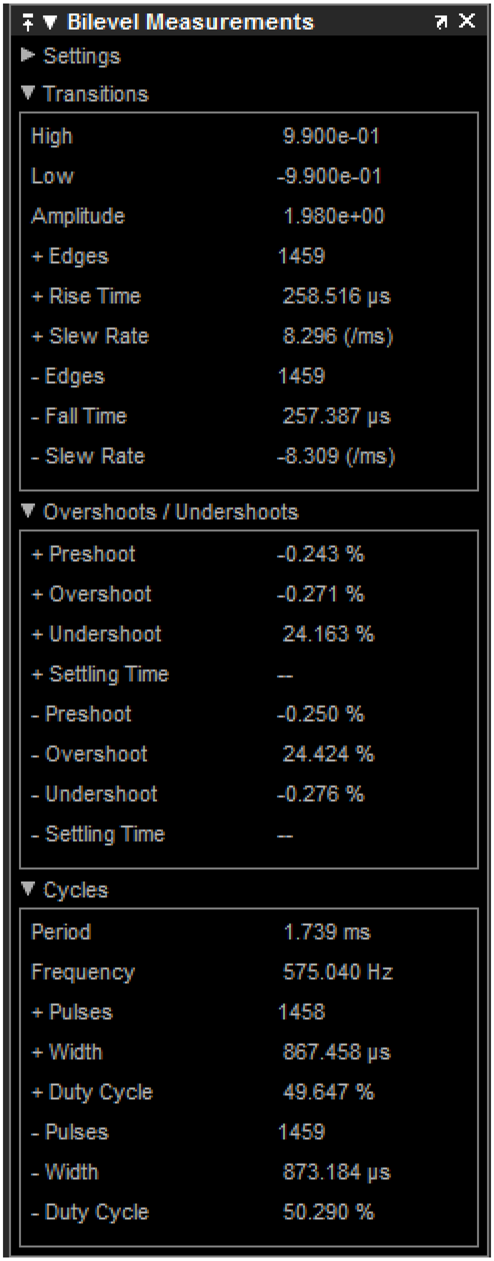
Bilevel Measurements Panel

Note: The **Bilevel Measurements** panel requires a DSP System Toolbox or Simscape license.

The **Bilevel Measurements** panel shows information about transitions, overshoots, undershoots, and cycles for a selected signal. You can choose to hide or display the

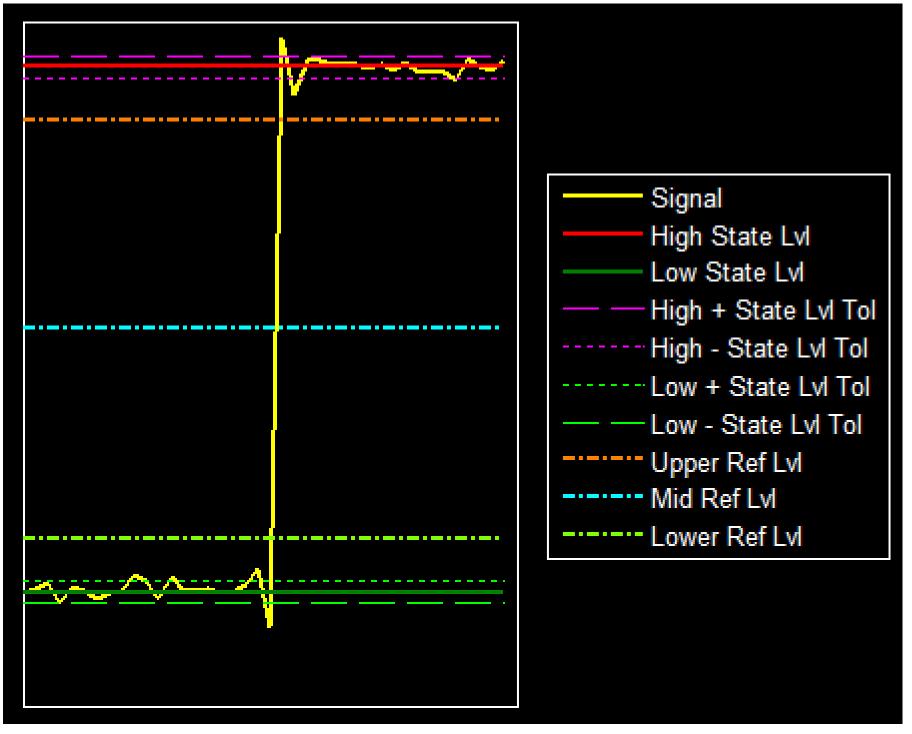
Bilevel Measurements panel. In the scope menu, select **Tools** > **Measurements** > **Bilevel Measurements**. Alternatively, in the scope toolbar, you can select the Bilevel Measurements  button.

When you use the zoom options in the Scope, the bilevel measurements automatically adjust to the time range shown in the display. For example, you can zoom in on one rising edge to make the **Bilevel Measurements** panel display information about only that particular rising edge. This feature does not apply to the **High** and **Low** measurements.


Bilevel Measurements
Settings
Transitions
High 9.900e-01
Low -9.900e-01
Amplitude 1.980e+00
+ Edges 1459
+ Rise Time 258.516 μs
+ Slew Rate 8.296 (/ms)
- Edges 1459
- Fall Time 257.387 μs
- Slew Rate -8.309 (/ms)
Overshoots / Undershoots
+ Preshoot -0.243 %
+ Overshoot -0.271 %
+ Undershoot 24.163 %
+ Settling Time --
- Preshoot -0.250 %
- Overshoot 24.424 %
- Undershoot -0.276 %
- Settling Time --
Cycles
Period 1.739 ms
Frequency 575.040 Hz
+ Pulses 1458
+ Width 867.458 μs
+ Duty Cycle 49.647 %
- Pulses 1459
- Width 873.184 μs
- Duty Cycle 50.290 %

The **Bilevel Measurements** panel is separated into four panes, labeled **Settings**, **Transitions**, **Overshoots/Undershoots**, and **Cycles**. You can expand each pane to see the available options.

Bilevel Measurements Plot



Note: Opening the **Overshoots/Undershoots** pane may reduce the number of available edges by 1 if the last edge of the pulse is not within the **Settle Seek** time of the end of the plot. This reduction in the number of edges changes the values of **Edges**, **Fall Time**, and **Slew Rate** in the **Transitions** pane.

The **Settings** pane enables you to modify the properties used to calculate various measurements involving transitions, overshoots, undershoots, and cycles. You can modify the high-state level, low-state level, state-level tolerance, upper-reference level, mid-reference level, and lower-reference level, as shown in the following figure.

- **Auto State Level** — When this check box is selected, the Bilevel measurements panel autodetects the high- and low- state levels of a bilevel waveform. For more information on the algorithm this option uses, see the Signal Processing Toolbox `statelevels` function reference. When this check box is cleared, you can enter in values for the high- and low- state levels manually.
  - **High** — Manually specify the value for a positive polarity or high-state level.
  - **Low** — Manually specify the value for a negative polarity or low-state level.
- **State Level Tolerance** — Tolerance within which the initial and final levels of each transition must be within their respective state levels. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Upper Ref Level** — Used to compute the end of the rise-time measurement or the start of the fall time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Mid Ref Level** — Used to determine when a transition occurs. This value is expressed as a percentage of the difference between the high- and low- state levels. The mid-reference level is shown as a horizontal line, and its corresponding mid-reference level instant is shown as a vertical line.
- **Lower Ref Level** — Used to compute the end of the fall-time measurement or the start of the rise-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Settle Seek** — The duration after the mid-reference level instant when each transition occurs used for computing a valid settling time. This value is equivalent to the input parameter, `D`, which you can set when you run the `settlingtime` function. The settling time is displayed in the **Overshoots/Undershoots** pane.

The **Transitions** pane displays calculated measurements associated with the input signal changing between its two possible state level values, high and low. The Transition measurements assume that the amplitude of the input signal is in units of volts. Convert all input signals to volts for the Transition measurements to be valid.

A positive-going transition, or *rising edge*, in a bilevel waveform is a transition from the low-state level to the high-state level. A positive-going transition has a slope value greater than zero. Whenever there is a plus sign (+) next to a text label, this symbol refers to measurement associated with a rising edge, a transition from a low-state level to a high-state level.

A negative-going transition, or falling edge, in a bilevel waveform is a transition from the high-state level to the low-state level. A negative-going transition has a slope value less than zero. Whenever there is a minus sign (–) next to a text label, this symbol refers to measurement associated with a falling edge, a transition from a high-state level to a low-state level.

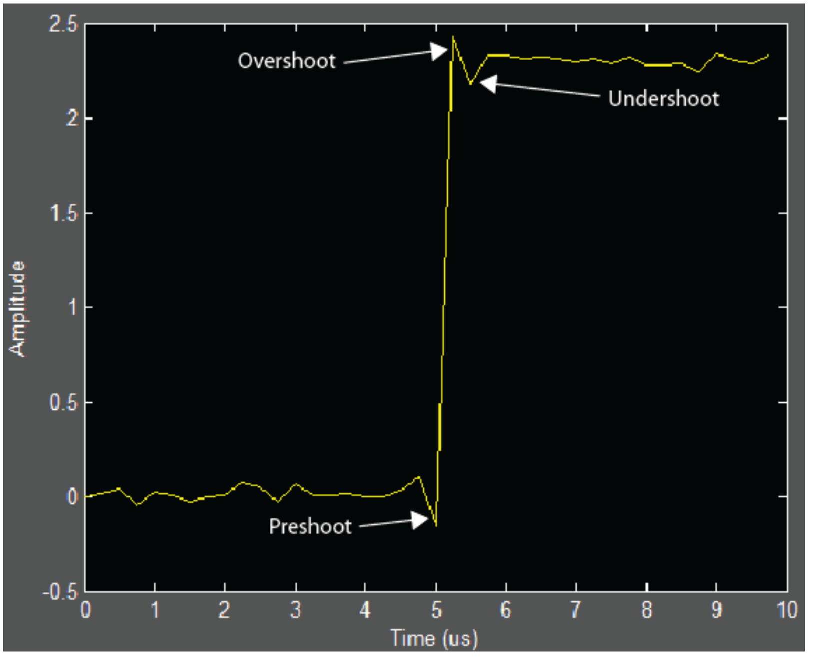
- **High** — The high-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** or**Time** tab of the Visuals—Time Domain Properties dialog box. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `statelevels` function reference.
- **Low** — The low-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** or **Time** tab of the Visuals—Time Domain Properties dialog box. The tab in which **Time Span** appears depends on whether you launched the scope from a from a MATLAB System object or from a Simulink block, respectively. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `statelevels` function reference.
- **Amplitude** — Difference in amplitude between the high-state level and the low-state level.
- **+ Edges** — Total number of positive-polarity, or rising, edges counted within the displayed portion of the input signal.
- **+ Rise Time** — Average amount of time required for each rising edge to cross from the lower-reference level to the upper-reference level. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `risetime` function reference.
- **+ Slew Rate** — Average slope of each rising-edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. The region in which the slew rate is calculated appears in gray.

  For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `slewrate` function reference.
- **– Edges** — Total number of negative-polarity or falling edges counted within the displayed portion of the input signal.
- **– Fall Time** — Average amount of time required for each falling edge to cross from the upper-reference level to the lower-reference level. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `falltime` function reference.
- **– Slew Rate** — Average slope of each falling edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. For

more information on the algorithm this measurement uses, see the Signal Processing Toolbox `slewrate` function reference.

The **Overshoots/Undershoots** pane displays calculated measurements involving the distortion and damping of the input signal. *Overshoot* and *undershoot* refer to the amount that a signal, respectively, exceeds and falls below its final steady-state value. *Preshoot* refers to the amount before a transition that a signal varies from its initial steady-state value. This figure shows preshoot, overshoot, and undershoot for a rising-edge transition.

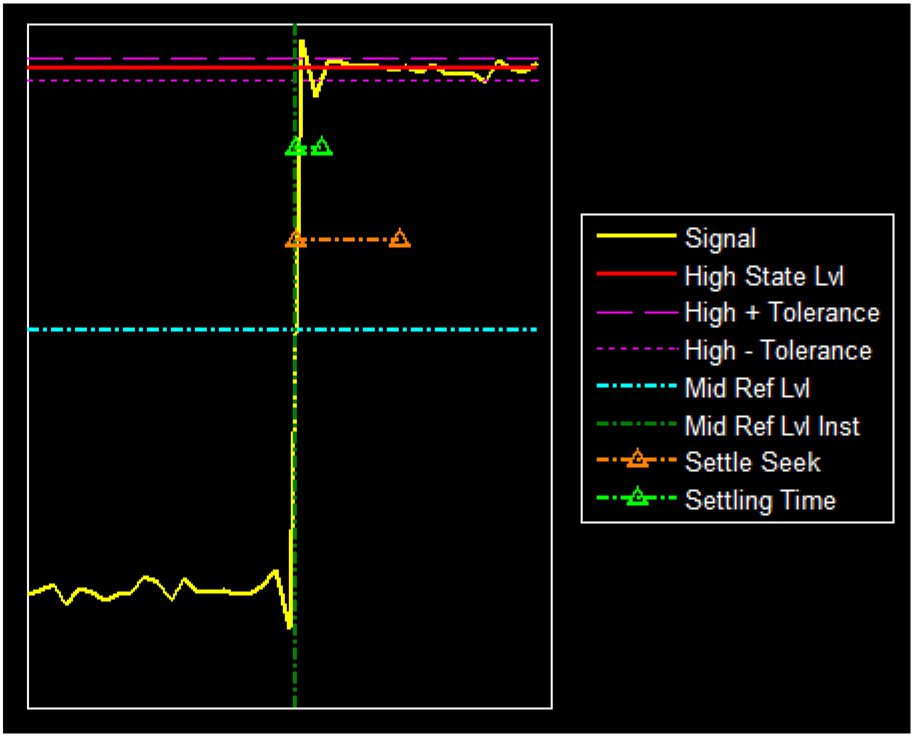
Overshoot/Undershoot Plot



- + *Preshoot* — Average lowest aberration in the region immediately preceding each rising transition.
- + *Overshoot* — Average highest aberration in the region immediately following each rising transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `overshoot` function reference.
- + *Undershoot* — Average lowest aberration in the region immediately following each rising transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `undershoot` function reference.

- + *Settling Time* — Average time required for each rising edge to enter and remain within the tolerance of the high-state level for the remainder of the settle seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the high-state level. This crossing is illustrated in the following figure.

Settling Time Plot



You can modify the settle seek duration parameter in the **Settings** pane. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `settlingtime` function reference.

- – *Preshoot* — Average highest aberration in the region immediately preceding each falling transition.

- – *Overshoot* — Average highest aberration in the region immediately following each falling transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `overshoot` function reference.
- – *Undershoot* — Average lowest aberration in the region immediately following each falling transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `undershoot` function reference.
- – *Settling Time* — Average time required for each falling edge to enter and remain within the tolerance of the low-state level for the remainder of the settle seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the low-state level. You can modify the settle seek duration parameter in the **Settings** pane. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `settlingtime` function reference.

The **Cycles** pane displays calculated measurements of to repetitions or trends in the displayed portion of the input signal.

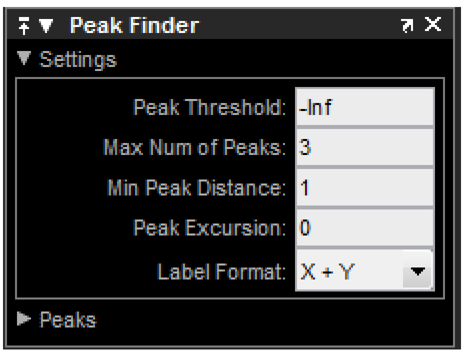
- **Period** — Average duration between adjacent edges of identical polarity within the displayed portion of the input signal. The Bilevel measurements panel calculates period as follows. It takes the difference between the mid-reference level instants of the initial transition of each positive-polarity pulse and the next positive-going transition. These mid-reference level instants appear as red dots.

  For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulseperiod` function reference.
- **Frequency** — Reciprocal of the average period. Whereas period is typically measured in some metric form of seconds, or seconds per cycle, frequency is typically measured in hertz or cycles per second.
- **+ Pulses** — Number of positive-polarity pulses counted.
- **+ Width** — Average duration between rising and falling edges of each positive-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulsewidth` function reference.
- **+ Duty Cycle** — Average ratio of pulse width to pulse period for each positive-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `dutycycle` function reference.
- **– Pulses** — Number of negative-polarity pulses counted.

- – **Width** — Average duration between rising and falling edges of each negative-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulsewidth` function reference.
- – **Duty Cycle** — Average ratio of pulse width to pulse period for each negative-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `dutycycle` function reference.

Peak Finder Panel

The **Peak Finder** panel displays the maxima, showing the *x*-axis values at which they occur. Peaks are defined as a local maximum where lower values are present on both sides of a peak. Endpoints are not considered to be peaks. This panel allows you to modify the settings for peak threshold, maximum number of peaks, and peak excursion. You can choose to hide or display the **Peak Finder** panel. In the scope menu, select **Tools** > **Measurements** > **Peak Finder**. Alternatively, in the scope toolbar, select the Peak Finder  button.



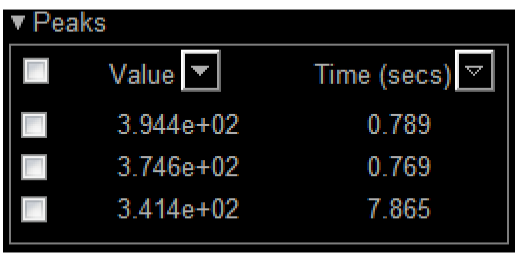
The **Peak finder** panel is separated into two panes, labeled **Settings** and **Peaks**. You can expand each pane to see the available options.

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input signal. For more information on the algorithms this pane uses, see the Signal Processing Toolbox `findpeaks` function reference.

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the `MINPEAKHEIGHT` parameter, which you can set when you run the `findpeaks` function.
- **Max Num of Peaks** — The maximum number of peaks to show. The value you enter must be a scalar integer from 1 through 99. This setting is equivalent to the `NPEAKS` parameter, which you can set when you run the `findpeaks` function.
- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the `MINPEAKDISTANCE` parameter, which you can set when you run the `findpeaks` function.
- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples The peak excursion setting is equivalent to the `THRESHOLD` parameter, which you can set when you run the `findpeaks` function.
- **Label Format** — The coordinates to display next to the calculated peak values on the plot. To see peak values, expand the **Peaks** pane and select the check boxes associated with individual peaks of interest. By default, both *x*-axis and *y*-axis values are displayed on the plot. Select which axes values you want to display next to each peak symbol on the display.

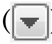
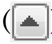
  - `X+Y` — Display both *x*-axis and *y*-axis values.
  - `X` — Display only *x*-axis values.
  - `Y` — Display only *y*-axis values.

The **Peaks** pane displays all of the largest calculated peak values. It also shows the coordinates at which the peaks occur, using the parameters you define in the **Settings** pane. You set the **Max Num of Peaks** parameter to specify the number of peaks shown in the list.



The numerical values displayed in the **Value** column are equivalent to the `pks` output argument returned when you run the `findpeaks` function. The numerical values

displayed in the second column are similar to the `locs` output argument returned when you run the `findpeaks` function.

The Peak Finder displays the peak values in the **Peaks** pane. By default, the **Peak Finder** panel displays the largest calculated peak values in the **Peaks** pane in decreasing order of peak height. Use the sort descending button () to rearrange the category and order by which Peak Finder displays peak values. Click this button again to sort the peaks in ascending order instead. When you do so, the arrow changes direction to become the sort ascending button (). A filled sort button indicates that the peak values are currently sorted in the direction of the button arrow. If the sort button is not filled (), then the peak values are sorted in the opposite direction of the button arrow. The **Max Num of Peaks** parameter still controls the number of peaks listed.

Use the check boxes to control which peak values are shown on the display. By default, all check boxes are cleared and the **Peak Finder** panel hides all the peak values. To show all the peak values on the display, select the check box in the top-left corner of the **Peaks** pane. To hide all the peak values on the display, clear this check box. To show an individual peak, select the check box directly to the left of its **Value** listing. To hide an individual peak, clear the check box directly to the left of its **Value** listing.

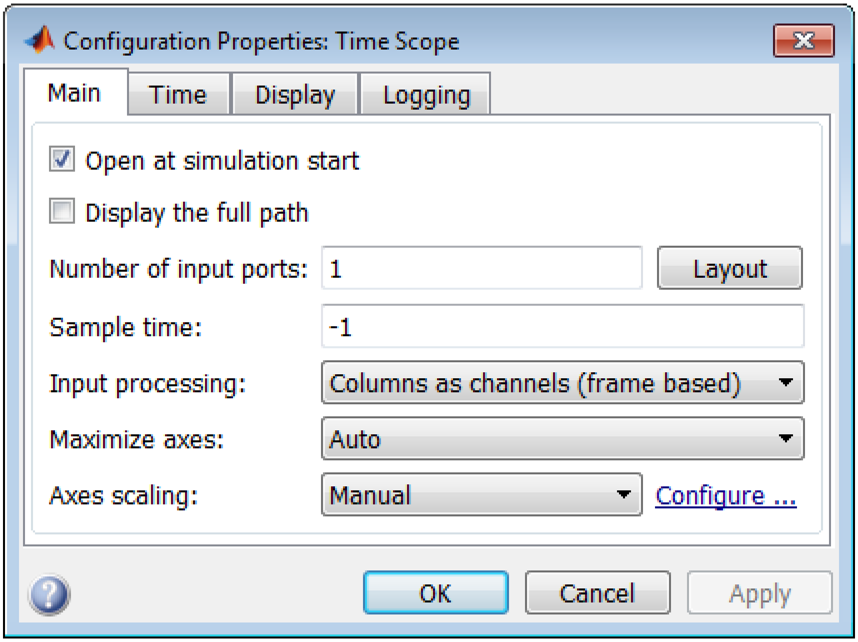
The Peaks are valid for any units of the input signal. The letter after the value associated with each measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Configuration Properties Dialog Box

The Configuration Properties dialog box controls various properties about the Time Scope displays. From the Time Scope menu, select **View** > **Configuration Properties** to open this dialog box. Alternatively, in the Time Scope toolbar, click the Configuration Properties  button.

Main Pane

The **Main** pane of the Configuration Properties dialog box appears as follows.



Open at simulation start

Select this check box to ensure that the scope opens when the simulation starts. The following table summarizes the interaction between the **Open at simulation start** check box and the Scope figure.

Open at simulation start	Scope figure status when model saved	Scope figure opens
Checked	Closed	At simulation start
Checked	Open	At model loading
Not checked	Closed	Only if you double-click the Scope block icon in the model
Not checked	Open	At model loading

Display the full path

Select this check box to display in the title bar the path of this scope in this model.

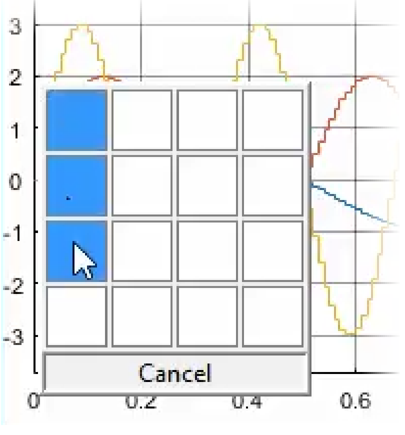
Number of input ports

Specify the number of input ports to show on the left side of the scope block. You can also specify the number of input ports using **File** > **Number of input ports**.

Layout

Specify number and arrangement of displays. The maximum layout is 16 rows by 16 columns.

To expand the layout grid beyond 4 by 4, click within the dialog box and drag. Maximum of 16 rows by 16 columns.



- If the number of displays are equal to the number of ports, signals from each port appear on separate displays.
- If the number of displays are less than the number of ports, signals from additional ports appear on the last display.
- For layouts with multiple columns and rows, ports are mapped down then across.

Sample time

Specify the sample time in seconds, as a scalar value. If you enter -1, the sample time is inherited. For information on sample time, see “What Is Sample Time?” (Simulink) and “Types of Sample Time” (Simulink).

Input processing

Specify whether the Time Scope treats the input signal as `Columns as channels (frame based)` or `Elements as channels (sample based)`.

Frame-based processing is only available for discrete input signals. For more information about frame-based input channels, see “What Is Frame-Based Processing?”. For an example that uses the Time Scope block and frame-based input signals, see “Display Time-Domain Data”.

Maximize axes

Specify whether to display the scope in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- `Auto` — In this mode, the axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- `On` — In this mode, the axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- `Off` — In this mode, none of the axes appear maximized.

This property is Tunable (Simulink).

The default setting is `Auto`.

Axes scaling

Specify when the scope automatically scales the axes. You can select one of the following options:

- `Manual` — When you select this option, the scope does not automatically scale the axes. You can manually scale the axes in any of the following ways:
  - Select **Tools** > **Axes Scaling Properties**.
  - Press one of the **Scale Axis Limits** toolbar buttons.
  - When the scope figure is the active window, press **Ctrl** and **A** simultaneously.
- `Auto` — When you select this option, the scope scales the axes as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** check box.

- `After N Updates` — Selecting this option causes the scope to scale the axes after a specified number of updates. This option is useful and more efficient when your scope display starts with one axis scale, but quickly reaches a different steady state axis scale. Selecting this option shows the **Number of updates** edit box.

By default, this property is set to `Auto`. This property is Tunable (Simulink).

Axes scaling - Configure — (Available only on the **Configuration Properties Main** tab) Click the **Configure** link to the right of the **Axes scaling** property to see additional axes scaling properties. After you click this button, its label changes to **Hide**.

By default, this property is set to `Auto`. This property is Tunable (Simulink).

Number of updates

Specify as a positive integer the number of updates after which to scale the axes. This property appears only when you select `After N Updates` for the **Axes scaling** property. Tunable (Simulink).

Do not allow Y-axis limits to shrink

When you select this property, the *y*-axis is allowed only to grow during axes scaling operations. If you clear this check box, the *y*-axis or color limits may shrink during axes scaling operations.

This property appears only when you select `Auto` for the **Axis scaling** property. When you set the **Axes scaling** property to `Manual` or `After N Updates`, the *y*-axis or color limits are allowed to shrink. Tunable (Simulink).

Scale axes limits at stop

Select this check box to scale the axes when the simulation stops. The *y*-axis is always scaled. The *x*-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

Y-axis Data range (%)

Set the percentage of the *y*-axis that the scope uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the Scope scales the *y*-axis limits such that your data uses the entire *y*-axis range. If you then set this property to `30`, the scope increases the *y*-axis range such that your data uses only 30% of the *y*-axis range. Tunable (Simulink).

Y-axis Align

Specify where the scope aligns your data along the *y*-axis when it scales the axes. You can select `Top`, `Center`, or `Bottom`. Tunable (Simulink).

Autoscale X-axis limits

Check this box to allow the scope to scale the *x*-axis limits when it scales the axes. If **Axes scaling** is set to `Auto`, checking **Autoscale X-axis limits** only scales the data currently within the axes, not the entire signal in the data buffer. If **Autoscale X-axis limits** is on and the resulting axis is greater than the span of the scope, trigger position markers will not be displayed. Triggers are controlled using the Trigger Measurements panel. Tunable (Simulink).

X-axis Data range (%)

Set the percentage of the *x*-axis that the scope uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the scope scales the *x*-axis limits such that your data uses the entire*x*-axis range. If you then set this property to `30`, the scope increases the *x*-axis range such that your data uses only 30% of the *x*-axis range. Use the *x*-axis **Align** property to specify data placement along the *x*-axis.

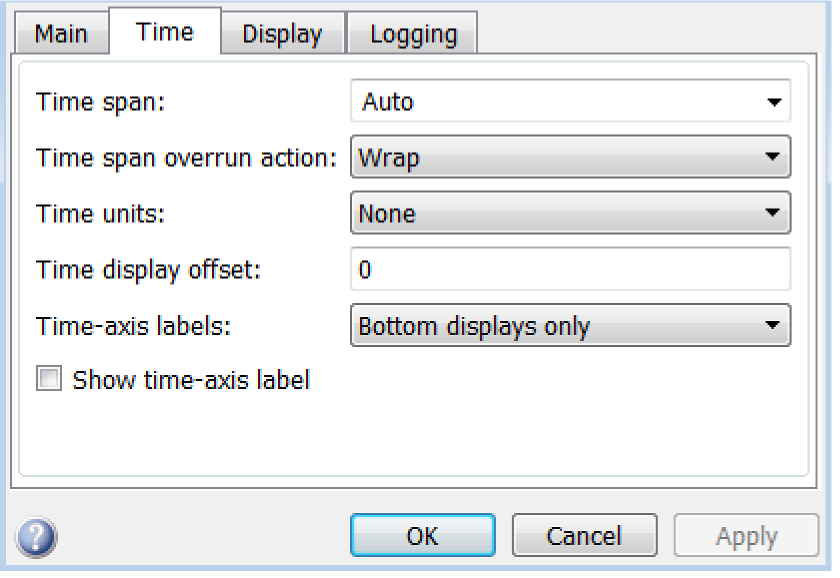
This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

X-axis Align

Specify how the scope aligns your data along the *x*-axis: `Left`, `Center`, or `Right`. This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

Time Pane

The **Time** pane of the Configuration Properties dialog box appears as follows.



Time span

Specify the time span, either by selecting a predefined option or by entering a numeric value in seconds. You can select one of the following options:

- `Auto` — Time Scope automatically calculates the minimum and maximum time-axis limits for time span.

  - Minimum time-axis limit = Simulation “Start time” (Simulink)
  - Maximum time-axis limit = Simulation “Stop time” (Simulink) + max(*FrameRate* * (*FrameSize*–1) / *FrameSize*)

  FrameSize is a vector equal to the number of rows in each input signal. *FrameRate* is the reciprocal of the sample time for each frame. The Time Scope System object calculates the minimum and maximum time-axis limits as follows:

  - Minimum time-axis limit = min(`TimeDisplayOffset`)
  - Maximum time-axis limit = max(`TimeDisplayOffset`) + max(1/`SampleRate`.**FrameSize*)

  where `TimeDisplayOffset` on page 4-1707 and `SampleRate` on page 4-1706 are the values of their respective properties. This property is Tunable (Simulink).

- `One frame period` — In this mode, the Time Scope uses the frame period of the input signal to the Time Scope block. This option is only available when the **Input processing** parameter is set to `Columns as channels (frame based)`. This option is not available when you set the **Input processing** parameter to `Elements as channels (sample based)`.
- `<user defined>` — In this mode, you specify the time span by replacing the text `<user defined>` with a numeric value in seconds.

The scope sets the time-axis limits using the value of this property and the value of the **Time display offset** property. Tunable (Simulink)

Time span overrun action

Specify how the scope displays new data beyond the visible time span. You can select one of the following options:

- `Wrap` — In this mode, the scope displays new data until the data reaches the maximum time-axis limit. When the data reaches the maximum time-axis limit of the scope window, the scope clears the display. The scope then updates the time offset value and begins displaying subsequent data points starting from the minimum time-axis limit.
- `Scroll` — In this mode, the scope scrolls old data to the left to make room for new data on the right side of the scope display. This mode is graphically intensive and can affect run-time performance. However, it is beneficial for debugging and monitoring time-varying signals.

This property is Tunable (Simulink).

The default setting is `Wrap`.

Time units

Specify the units used to describe the time axis. You can select one of the following options:

- `Metric` — In this mode, the scope converts the times on the time axis to the most appropriate measurement units. These units include milliseconds, microseconds, nanoseconds, minutes, days, etc. The scope chooses the appropriate measurement units based on the minimum time-axis limit and the maximum time-axis limit of the scope window.
- `Seconds` — In this mode, the scope always displays the units on the time axis as seconds.

- `None` — In this mode, the scope does not display any units on the time axis. The scope only shows the word `Time` on the time axis.

This property is Tunable (Simulink).

The default setting is `Metric`.

Time display offset

You use this property to offset the values displayed on the time axis by a specified number of seconds. When you specify a scalar value, the scope offsets all channels equally. When you specify a vector of offset values, the scope offsets each channel independently. Tunable (Simulink).

When you specify a **Time display offset** vector of length *N*, the scope offsets the input channels as follows:

- When *N* is equal to the number of input channels, the scope offsets each channel according to its corresponding value in the offset vector.
- When *N* is less than the number of input channels, the scope applies the values you specify in the offset vector to the first *N* input channels. The scope does not offset the remaining channels.
- When *N* is greater than the number of input channels, the scope offsets each input channel according to the corresponding value in the offset vector. The scope ignores all values in the offset vector that do not correspond to a channel of the input.

The scope computes the time-axis range using the values of the **Time display offset** and **Time span** properties. For example, if you set the **Time display offset** to `5e-6` and the **Time span** to `25e-6`, the scope sets the time-axis minimum to 5 seconds and the maximum to 30 seconds.

Similarly, when you specify a vector of values, the scope sets the minimum time-axis limit using the smallest value in the vector. To set the maximum time-axis limit, the scope sums the largest value in the vector with the value of the **Time span** property. For more information, see "Signal Display" on page 2-1650.

Time-axis labels

Specify how to display the time units used to describe the time axis. The default setting is `All`. You can select one of the following options:

- `All` — The time-axis labels appear in all displays.
- `None` — The time-axis labels do not appear in the displays.

- `Bottom Displays Only` — The time-axis labels appear in only the bottom row of the displays.

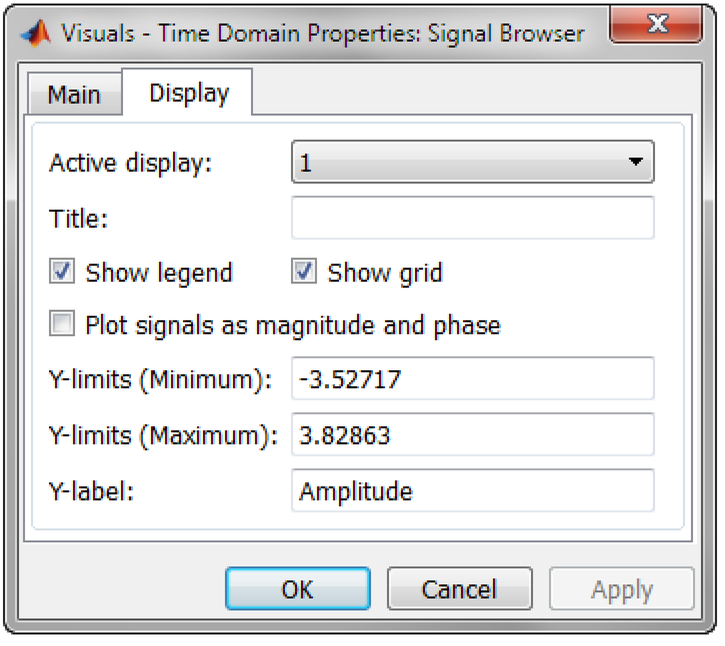
Tunable (Simulink).

Show time-axis label

Select this check box to show the time-axis label on the scope display. If this box is not checked, the scope does not display the time-axis label, but still displays tick marks and other time-axis items. This check box is not available if **Time-axis labels** is `None`.

Display Pane

The **Display** pane of the Configuration Properties dialog box appears as follows.



Active display

Specify the active display as an integer to get and set relevant properties. The number of a display corresponds to its column-wise placement index. Set this property to control which display has its axes colors, line properties, marker properties, and visibility changed. Tunable (Simulink)

When you use the Layout option to tile the window into multiple displays, the display highlighted in blue is referred to as the *active display*. The default setting is `1`.

Title

Specify the active display title as a character vector. Enter `%<SignalLabel>` to use the signal labels in the Simulink Model as the axes titles. By default, the active display has no title. Tunable (Simulink).

Show legend

Select this check box to show the legend in the display. The channel legend displays a name for each channel of each input signal. When the legend appears, you can place it anywhere inside of the scope window. To turn off the legend, clear the **Show legend** check box. This parameter applies only when the **View Type** is `Spectrum` or `Spectrum and spectrogram`.

You can edit the name of any channel in the legend. To do so, double-click the current name, and enter a new channel name. By default, the scope names each channel according to either its signal name or the name of the block from which it comes. If the signal has multiple channels, the scope uses an index number to identify each channel of that signal.

Time Scope does not display signal names that were labeled within an unmasked subsystem. You must label all input signals to the scope block that originate from an unmasked subsystem.

The legend lets you modify what signals are shown. To show only one signal, click the signal name. To toggle a signal on/off, right-click the signal name.

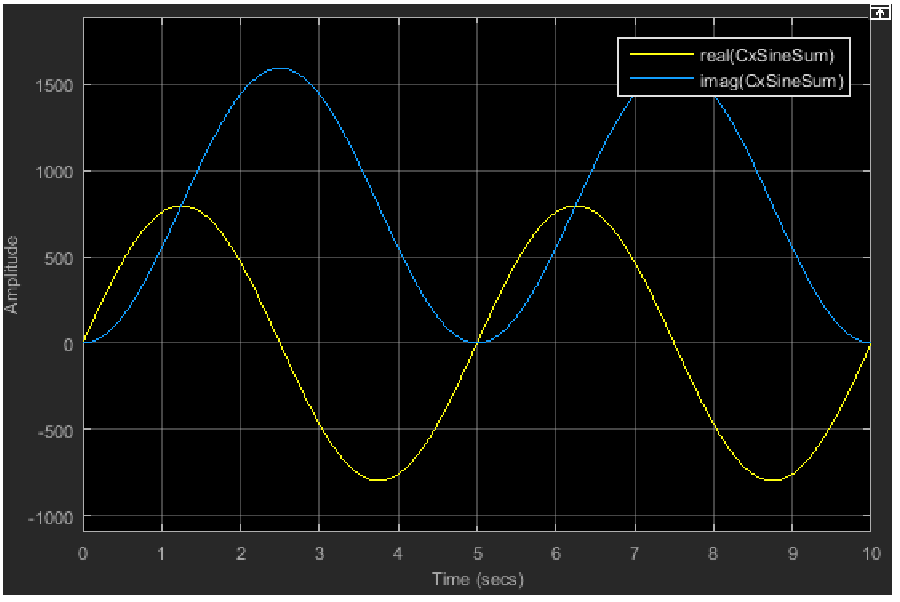
To change the appearance of any channel of any input signal in the scope window, from the menu, select **View > Style**.

Show grid

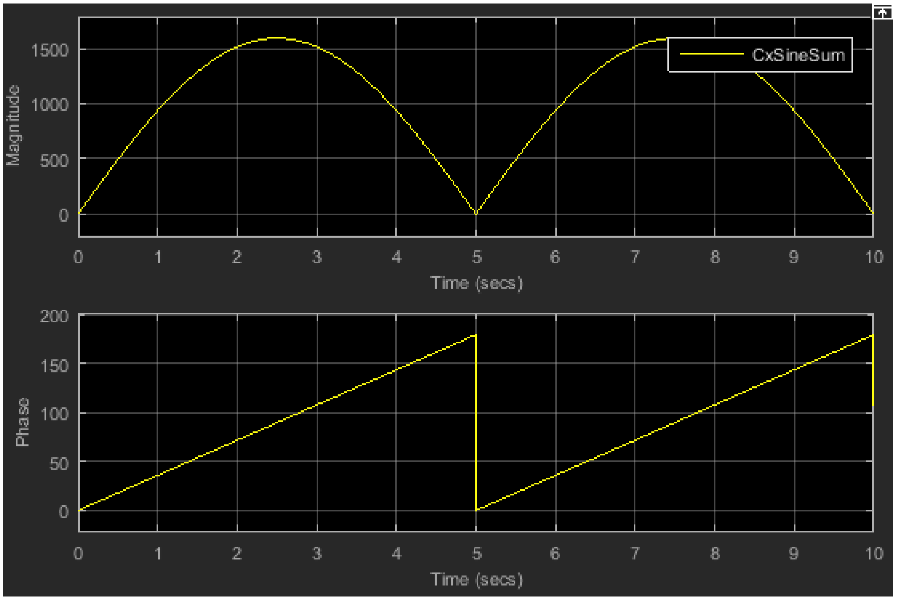
When you select this check box, a grid appears in the display of the scope figure. To hide the grid, clear this check box. Tunable (Simulink)

Plot signals as magnitude and phase

When you select this check box, the scope splits the display into a magnitude plot and a phase plot. By default, this check box is cleared. If the input signal has complex values, the scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes, as shown in the following figure.



Selecting this check box and clicking the **Apply** or **OK** button changes the display. The magnitude of the input signal appears on the top axes and its phase, in degrees, appears on the bottom axes. See the following figure.



This feature is useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box returns the absolute value of the signal for the magnitude. The phase is 0 degrees for nonnegative input and 180 degrees for negative input. Tunable (Simulink)

Y-limits (Minimum)

Specify the minimum value of the *y*-axis. Tunable (Simulink)

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this property always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a minimum value of -180 degrees.

Y-limits (Maximum)

Specify the maximum value of the *y*-axis. Tunable (Simulink)

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this property always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a maximum value of 180 degrees.

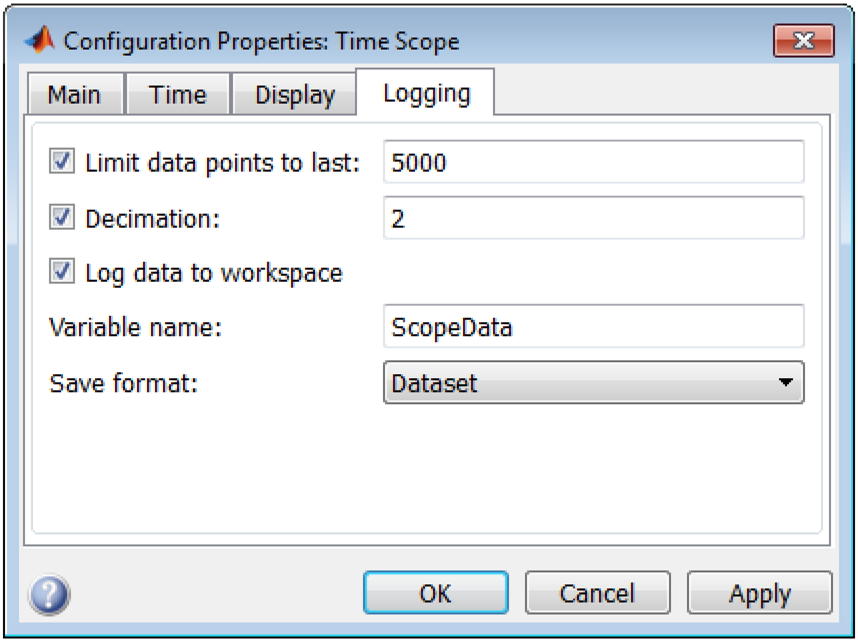
Y-label

Specify the text for the scope to display to the left of the *y*-axis. Tunable (Simulink)

This property becomes invisible when you select the **Plot signal(s) as magnitude and phase** check box. When you enable that property, the *y*-axis label always appears as `Magnitude` on the top axes and `Phase` on the bottom axes.

Logging Pane

The **Logging** pane of the Configuration Properties dialog box appears as follows.



Limit data points to last

Specify the number of data points a scope collects. The scope relies on its data history for zooming and auto-scaling operations. If the number of data points is limited to 1,000 and the simulation generates 2,000 data points, only the last 1,000 are available for regenerating the display.

Note: If you do not select the **Limit data points to last** check box, the memory consumed by MATLAB steadily increases as the simulation stores the entire simulation history. An out of memory error can occur. Memory is limited by available memory on your system.

For simulations with the **Stop time** set to `inf`, always select the **Limit data points to last** check box and enter a value.

The default setting is unchecked, so that all data is saved. When checked, the default is the last `5000` data points.

Decimation

When you select this check box, the scope logs every N^{th} data point, where N is the decimation factor you specify.

The default setting is unchecked, so that logged data is not decimated. When checked, the default decimation rate is `2`.

Log data to workspace

When you select this check box, the scope logs data in the format you select in **Save format**.

The default setting is unchecked and no data is logged.

Variable name

Specify as a character vector the name of the variable in the MATLAB workspace to which the scope logs data. Any existing variable is overwritten.

Save format

Select the format in which to save logged data. Unless otherwise noted, you can save logged data for single- and multi-port data, sample-based and frame-based data, variable-size data, MAT-file logging, and external mode archiving.

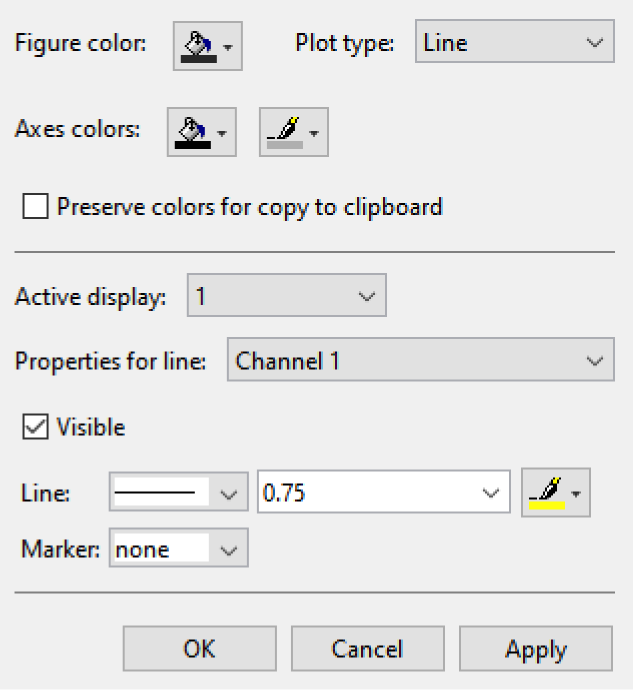
Note: If a Scope is in a disabled subsystem, you may see discontinuities in the Scope display. Data logged to the workspace does not include any of these discontinuity values.

Valid values for **Save format** are:

- `Structure With Time` — Save logged data as a structure with associated time information to the MATLAB workspace. Structure With Time format does not support multirate data.
- `Structure` — Save logged data as a structure to the MATLAB workspace. Structure format does not support multirate data.
- `Array` — Save logged data as an array with associated time information to the MATLAB workspace. Array format does not support multiport sample-based data, single port or multiport frame-based data, variable-size data, or multirate data.
- `Dataset` — Save logged data as a dataset object to the MATLAB workspace. Dataset format does not support variable-size data, MAT-file logging, or external mode archiving. See Simulink.SimulationData.Dataset for information.

Style Dialog Box

Select **View** > **Style** or the Style button () in the dropdown below the Configuration Properties button to open the Style dialog box. In this dialog box, you can change the figure colors, background axes colors, foreground axes colors, and properties of lines in a display.



Properties

The **Style** dialog box allows you to modify the following properties of the scope figure:

Figure color

Specify the color that you want to apply to the background of the scope figure. By default, the figure color is gray.

Plot type

Specify the type of plot to use. The default setting is `Line`. Valid values for **Plot type** are:

- `Line` — line graph, similar to the `line` or `plot` function.
- `Stairs` — *stair-step* graph, similar to the `stairs` function. Stair-step graphs are useful for drawing time history graphs of digitally sampled data.

- `Stem` — stem graph, similar to the `stem` function.
- `Auto` — displays input signal as a line graph for a continuous signal, stair step graph for a discrete signal, and stem for Simulink message signal.

This property is Tunable (Simulink).

Active display

Specify the active display as an integer to get and set relevant properties. The number of a display corresponds to its column-wise placement index. Set this property to control which display has its axes colors, line properties, marker properties, and visibility changed. Tunable (Simulink)

When you use the Layout option to tile the window into multiple displays, the display highlighted in blue is referred to as the *active display*. The default setting is `1`.

Axes colors

Specify the color that you want to apply to the background of the axes for the active display.

Preserve colors for copy to clipboard

Specify whether or not to use the displayed color of the scope when copying.

When you select **File** > **Copy to Clipboard**, the software changes the color of the scope to be printer friendly (white background, visible lines). If you want to copy and paste the scope with the colors displayed, select this check box.

Default: Off

Properties for line

Specify the signal for which you want to modify the visibility, line properties, and marker properties.

Visible

Specify whether the selected signal on the active display is visible. If you clear this check box, the line disappears.

Line

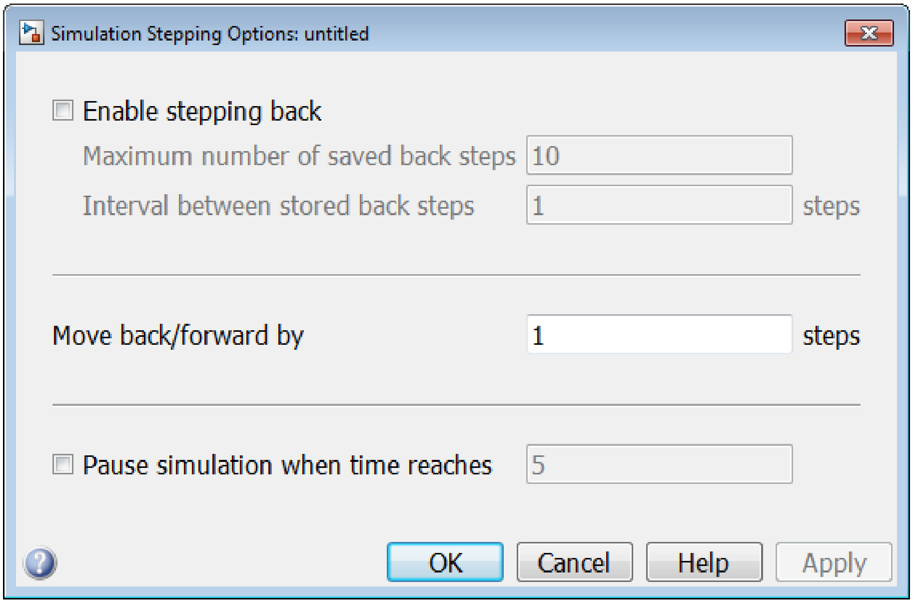
Specify the line style, line width, and line color for the selected signal on the active display.

Marker

Specify marks for the selected signal on the active display to show at data points. This property is similar to the `Marker` property for plot objects. Choose any of the marker symbols from the dropdown list.

Stepping Options

Select **Simulation** > **Stepping Options** to open the Simulation Stepping Options dialog box. If stepping back is disabled, in the Time Scope toolbar, click the step back  button. This dialog box lets you pause the simulation at a specified time, enable stepping back, and specify options for stepping back. You can also modify the number of steps by which to step forward or backward.



The Simulation Stepping Options dialog box is not unique to Time Scope; it can also be launched from any Simulink model. To open this dialog box from any Simulink model, select **Simulation** > **Stepping Options**. For more information, see "Simulation Stepping Options" (Simulink).

Enable stepping back

Select this check box to enable the scope to take steps back in time. When selected, the scope enables the step back button ( ) on the simulation toolbar.

Maximum number of saved back steps

Specify the maximum number of back steps that the scope saves in memory. To maximize simulation speed, keep the value for this property small. The default setting is 10.

Interval between stored back steps

Specify the number of steps between back steps that the scope saves in memory for stepping backward. Set this property to a larger number to increase the time span of a back step without increasing the amount of memory used. The default setting is 1.

Move back/forward by

Specify the number of steps forward or backward that the Scope progresses when you click the step forward ( ) and step back ( ) buttons. The default setting is 1.

Pause simulation when time reaches

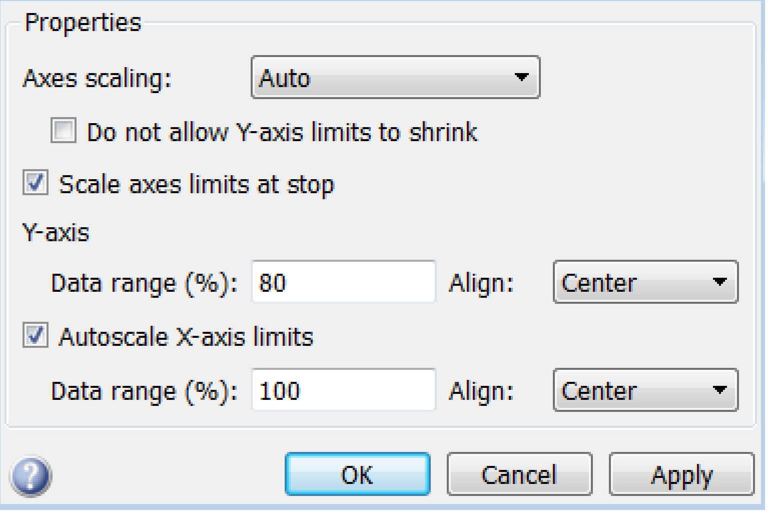
Select this check box to enable the Scope to pause the simulation when it reaches a specified time and then, specify the time at which you want the scope to pause.

Tools—Axes Scaling Properties

Select **Tools** > **Axes Scaling Properties** to open the Axes Scaling Properties dialog box. This dialog box provides you with the ability to zoom automatically in on and out of your data, and to scale the axes of the scope.

Properties

The Tools—Axes Scaling Properties dialog box appears as follows.



Axes scaling

Specify when the scope automatically scales the axes. You can select one of the following options:

- `Manual` — When you select this option, the scope does not automatically scale the axes. You can manually scale the axes in any of the following ways:
  - Select **Tools** > **Axes Scaling Properties**.
  - Press one of the **Scale Axis Limits** toolbar buttons.
  - When the scope figure is the active window, press **Ctrl** and **A** simultaneously.
- `Auto` — When you select this option, the scope scales the axes as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** check box.
- `After N Updates` — Selecting this option causes the scope to scale the axes after a specified number of updates. This option is useful and more efficient when your scope display starts with one axis scale, but quickly reaches a different steady state axis scale. Selecting this option shows the **Number of updates** edit box.

By default, this property is set to `Auto`. This property is Tunable (Simulink).

Do not allow Y-axis limits to shrink

When you select this property, the *y*-axis is allowed only to grow during axes scaling operations. If you clear this check box, the *y*-axis or color limits may shrink during axes scaling operations.

This property appears only when you select `Auto` for the **Axis scaling** property. When you set the **Axes scaling** property to `Manual` or `After N Updates`, the *y*-axis or color limits are allowed to shrink. Tunable (Simulink).

Number of updates

Specify as a positive integer the number of updates after which to scale the axes. This property appears only when you select `After N Updates` for the **Axes scaling** property. Tunable (Simulink).

Scale axes limits at stop

Select this check box to scale the axes when the simulation stops. The *y*-axis is always scaled. The *x*-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

Y-axis Data range (%)

Set the percentage of the *y*-axis that the scope uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the Scope scales the *y*-axis limits such that your data uses the entire *y*-axis range. If you then set this property to `30`, the scope increases the *y*-axis range such that your data uses only 30% of the *y*-axis range. Tunable (Simulink).

Y-axis Align

Specify where the scope aligns your data along the *y*-axis when it scales the axes. You can select `Top`, `Center`, or `Bottom`. Tunable (Simulink).

Autoscale X-axis limits

Check this box to allow the scope to scale the *x*-axis limits when it scales the axes. If **Axes scaling** is set to `Auto`, checking **Autoscale X-axis limits** only scales the data currently within the axes, not the entire signal in the data buffer. If **Autoscale X-axis limits** is on and the resulting axis is greater than the span of the scope, trigger position markers will not be displayed. Triggers are controlled using the Trigger Measurements panel. Tunable (Simulink).

X-axis Data range (%)

Set the percentage of the *x*-axis that the scope uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the scope scales the *x*-axis limits such that your data uses the entire*x*-axis range. If you then set this property to `30`, the scope increases the *x*-axis range such that your data uses only 30% of the *x*-axis range. Use the *x*-axis **Align** property to specify data placement along the *x*-axis.

This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

X-axis Align

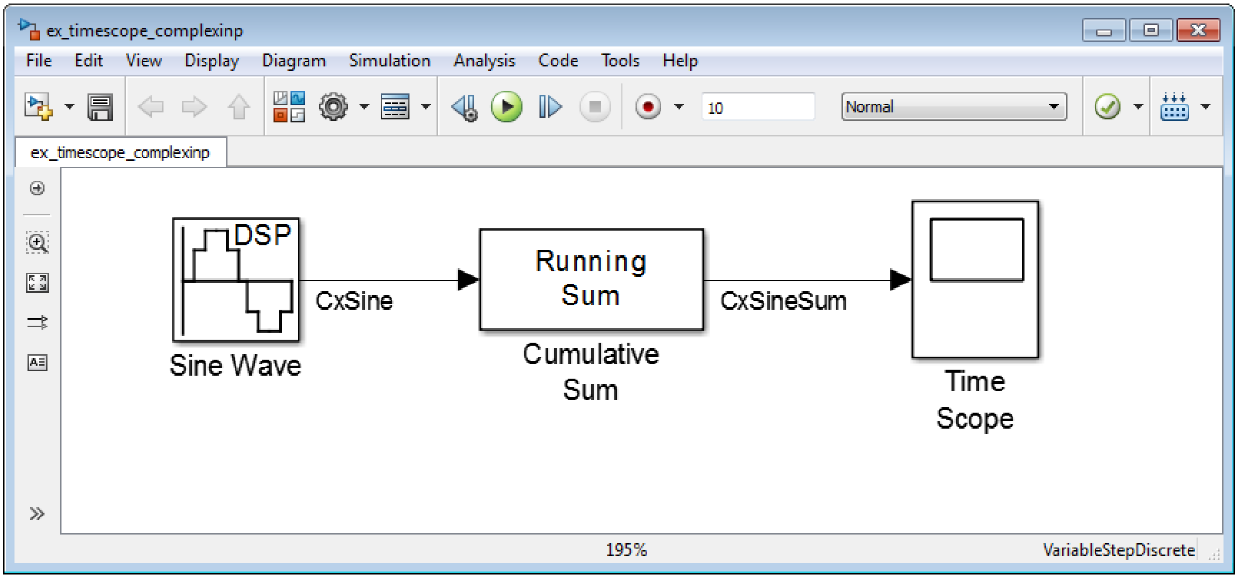
Specify how the scope aligns your data along the *x*-axis: `Left`, `Center`, or `Right`. This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

Examples

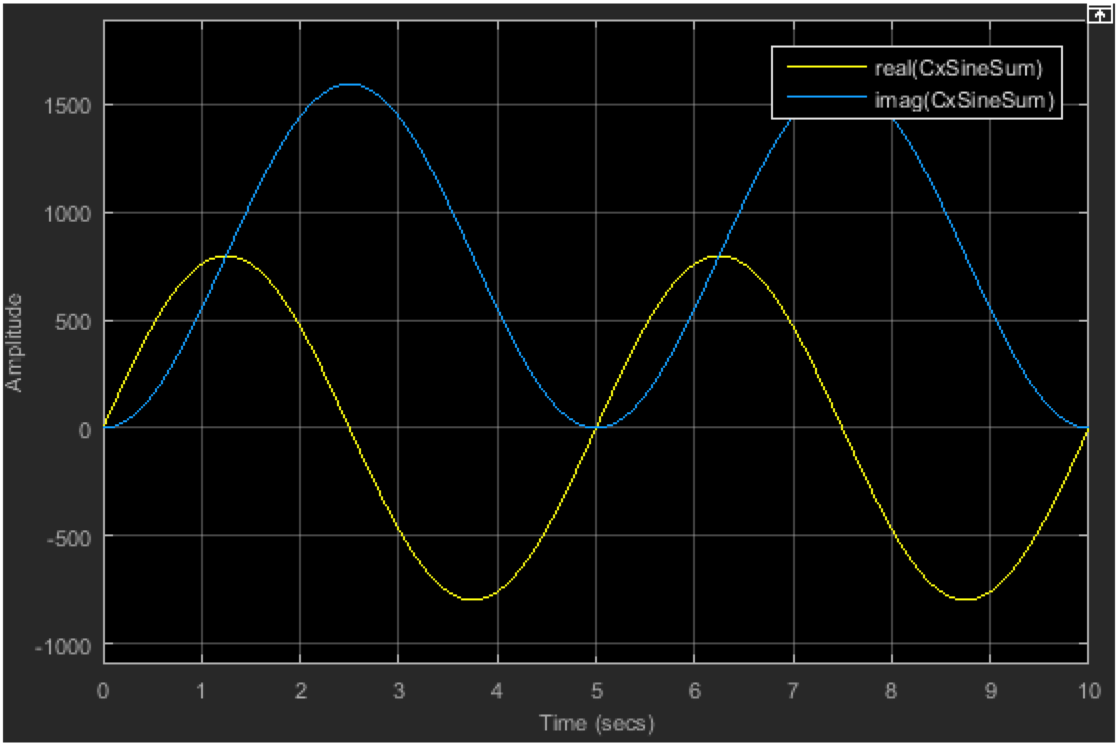
The first three examples illustrate how to use the Time Scope block to view various input signals in the time domain. The remaining examples demonstrate how to use the **Measurements Panels** in the Time Scope figure to glean information about the input signals.

Display Complex-Valued Input Signal

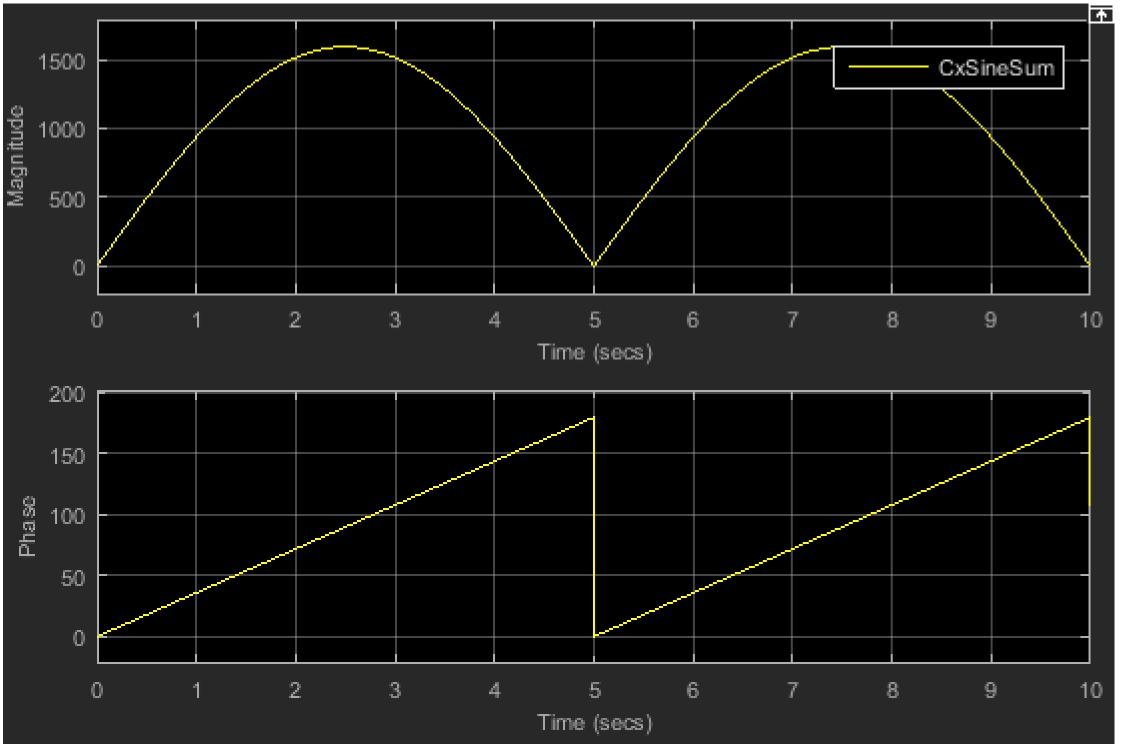
At the MATLAB® command prompt, type ex_timescope_complexinp to open the example model. The following Simulink® model appears.



By default, when the input is a complex-valued signal, Time Scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes within the same active display. Run your model to see the time domain output, as shown in the following figure.

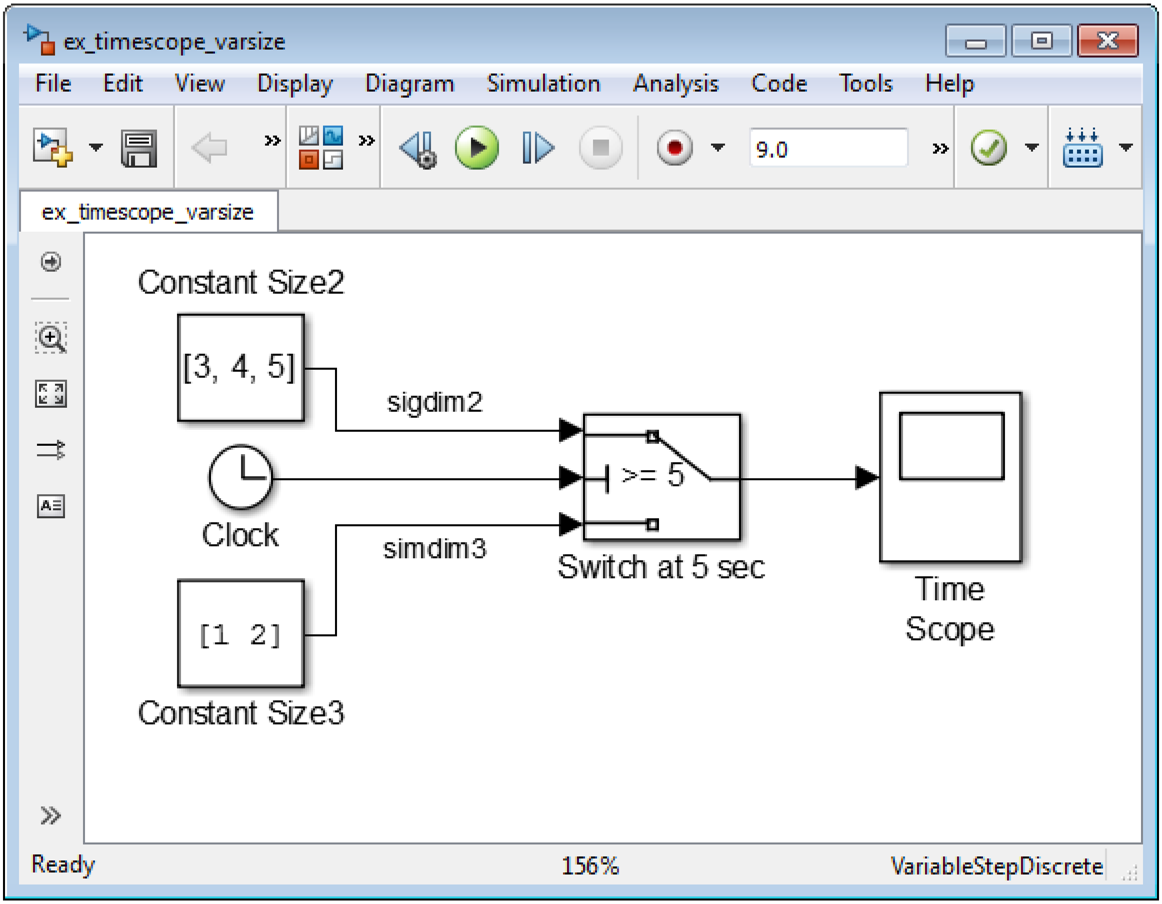


The Configuration Properties dialog box controls the visual configuration settings of the Scope displays. From the Scope menu, select **View** > **Configuration Properties** to open this dialog box. Go to the Display tab. Selecting the **Plot signal(s) as magnitude and phase** check box specifies the Scope to plot the magnitude and phase of the input signal. The magnitude and phase appear on two separate axes within the same active display. After you select this check box, click **OK**. The active display shows the magnitude of the input signal on the top axes. The signal phase, in degrees, appears on the bottom axes.

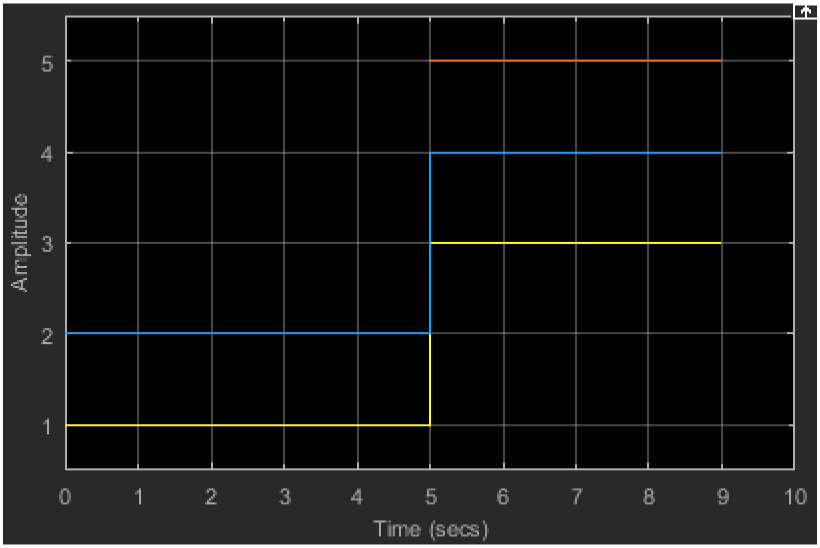


Display Input Signal of Changing Size

At the MATLAB® command prompt, type ex_timescope_varsize to open the example model. The following Simulink® model appears.



In this example, the size of the input signal to the Time Scope block changes as the simulation progresses. When the simulation time is less than 5 seconds, Time Scope plots the signal connected to the third input port of the Switch block, which has signal dimensions 1 by 2. After 5 seconds, Time Scope plots the signal connected to the first input port of the Switch block, which has signal dimensions 1 by 3. Run your model to see the time domain output.

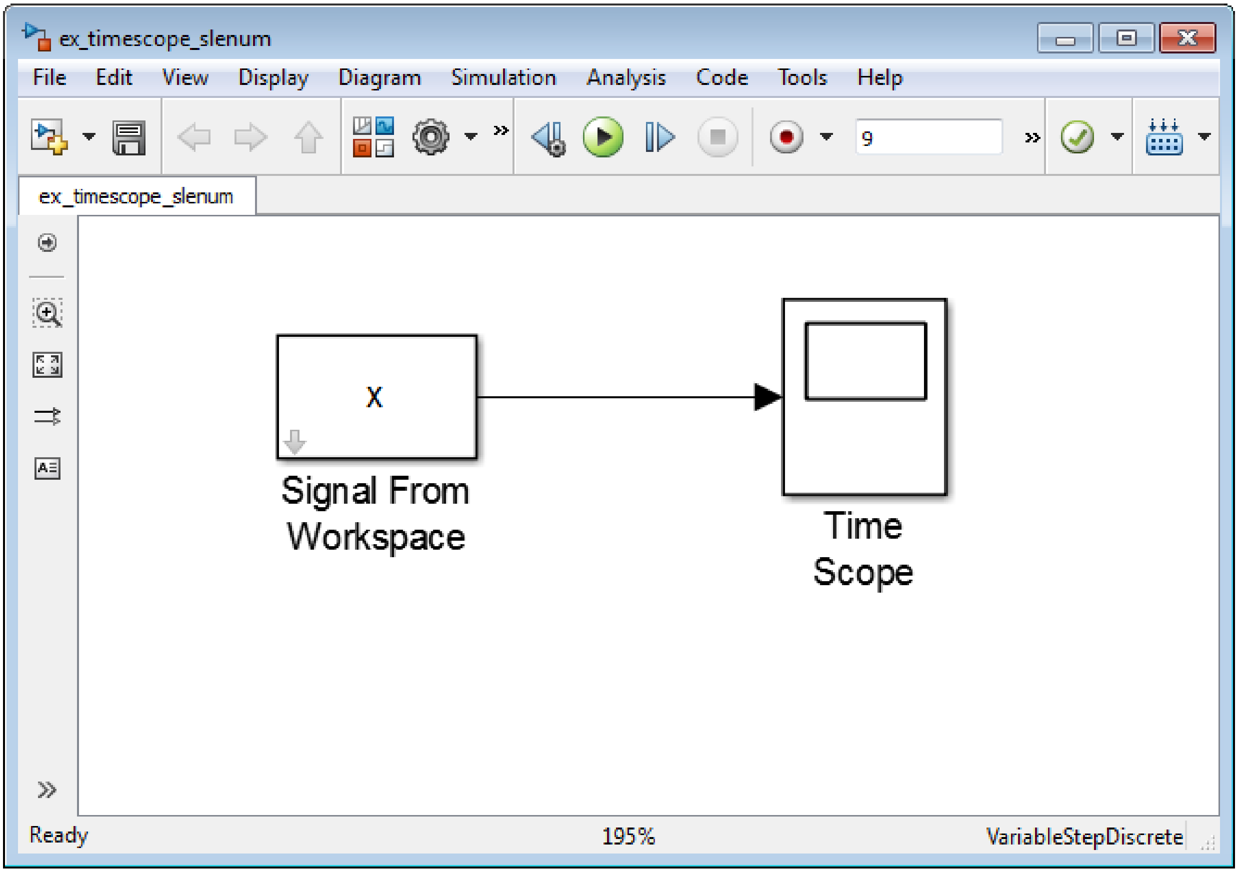


As you can see in the figure, the third line on the display, colored red, appears only after 5 seconds.

Display Simulink Enumeration Input Signal

This example shows how to use the Time Scope to display an enumerated input signal.

At the MATLAB® command prompt, type ex_timescope_slenum to open the example model. The following Simulink® model appears.

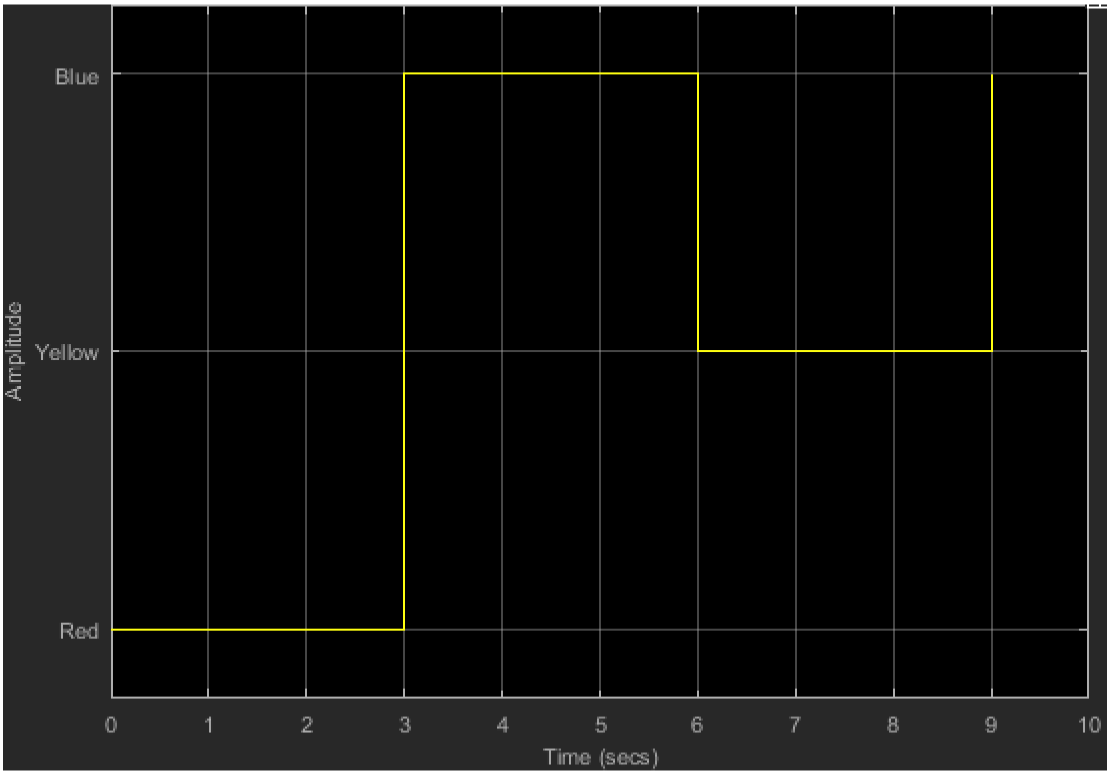


In this example, Simulink® imports the variable `x`, which is a Simulink® enumeration data type, from the MATLAB® workspace. This variable is created when the model loads because the commands that construct it reside in the model pre-load function. To view these commands, in the Simulink® menu, select `File > Model Properties > Model Properties`. The following lines of MATLAB® code appear when you click the `Callbacks` tab.

```
if ~exist('BasicColors','class')
    Simulink.defineIntEnumType('BasicColors', ...
        {'Red', 'Yellow', 'Blue'}, ...
        [0;1;2], ...
        'Description', 'Basic colors', ...
        'DefaultValue', 'Blue', ...
```

```
        'AddClassNameToEnumNames', true);
end
x = [BasicColors(0), BasicColors(2), BasicColors(1)];
```

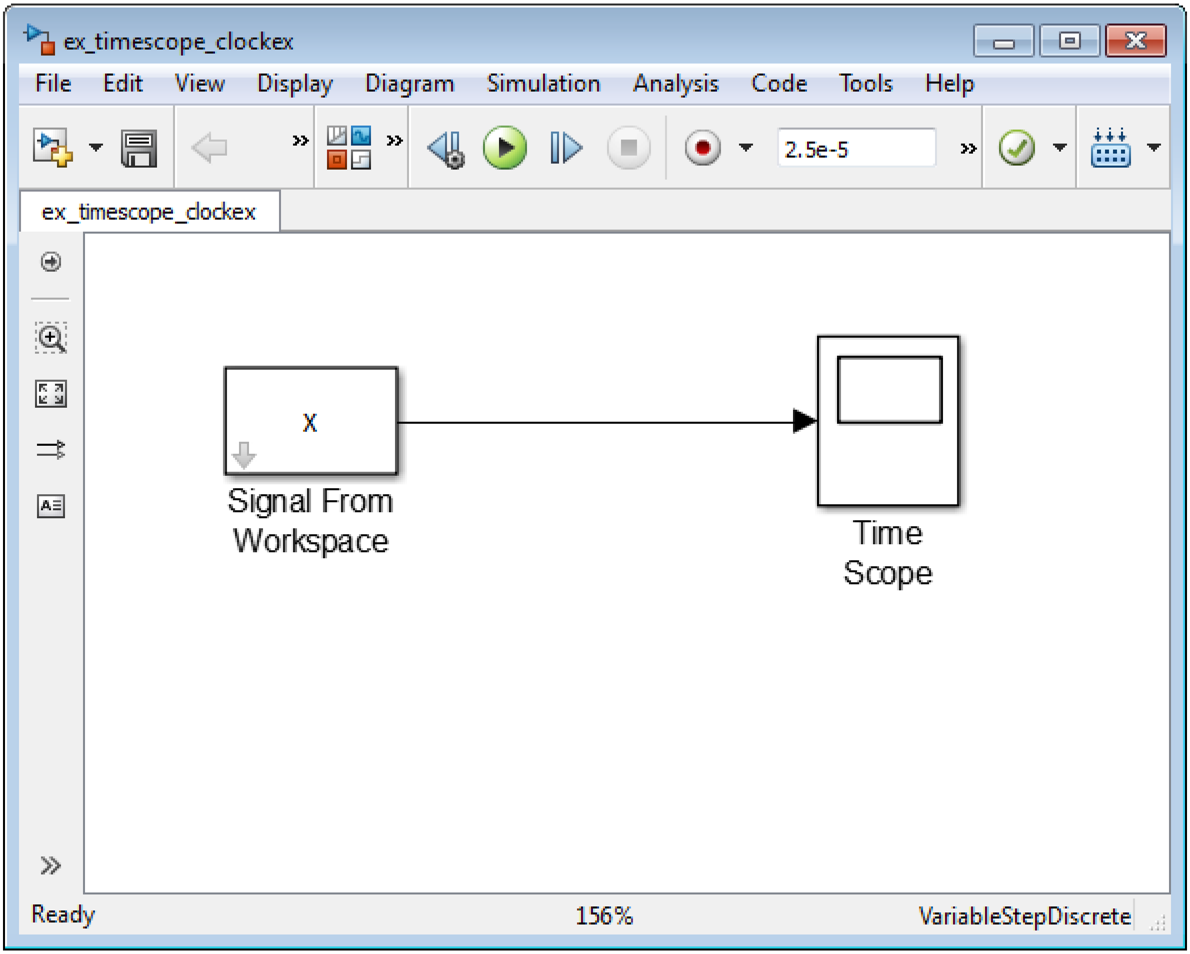
The Signal from Workspace block has a Sample time of 3 seconds. Thus, the input signal changes to the next value in vector x every 3 seconds. Run your model to see the time domain output, as shown in the following figure.



Use Bilevel Measurements Panel with Clock Input Signal

This example shows how to use the Time Scope Bilevel Measurements panel.

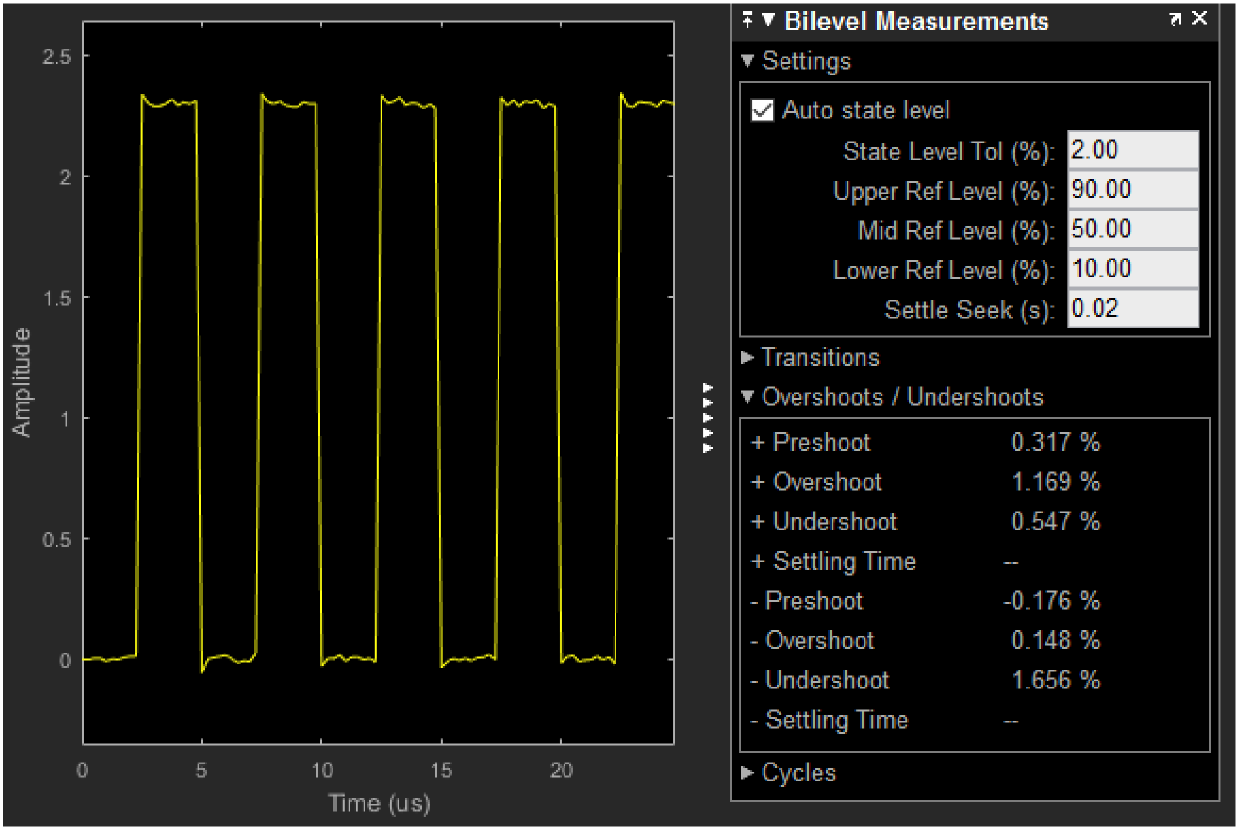
At the MATLAB® command prompt, type ex_timescope_clockex to open the example model. The following Simulink® model appears.



In this example, Simulink® imports the variable `x`, from the MATLAB® workspace. This variable is created when the model loads because the commands that construct it reside in the model Preload function. To view these commands, in the Simulink® menu, select `File > Model Properties > Model Properties`. The Model Properties dialog box appears. Click the `Callbacks` tab. The following lines of MATLAB® code appear.

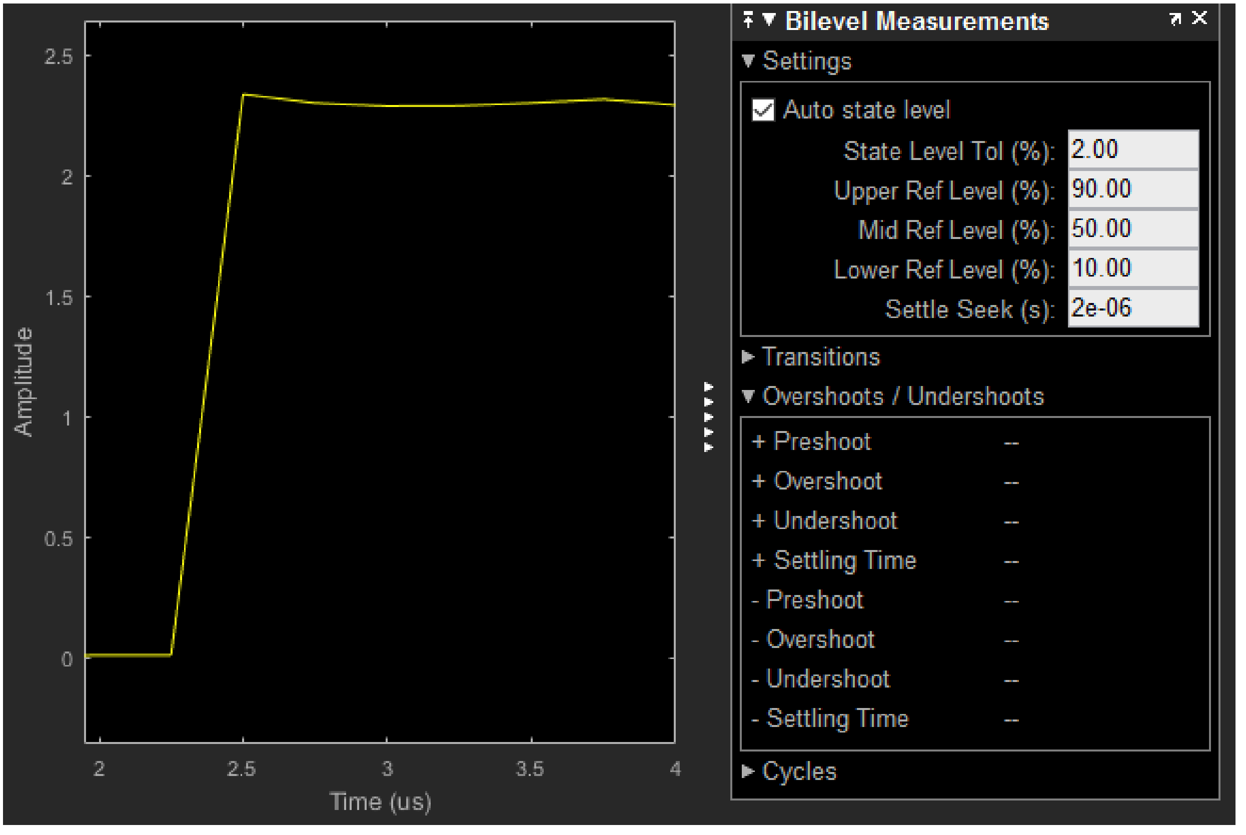
```
load clockex;
ts = t(2)-t(1);
```

Run your model to see the time domain output. To show the `Bilevel Measurements` panel, in the Time Scope menu, select `Tools > Measurements > Bilevel Measurements`. Collapse the `Transitions` section, click the arrow next to the section label. Expand the `Settings` and `Overshoots/Undershoots` sections, click the arrow next to each label. The Time Scope appears as shown in the following figure.



The value for the rising edge `Settling Time` parameter is not displayed initially because the default value for the `Settle Seek` parameter is too large for this example. Here, the settle seek time is longer than the entire simulation duration. Enter a value for `Settle seek` of 2e-6 and press the Enter key. The Time Scope now displays a rising edge settling time value of 118.392 ns.

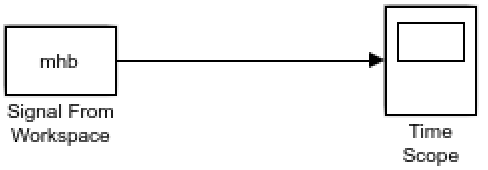
The settling time value displayed is the statistical average of the settling times for all five rising edges. To show the settling time for one rising edge, zoom in on that transition. In the Time Scope toolbar, click the Zoom X button.. Click the display near a value of 2 microseconds on the time-axis. Drag to the right and release near a value of 4 microseconds on the time-axis. The Time Scope updates the rising edge settling time value to reflect the new time window, as shown in the following figure.



Use Peak Finder to Find Heart Rate from ECG Input

This example shows how to use the Time Scope Peak Finder panel to measure the heart rate from an ECG.

This model emulates an ECG heart rate.



To emulate a heart beat, the model imports the variable `mhb` from the MATLAB® workspace. This variable is created when the model loads because the commands that construct it are in the model Preload function. To view these commands, in the Simulink® menu, select **File > Model Properties > Model Properties**. The Model Properties dialog box appears. Click the **Callbacks** tab. The following lines of MATLAB® code appear.

```
x1 = 3.5*ecg(2700);
y1 = sgolayfilt(kron(ones(1,13),x1),0,21);
n = (1:30000)';
del = round(2700*rand(1));
mhb = y1(n + del);
ts = 0.00025;
```

`ecg` is a custom fuction that helps generate the heartbeat signal.

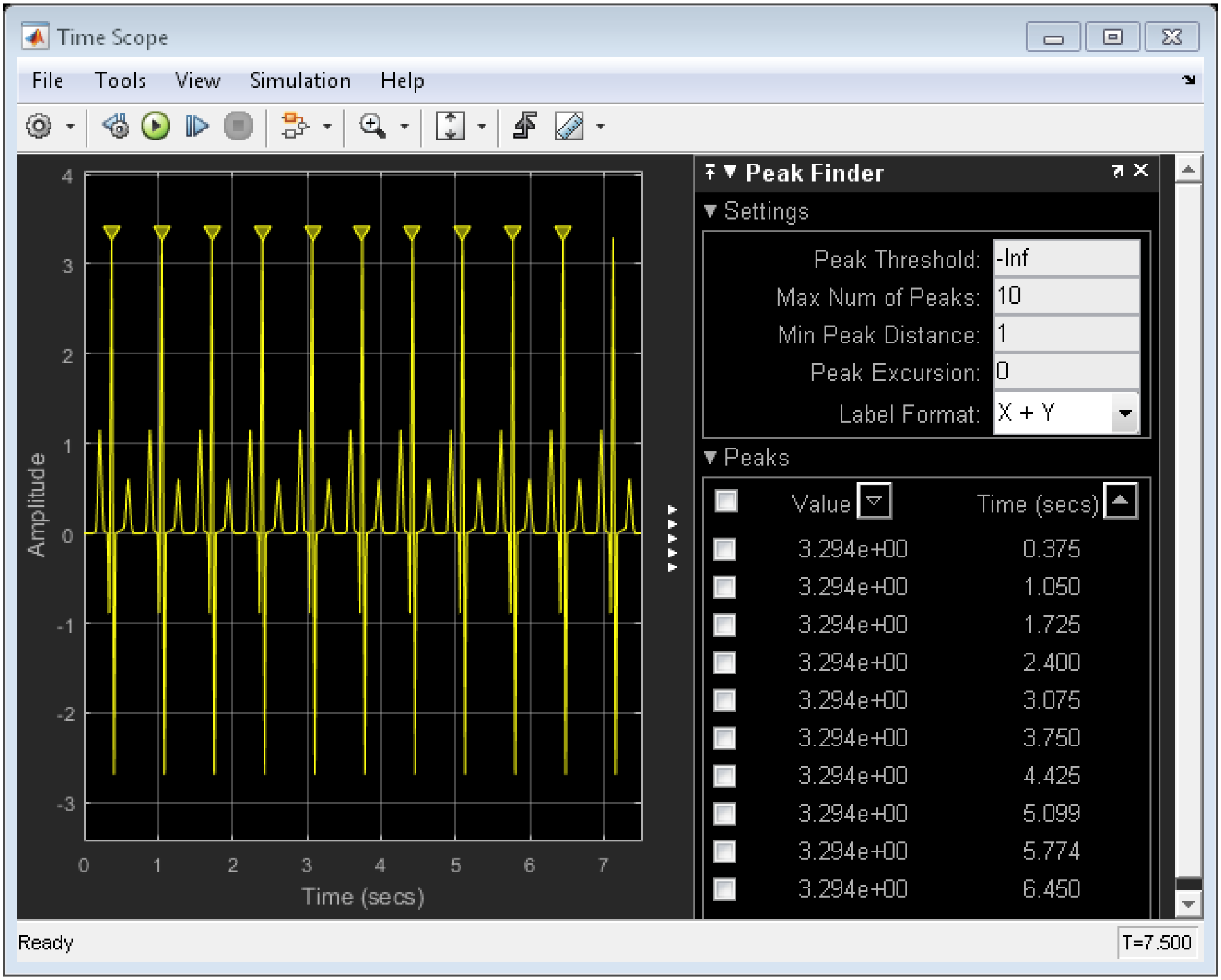
```
function x = ecg(L)
a0 = [0,  1, 40,  1,   0, -34, 118, -99,   0,   2,  21,   2,   0,   0,   0];
d0 = [0, 27, 59, 91, 131, 141, 163, 185, 195, 275, 307, 339, 357, 390, 440];
a = a0 / max(a0);
d = round(d0 * L / d0(15));
d(15) = L;
for i = 1:14
    m = d(i) : d(i+1) - 1;
    slope = (a(i+1) - a(i)) / (d(i+1) - d(i));
    x(m+1) = a(i) + slope * (m - d(i));
end
```

Run your model to see the time domain output.

1 To show the **Peak Finder** panel, in the Time Scope menu, select **Tools > Measurements > Peak Finder**.

2 To expand the **Settings** section, click the arrow next to the label.

3 Enter 10 for `Max Num of Peaks`. The Time Scope Peaks section now displays a list of 10 peak amplitude values, and the times at which they occur.



The list of peak values show a constant time difference of 0.675 seconds between each heartbeat. Therefore, the heart rate of the ECG signal is 88.89 beats per min (bpm).

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Simulink enumerations

You can use nonvirtual buses and array of buses as inputs. For details, see “Bus Support” on page 2-1719.

Supported Simulation Modes

You can use the scope block in models running the following supported simulation modes.

Mode	Supported	Notes and Limitations
Normal	Yes	
Accelerator	Yes	
Rapid Accelerator	Yes	You can use Rapid Accelerator mode as a method to increase the execution speed of your Simulink model. Rapid Accelerator mode creates an executable that includes the solver and model methods. This executable resides outside MATLAB and Simulink. Rapid Accelerator mode uses External mode to communicate with Simulink. For more information about Rapid Accelerator mode, see “Acceleration” (Simulink).
PIL	No	

Mode	Supported	Notes and Limitations
SIL	No	
External	Yes	You can use External mode to tune block parameters in real time and view block outputs in many types of blocks and subsystems. External mode establishes communication between a host system, where the Simulink environment resides, and a target system, where the executable runs after code generation and the build process. For more information about External mode, see “Set Up and Use Host/Target Communication Channel” (Simulink Coder). The scope does not support data archiving. See “Set External Mode Data Archiving Parameters” (Simulink Desktop Real-Time).

For more information about these modes, see “How Acceleration Modes Work” (Simulink).

To log bus signals with a Time Scope block, set the **Save format** block parameter to `Dataset`.

Bus Support

You can connect nonvirtual bus and array of buses signals to a Time Scope block. To display the bus signals, use normal or accelerator simulation mode. The Time Scope block displays each bus element signal, in the order the elements appear in the bus, from the top to the bottom. Nested bus elements are flattened.

To log bus signals with a Time Scope block, set the **Save format** block parameter to `Dataset`.

See Also

dsp.TimeScope | `Array Plot` | `sptool` | Scope | `Spectrum Analyzer`

Topics

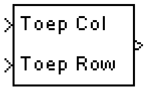
“Display Time-Domain Data”

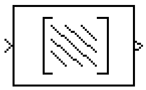
“Scope Blocks and Scope Viewer Overview” (Simulink)

Introduced before R2006a

Toeplitz

Generate matrix with Toeplitz symmetry





Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

`dspmtrx3`

Description

The Toeplitz block generates a Toeplitz matrix from inputs defining the first column and first row. The top input (`Col`) is a vector containing the values to be placed in the first *column* of the matrix, and the bottom input (`Row`) is a vector containing the values to be placed in the first *row* of the matrix.

```
y = toeplitz(Col,Row)       % Equivalent MATLAB code
```

The other elements of the matrix obey the relationship

```
y(i,j) = y(i-1,j-1)
```

and the output has dimension `[length(Col) length(Row)]`. The y(1,1) element is inherited from the `Col` input. For example, the following inputs

```
Col = [1 2 3 4 5]
Row = [7 7 3 3 2 1 3]
```

produce the Toeplitz matrix

$$\begin{bmatrix} 1 & 7 & 3 & 3 & 2 & 1 & 3 \\ 2 & 1 & 7 & 3 & 3 & 2 & 1 \\ 3 & 2 & 1 & 7 & 3 & 3 & 2 \\ 4 & 3 & 2 & 1 & 7 & 3 & 3 \\ 5 & 4 & 3 & 2 & 1 & 7 & 3 \end{bmatrix}$$

When you select the **Symmetric** check box, the block generates a symmetric (Hermitian) Toeplitz matrix from a single input, `u`, defining both the first row and first column of the matrix.

```
y = toeplitz(u)      % Equivalent MATLAB code
```

The output has dimension `[length(u) length(u)]`. For example, the Toeplitz matrix generated from the input vector `[1 2 3 4]` is

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 2 & 3 \\ 3 & 2 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{bmatrix}$$

The Toeplitz block supports real and complex floating-point and fixed-point inputs.

Parameters

Symmetric

When selected, enables the single-input configuration for symmetric Toeplitz matrix output.

Saturate on integer overflow

When you generate a symmetric Toeplitz matrix with this block, if the input vector is complex, the output is a symmetric Hermitian matrix whose elements satisfy the relationship

$$y(i,j) = \mathrm{conj}(y(j,i))$$

For fixed-point signals the conjugate operation could result in an overflow. When you select this parameter, overflows saturate. This parameter is only visible with the **Symmetric** parameter is selected. This parameter is ignored for floating-point signals.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers (real signals only)
Toep Col	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Toep Row	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

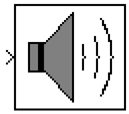
See Also

`toeplitz`	MATLAB

Introduced before R2006a

To Audio Device

Write audio data to computer's audio device



Note: The To Audio Device block will be removed in a future release. Existing instances of the block continue to run. For new models, use the `Audio Device Writer` block instead.

Library

Sinks

`dspsnks4`

Description

The To Audio Device block sends audio data to your computer's audio device. This block is not supported for use with the Simulink Model block.

Use the **Device** parameter to specify the device to which you want to send the audio data. This parameter is automatically populated based on the audio devices installed on your system. If you plug or unplug an audio device from your system, type `clear mex` at the MATLAB command prompt to update the list.

Select the **Inherit sample rate from input** check box if you want the block to inherit the sample rate of the audio signal from the input to the block. If you clear this check box, the **Sample rate (Hz)** parameter appears on the block. Use this parameter to specify the number of samples per second in the signal.

The range of supported audio device sample rates and data type formats, depend on both the sound card and the API which is chosen for the sound card.

Use the **Device data type** to specify the data type of the audio data that is sent to the device. You can choose:

- `8-bit integer`
- `16-bit integer`
- `24-bit integer`
- `32-bit float`
- `Determine from input data type`

If you choose `Determine from input data type`, the following table summarizes the block's behavior.

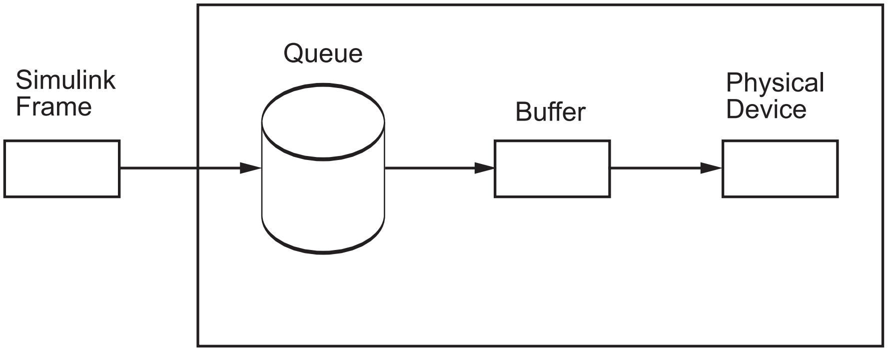
Input Data Type	Device Data Type
Double-precision floating point or single-precision floating point	32-bit floating point
32-bit integer	24-bit integer
16-bit integer	16-bit integer
8-bit integer	8-bit integer

If you choose `Determine from input data type` and the device does not support the input data type, the block uses the next lowest-precision data type supported by the device.

The generated code for this block relies on prebuilt .dll files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra .dll files when doing so. The packNGo function creates a single zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

Buffering

The To Audio Device block buffers the data from a Simulink signal using the process illustrated by the following figure.



To Audio Device Block

1 At the start of the simulation, the queue is filled with silence. Specify the size of this queue using the **Queue duration (seconds)** parameter. As Simulink runs, the block appends Simulink frames to the bottom of the queue.

2 At each time step, the blocks sends a buffer of samples from the top of the queue to the audio device. Select the **Automatically determine buffer size** check box to allow the block to use a conservative buffer size. See the `From Audio Device` block reference page for the equation the block uses to calculate this buffer size. If you clear this check box, the **Buffer size (samples)** parameter appears on the block. Use this parameter to specify the size of the buffer in samples.

3 The block writes the buffer of audio data to the device. If the queue did not contain enough data to completely fill the buffer, the block fills the remaining portion of the buffer with zeros. This data has a the data type specified by the **Device data type** parameter.

When the simulation throughput rate is lower than the hardware throughput rate, the queue, which is initially full, becomes empty. If the queue is empty, the block sends zeros (silence) to the audio device. You can monitor inserted zeroes using the optional `Underrun` output port. When the simulation throughput rate is higher than the hardware throughput rate, the To Audio Device block waits to write data to the queue.

To minimize the chance of dropouts, the block checks to make sure the queue duration is at least as large as the maximum of the buffer size and the frame size. If it is not, the queue duration is automatically set to this maximum value.

Channel Mapping

The term *Channel Mapping* refers to a 1-to-1 mapping that associates channels on the selected audio device to channels of the data. When you play audio, channel mapping allows you to specify which channel of the audio device directs input to a specific channel of audio data. You can specify channel mapping as a vector of output channel indices corresponding to each output channel of data being written. The default value in the **Device Output Channels** parameter is 1:MAXOUTPUTCHANNELS. If you do not select the default mapping, you must specify the **Device Output Channels** parameter in the dialog box.

Example: The selected output audio device contains 8 channels. The data being output has dimensions N×3 (3–channel data). You want the output to be redirected as follows:

- First data channel to Audio Device channel 3
- Second data channel to Audio Device channel 1
- Third data channel to Audio Device channel 8

Thus, you would specify the **Device Output Channels** as [3 1 8].

Troubleshooting

Not Keeping Up in Real Time

When Simulink cannot keep up with an audio device that is operating in real time, the queue becomes empty and gaps occur in the audio data that the block sends to the device. Select the **Output number of samples by which the queue was underrun** check box to add an output port indicating when the queue was empty. Here are several ways to deal with this situation:

- *Increase the queue duration.*

  The **Queue duration (seconds)** parameter specifies the duration of the signal, in seconds, that can be buffered during the simulation. This is the maximum length of time that the block's data supply can lag the hardware's data demand.
- *Increase the buffer size.*

  The size of the buffer processed in each interrupt from the audio device affects the performance of your model. If the buffer is too small, a large portion of hardware resources are used to write data to the device. If the buffer is too big, Simulink must wait for the device to empty the buffer before it can write the data to the queue, which introduces latency.

- *Increase the simulation throughput rate.*

  Two useful methods for improving simulation throughput rates are increasing the signal frame size and compiling the simulation into native code:

  - Increase frame sizes and use frame-based processing throughout the model to reduce the amount of block-to-block communication overhead. This can increase throughput rates in many cases. However, larger frame sizes generally result in greater model latency due to initial buffering operations.
  - Generate executable code with Simulink Coder code generation software. Native code runs much faster than Simulink and should provide rates adequate for real-time audio processing.

Other ways to improve throughput rates include simplifying the model and running the simulation on a faster PC processor. For other ideas on improving simulation performance, see “Delay and Latency” and “Performance” (Simulink).

Running an Executable Outside MATLAB

To run your generated standalone executable application in Shell, you need to set your environment to the following:

Platform	Command

Mac	`setenv DYLD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/maci64 (csh/tcsh)`
	
	`export DYLD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/maci64 (Bash)`

Linux	`setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/glnxa64 (csh/tcsh)`
	
	`export LD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/glnxa64 (Bash)`

Platform	Command
Windows	`set PATH = $MATLABROOT\bin\win64;%PATH%`

Channel-to-Speaker Mapping on Windows Operating Systems

The To Audio Device and From Audio Device blocks can support multiple channels. On Windows operating systems, the channel-to-speaker mapping is defined as listed below. This mapping only applies when your sound card is properly configured and capable of receiving the audio data you send. If the number of channels on the card does not match the number of channels on the block, or if you specify a data type for the **Device data type** parameter that is not supported by your device, the Windows mixer intervenes to translate from one format to another. If the Windows mixer does intervene, the channel-to-speaker mapping might differ from what is specified here.

- Single channel input — Front center speaker

  On systems with two speakers, the front center channel is split between the right and left speakers.
- Multichannel input — Channels are assigned to speakers as follows:

  - One channel — Front center
  - Two channels — Front left, front right
  - Four channels — Front left, front right, rear left, rear right
  - Six channels — Front left, front right, front center, low frequency, rear left, rear right
  - Eight channels — Front left, front right, front center, low frequency, rear left, rear right, front left center, front right center
  - For all other channel combinations, the channel assignment is dictated by the audio card.

Audio Hardware API

The To Audio Device and From Audio Device blocks use the open-source PortAudio library in order to communicate with the audio hardware on a given computer. The PortAudio library supports a range of API's designed to communicate with the audio hardware on a given platform. The following API choices were made when building the PortAudio library for the DSP System Toolbox product:

- Windows: DirectSound, WDM-KS, ASIO

- Linux: ALSA, OSS
- Mac: CoreAudio

For Windows, the default is DirectSound, for Linux, the default is ALSA, and for Mac there is only one choice.

To determine the audio hardware API currently selected, type the following command in the MATLAB command prompt.

```
getpref('dsp','portaudioHostApi')
```

The output is a scalar indicating the choice of the API.

- 1 — DirectSound
- 3 — ASIO
- 7 — OSS
- 8 — ALSA
- 11 — WDM-KS

To select a particular API, type the following command in the MATLAB command prompt.

```
setpref('dsp','portaudioHostApi',N)
```

where *N* is a scalar. Choose *N* based on the API choice above.

Parameters

Device

Specify which device to send the audio data to.

Inherit sample rate from input

Select this check box if you want the block to inherit the sample rate of the audio signal from the input to the block.

Sample rate (Hz)

Specify the number of samples per second in the signal. This parameter is visible when the **Inherit sample rate from input** check box is cleared.

Device data type

Specify the data type of the audio data sent to the device.

Automatically determine buffer size

Select this check box to allow the block to calculate a conservative buffer size.

Buffer size (samples)

Specify the size of the buffer. This parameter is visible when the **Automatically determine buffer size** check box is cleared.

Queue duration (seconds)

Specify the size of the queue in seconds.

Use default mapping between Data and Device Output Channels

Select this check box to have the default mapping, where the data from the first channel of audio device is sent to the first channel of the input data, data from second channel of audio device is sent to second channel of data and so on. The maximum number of channels in the input data is determined by the **Number of channels** property.

Device Output Channels

Specify the channel mapping. This parameter is visible when the **Use default mapping between Device Input Channels and Data** check box is disabled.

Output number of samples by which the queue was underrun

Select this check box to output the number of zero samples inserted into the audio stream due to queue underrun since the last transfer of a frame to the audio device. You can use this value to debug throughput problems and adjust the queues and buffers in your model. To learn how to improve throughput, see "Troubleshooting" on page 2-1728.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • 32-bit signed integers • 16-bit signed integers • 8-bit unsigned integers
Underrun	32-bit unsigned integer

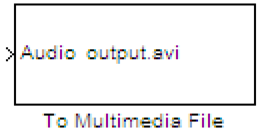
See Also

`From Audio Device`	DSP System Toolbox
`To Multimedia File`	DSP System Toolbox
`audioplayer`	MATLAB
`sound`	MATLAB
dsp.AudioPlayer	System Object

Introduced in R2007b

To Multimedia File

Stream video frames and audio samples to multimedia file



Library

Sinks

`dspsnks4`

Description

The To Multimedia File block writes video frames, audio samples, or both to a multimedia (`.avi, .wav, .wma, .mp4, .ogg, .flac, or .wmv`) file.

You can compress the video frames or audio samples by selecting a compression algorithm. You can connect as many of the input ports as you want. Therefore, you can control the type of video and/or audio the multimedia file receives.

Note This block supports code generation for platforms that have file I/O available. You cannot use this block with Simulink Desktop Real-Time software, because that product does not support file I/O.

This block performs best on platforms with Version 11 or later of Windows Media Player software. This block supports only uncompressed RGB24 AVI files on Linux and Mac platforms.

Windows 7 UAC (User Account Control), may require administrative privileges to encode `WMV` and `WMA` files.

The generated code for this block relies on prebuilt library files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra library files when doing so. The packNGo function creates a single zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

To run an executable file that was generated from a model containing this block, you may need to add precompiled shared library files to your system path. See “Understanding C Code Generation” for details.

Cross-Platform Supported File Formats for Audio Files

Audio files can be of the following formats on all platforms:

- WAV
- FLAC
- OGG
- MPEG4 (only on Windows 7 and Mac OS X)

The default format is WAV. This block supports MPEG-4 AAC audio files on Windows 7, and Mac OS X. You can use both M4A and MP4 extensions. The following platform specific restrictions apply when writing these files:

<table>
<tr><th>Windows 7</th><th>Mac OS X</th></tr>
<tr><td>• Only sample rates of 44100 and 48000 Hz are supported.</td><td rowspan="2">• Only mono or stereo outputs are allowed for MPEG-4 AAC file format. For all other formats, more than two audio output channels are allowed.</td></tr>
<tr><td>• Only mono or stereo outputs are allowed for MPEG-4 AAC file format. For all other formats, more than two audio output channels are allowed.</td></tr>
<tr><td>• The output data is padded on both the front and back of the signal, with extra samples of silence.

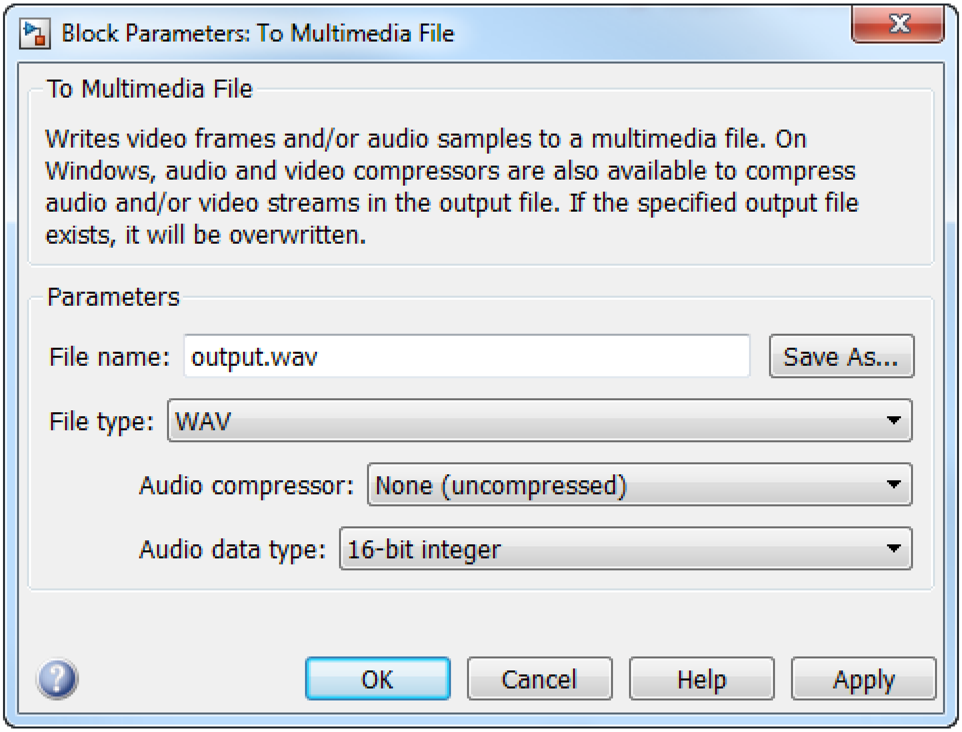
Windows AAC encoder places sharp fade-in and fade-out on audio signal, causing signal to be slightly longer in samples when written to disk.</td><td rowspan="2">• Not all sampling rates are supported, although the Mac Audio Toolbox API do not explicitly specify a restriction.</td></tr>
<tr><td>• A minimum of 1025 samples per channel must be written to the MPEG-4 AAC file.</td></tr>
</table>

Ports

Port	Description
Image	M-by-N-by-3 matrix RGB, Intensity, or YCbCr 4:2:2 signal.
R, G, B	Matrix that represents one plane of the RGB video stream. Inputs to the R, G, or B port must have the same dimensions and data type.
Audio	M-by-N matrix. M is the number of samples in each channel, and N is the number of channels.
Y, Cb, Cr	Matrix that represents one frame of the YCbCr video stream. The Y, Cb, and Cr ports use the following dimensions: Y: $M \text{ x } N$ Cb: $M \text{ x } \frac{N}{2}$ Cr: $M \text{ x } \frac{N}{2}$

Dialog Box

The **Main** pane of the To Multimedia File block dialog appears as follows.



File name

Specify the name of the multimedia file. The block saves the file in your current folder. To specify a different file or location, click the **Save As...** button.

File type

Specify the file type of the multimedia file. You can select `AVI`, `WAV`, `MJ2000`, `WMA`, `WMV`, `MPEG4`, `FLACC`, or `OGG`. By default, the **File type** is set to `WAV`.

Write

Specify whether the block writes video frames, audio samples, or both to the multimedia file. You can select `Video and audio`, `Video only`, or `Audio only`. This parameter is visible only when you set **File type** to `AVI`, `MPEG4`, or `OGG`.

Audio compressor

Select the type of compression algorithm to use to compress the audio data. This compression reduces the size of the multimedia file. Choose `None (uncompressed)` to save uncompressed audio data to the multimedia file.

Note: The other items available in this parameter list are the audio compression algorithms installed on your system. For information about a specific audio compressor, see the documentation for that compressor.

Audio data type

Select the audio data type. You can use the **Audio data type** parameter only for uncompressed wave files.

Video compressor

Select the type of compression algorithm to use to compress the video data. This compression reduces the size of the multimedia file. Choose `None (uncompressed)` to save uncompressed video data to the multimedia file.

Note: The other items available in this parameter list are the video compression algorithms installed on your system. For information about a specific video compressor, see the documentation for that compressor.

Compression Factor (>1)

Specify the compression factor as an integer scalar greater than `1`. This parameter is applicable only when the **File type** is set to `MJ2000` and **Video compressor** is set to `Lossy`. By default, this parameter is set to `10`.

File color format

Select the color format of the data stored in the file. You can select either `RGB` or `YCbCr 4:2:2`.

Image signal

Specify how the block accepts a color video signal. If you select `One multidimensional signal`, the block accepts an *M*-by-*N*-by-*P* color video signal, where *P* is the number of color planes, at one port. If you select `Separate color signals`, additional ports appear on the block. Each port accepts one *M*-by-*N* plane of an RGB video stream.

Video Quality (0-100)

Quality of the video specified as an integer scalar in the range `[0 100]`. This parameter is applicable only when **File name** is set to `MPEG4` and **Write** is set to `Video only`. By default, this parameter is set to `75`.

Troubleshooting

Running an Executable Outside MATLAB

To run your generated standalone executable application in Shell, you need to set your environment to the following:

Platform	Command

Mac	`setenv DYLD_LIBRARY_PATH "${DYLD_LIBRARY_PATH}:$MATLABROOT/bin/maci64" (csh/tcsh)`
	
	`export DYLD_LIBRARY_PATH=$LD_LIBRARY_PATH:$MATLABROOT/bin/maci64 (Bash)`

Linux	`setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:$MATLABROOT/bin/glnxa64 (csh/tcsh)`
	
	`export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$MATLABROOT/bin/glnxa64 (Bash)`

Windows	`set PATH=%PATH%;%MATLABROOT%\bin\win64`

Supported Data Types

For the block to display video data properly, double- and single-precision floating-point pixel values must be between 0 and 1. Any other data type requires the pixel values between the minimum and maximum values supported by their data type.

Check the specific codecs you are using for supported audio rates.

Port	Supported Data Types	Supports Complex Values?
Image	• Double-precision floating point • Single-precision floating point	No

Port	Supported Data Types	Supports Complex Values?
	• Boolean • 8-, 16- 32-bit signed integers • 8-, 16- 32-bit unsigned integers	
R, G, B	Same as Image port	No
Audio	• Double-precision floating point • Single-precision floating point • 16-bit signed integers • 32-bit signed integers • 8-bit unsigned integers	No
Y, Cb, Cr	Same as Image port	No

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

The executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

See Also

See Also

Blocks
`From Multimedia File`

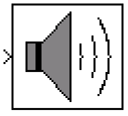
Topics

"How To Run a Generated Executable Outside MATLAB"

Introduced before R2006a

To Wave Device (Obsolete)

Send audio data to standard Windows audio device in real time



Library

`dspwin32`

Description

Note The To Wave Device block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the `To Audio Device` block.

The To Wave Device block sends audio data to a standard Windows audio device in real time. It is compatible with most popular Windows hardware, including Sound Blaster cards. The data is sent to the hardware in uncompressed pulse code modulation (PCM) format, and should typically be sampled at one of the standard Windows audio device rates: 8000, 11025, 22050, or 44100 Hz. Some hardware might support other rates in addition to these.

Note: Models that contain both the To Wave Device block and the From Wave Device block require a duplex-capable sound card.

The **Use default audio device** check box allows the To Wave Device block to detect and use the system's default audio hardware. You should select this option for systems that have a single sound device installed, or when the default sound device on a multiple-device system is your desired target. When the default sound device is *not* your desired output device, clear **Use default audio device**, and set the desired hardware in the **Audio device** parameter. This parameter lists the names of the installed audio devices.

The block input can contain audio data from a mono or stereo signal. A mono signal is represented as either a sample-based scalar or a frame-based length-M vector, where M is frame size. A stereo signal is represented as a sample-based length-2 vector or a frame-based M-by-2 matrix.

When the input data type is `uint8`, the block conveys the signal samples to the audio device using 8 bits. When the input data type is `double`, `single`, `int16`, or fixed point with a word length of 16 and a fraction length of 15, the block conveys the signal samples to the audio device using 16 bits by default. For inputs of data type `double` and `single`, you can also set the block to convey the signal samples using 24 bits by selecting the **Enable 24-bit output for double- and single-precision input signals** check box. The 24-bit sample width requires more memory but in general yields better fidelity.

The amplitude of the input must be in a valid range that depends on the input data type, as shown in the following table. Amplitudes outside the valid range are clipped to the nearest allowable value.

Input Data Type	Valid Input Amplitude Range
`double`	$-1 \leq amplitude < 1$
`single`	$-1 \leq amplitude < 1$
`int16`	$-32768 \leq amplitude \leq 32767$
`uint8`	$0 \leq amplitude \leq 255$
Fixed point with a word length of 16 and a fraction length of 15	$-1 \leq amplitude \leq 1 - 2^{-15}$

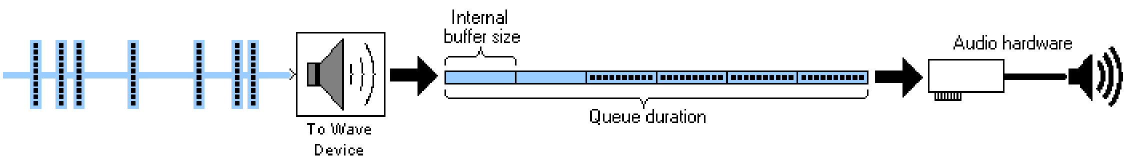
Buffering

Because audio devices generate real-time audio output, the Simulink environment must maintain a continuous flow of data to a device throughout simulation. Delays in passing data to the audio hardware can result in hardware errors or distortion of the output. This means that the To Wave Device block must in principle supply data to the audio hardware as quickly as the hardware reads the data. However, the To Wave Device block often *cannot* match the throughput rate of the audio hardware, especially when the simulation is running within Simulink rather than as generated code. Simulink execution speed can vary during the simulation as the host operating system services other processes. The block must therefore rely on a buffering strategy to ensure that signal data is available to the hardware on demand.

Note: This block requires real-time execution of the parent model for best performance.

The following block parameters control the memory management for this block:

- **Queue duration**
- **Automatically determine internal buffer size** or **User-defined internal buffer size**
- **Initial output delay**



The **Queue duration** parameter defines the overall size of the block's buffer. The block reads in chunks of data in the size of the input dimensions and stores them in the buffer. The internal buffer size defines the dimensions of the block output to the hardware. You can define the internal buffer size yourself in the **User-defined internal buffer size parameter**. If you select **Automatically determine internal buffer size** instead, the internal buffer size is calculated for you according to the following rules:

- If the input to the block has a frame size of 32 samples or larger, the internal buffer size be the same as the input frame size.
- If the input to the block has a frame size smaller than 32 samples, the internal buffer size is based on the input sample rate according to the following table, where

$$F_s = sampling\ frequency = \frac{1}{sample\ time}$$

F_s (Hz)	Internal Buffer Size (samples)
$F_s < 8000$	$\min(64, 2 * F_s)$
$8000 \le F_s < 22{,}050$	128
$22{,}050 \le F_s < 44{,}100$	256
$44{,}100 \le F_s < 96{,}000$	512
$F_s \ge 96{,}000$	1024

To minimize the chance of dropouts, the block checks to make sure that the queue duration is at least as big as twice the internal buffer size. If it is not, the queue duration is automatically set to twice the internal buffer size.

The **Initial output delay** parameter enables you to preload the buffer before the block starts to output data to the audio device, which can be helpful for models that do not run in real time. However, for real-time applications, it is best to set the initial output delay to zero (one frame of delay), or as close to zero as possible.

Troubleshooting

If you are getting undesirable audio output using the To Wave Device block, first determine whether your model can run in real time. Replace the To Wave Device block with a To Wave File block, run the model, and compare the model's simulation stop time to the elapsed time on your watch. If the model simulation stop time is less than the elapsed time on your watch, your model can probably run in real time. Then,

- If your model can run in real time,

  1. Select **Automatically determine internal buffer size**. This alone might solve the problem. If not,
  2. Try increasing the **Queue duration** parameter to a relatively large value, such as 0.5 s.

  If one or both of these options restores desirable audio output, you can try reducing the internal buffer size and/or queue duration until the quality of the audio output again degrades.
- If your model is not running in real time, try to make it run in real time by

1. Optimizing the model (using a more efficient implementation), or
2. Using a Simulink “Acceleration” (Simulink) mode, or
3. Generating stand-alone code

If none of these are possible, but the model only runs for a short period of time, set the **Queue duration** parameter to a size equal to a significant fraction of the model stop time and use a similarly large initial delay. This is not an optimal solution, but might work in some cases.

Parameters

Queue duration (seconds)

Specify the overall buffer size. To minimize the chance of dropouts, the block checks to make sure that the queue duration is as least as large as twice the internal buffer size. If it is not, the queue duration is automatically set to twice the internal buffer size.

Automatically determine internal buffer size

Select to have the block automatically select the internal buffer size for you. For details, see “Buffering” on page 2-1743.

User-defined internal buffer size (samples)

Define the internal buffer size, or the size of the chunks of data sent by the block to the audio hardware device.

This parameter is only visible when **Automatically determine internal buffer size** is not selected.

Initial output delay (seconds)

Specify the amount of time by which to delay the initial output to the audio device. During this time data accumulates in the block's buffer. Any value less than or equal to the queue duration specifies the smallest possible initial delay, which is a single frame.

Use default audio device

Select to direct audio output to the system's default audio device.

Audio device

This parameter lists the names of the installed audio devices. Specify the name of the audio device to receive the audio output. Select **Use default audio device** when the system has only a single audio card installed.

This parameter is only enabled when the **Use default audio device** check box is not selected.

Enable 24-bit output for double and single precision input signals

Select to output 24-bit data when inputs are double- or single-precision. Otherwise, the block outputs 16-bit data for double- and single-precision inputs.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Signed fixed point with a word length of 16 and a fraction length of 15 • 16-bit signed integers • 8-bit unsigned integers

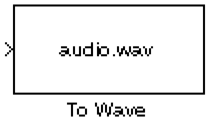
See Also

From Wave Device (Obsolete)	DSP System Toolbox
To Wave File (Obsolete)	DSP System Toolbox
`audioplayer`	MATLAB
`sound`	MATLAB

Introduced in R2008b

To Wave File (Obsolete)

Write audio data to file in Microsoft Wave (`.wav`) format



Library

`dspwin32`

Description

Note The To Wave File block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the `To Multimedia File` block.

The To Wave File block streams audio data to a Microsoft Wave (`.wav`) file in the uncompressed pulse code modulation (PCM) format. For compatibility reasons, the sample rate of the discrete-time input signal should typically be one of the standard Windows audio device rates (8000, 11025, 22050, or 44100 Hz), although the block supports arbitrary rates.

The input to the block, u, can contain audio data with one or more channels. A signal with C channels is represented as a sample-based length-C vector or a frame-based M-by-C matrix. The amplitude of the input should be in the range ±1. Values outside this range are clipped to the nearest allowable value.

```
wavwrite(u,Fs,bits,'filename')    % Equivalent MATLAB code
```

Note: AVI files are the only supported file type for non-Windows platforms.

Parameters

File name

Specify the path and name of the file to write. Paths can be relative or absolute. You do not need to specify the `.wav` extension.

Sample width (bits)

Specify the number of bits used to represent the signal samples in the file. The higher sample width settings require more memory but yield better fidelity for double- and single-precision inputs:

- `8` — Allocates 8 bits to each sample, allowing a resolution of 256 levels
- `16` — Allocates 16 bits to each sample, allowing a resolution of 65536 levels
- `24` — Allocates 24 bits to each sample, allowing a resolution of 16777216 levels
- `32` — Allocates 32 bits to each sample, allowing a resolution of 2^{32} levels ranging from -1 to 1

The 8-, 16-, and 24-bit modes output integer data, while the 32-bit mode outputs single-precision floating-point data.

Minimum number of samples for each write to file

Specify the number of consecutive samples, L, to write with each file access. To reduce the required number of file accesses, the block writes L consecutive samples to the file during each access for $L \geq M$. For $L < M$, the block instead writes M consecutive samples during each access. Larger values of L result in fewer file accesses, which reduces run-time overhead.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Signed fixed point with a word length of 16 and a fraction length of 15 • 16-bit signed integers • 8-bit unsigned integers

See Also

`From Multimedia File`	DSP System Toolbox
`To Audio Device`	DSP System Toolbox
To Workspace	Simulink

Introduced in R2010a

To Workspace

Write data to MATLAB workspace

Library

Sinks

`dspsnks4`

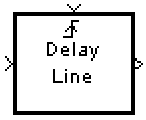
Description

The To Workspace block is an implementation of the Simulink To Workspace block. See To Workspace for more information.

Introduced in R2014b

Triggered Delay Line (Obsolete)

Buffer sequence of inputs into frame-based output



Library

dspobslib

Description

Note The Triggered Delay Line block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the `Delay Line` block.

The Triggered Delay Line block acquires a collection of M_o input samples into a frame, where you specify M_o in the **Delay line size** parameter. The block buffers a single sample from input 1 whenever it is triggered by the control signal at input 2 (⤒). When the next triggering event occurs, the newly acquired input sample is appended to the output frame so that the new output overlaps the previous output by M_o-1 samples. Between triggering events the block ignores input 1 and holds the output at its last value.

You specify the triggering event at input 2 in the **Trigger type** pop-up menu:

- `Rising edge` triggers execution of the block when the trigger input rises from a negative value to zero or a positive value, or from zero to a positive value.
- `Falling edge` triggers execution of the block when the trigger input falls from a positive value to zero or a negative value, or from zero to a negative value.
- `Either edge` triggers execution of the block when either a rising or falling edge (as described above) occurs.

The Triggered Delay Line block has zero latency, so the new input appears at the output in the same simulation time step. The output frame period is the same as the input sample period, $T_{fo}=T_{si}$.

Sample-Based Operation

In sample-based operation, the Triggered Delay Line block buffers a sequence of sample-based length-N vector inputs (1-D, row, or column) into a sequence of overlapping sample-based M_o-by-N matrix outputs, where you specify M_o in the **Delay line size** parameter (M_o>1). That is, each input vector becomes a *row* in the sample-based output matrix. When M_o=1, the input is simply passed through to the output, and retains the same dimension. Sample-based full-dimension matrix inputs are not accepted.

Frame-Based Operation

In frame-based operation, the Triggered Delay Line block rebuffers a sequence of frame-based M_i-by-N matrix inputs into an sequence of overlapping frame-based M_o-by-N matrix outputs, where M_o is the output frame size specified by the **Delay line size** parameter (that is, the number of consecutive samples from the input frame to rebuffer into the output frame). M_o can be greater or less than the input frame size, M_i. Each of the N input channels is rebuffered independently.

Initial Conditions

The Triggered Delay Line block's buffer is initialized to the value specified by the **Initial condition** parameter. The block always outputs this buffer at the first simulation step (*t*=0). When the block's output is a vector, the **Initial condition** can be a vector of the same size or a scalar value to be repeated across all elements of the initial output. When the block's output is a matrix, the **Initial condition** can be a matrix of the same size or a scalar to be repeated across all elements of the initial output.

Parameters

Trigger type

The type of event that triggers the block's execution.

Delay line size

The length of the output frame (number of rows in output matrix), M_o.

Initial condition

The value of the block's initial output, a scalar, vector, or matrix.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Trigger	• Any data type supported by the Trigger block
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

Buffer	DSP System Toolbox
Delay Line	DSP System Toolbox
Unbuffer	DSP System Toolbox

Introduced in R2008b

Triggered Signal From Workspace

Import signal samples from MATLAB workspace when triggered

Library

Signal Operations

`dspsigops`

Description

The Triggered Signal From Workspace block imports signal samples from the MATLAB workspace into the Simulink model when triggered by the control signal at the input port (). The **Signal** parameter specifies the name of a MATLAB workspace variable containing the signal to import, or any valid MATLAB expression defining a matrix or 3-D array.

When the **Signal** parameter specifies an M-by-N matrix (M≠1), each of the N columns is treated as a distinct channel. You specify the frame size in the **Samples per frame** parameter, M_o, and the output when triggered is an M_o-by-N matrix containing M_o consecutive samples from each signal channel. For convenience, an imported row vector (M=1) is treated as a single channel, so the output dimension is M_o-by-1.

When the **Signal** parameter specifies an M-by-N-by-P array, the block generates a single page of the array (an M-by-N matrix) at each trigger time. The **Samples per frame** parameter must be set to `1`.

Trigger Event

You specify the triggering event at the input port in the **Trigger type** pop-up menu:

- `Rising edge` triggers execution of the block when the trigger input rises from a negative value to zero or a positive value, or from zero to a positive value.

- `Falling edge` triggers execution of the block when the trigger input falls from a positive value to zero or a negative value, or from zero to a negative value.
- `Either edge` triggers execution of the block when either a rising or falling edge (as described above) occurs.

Initial and Final Conditions

The **Initial output** parameter specifies the output of the block from the start of the simulation until the first trigger event arrives. Between trigger events, the block holds the output value constant at its most recent value (that is, no linear interpolation takes place). For single-channel signals, the **Initial output** parameter value can be a vector of length M_o or a scalar to repeat across the M_o elements of the initial output frames. For matrix outputs (M_o-by-N or M-by-N), the **Initial output** parameter value can be a matrix of the same size or a scalar to be repeated across all elements of the initial output.

When the block has output all of the available signal samples, it can start again at the beginning of the signal, or simply repeat the final value or generate zeros until the end of the simulation. (The block does not extrapolate the imported signal beyond the last sample.) The **Form output after final data value by** parameter controls this behavior:

- When you specify `Setting To Zero`, the block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.
- When you specify `Holding Final Value`, the block repeats the final sample for the duration of the simulation after generating the last frame of the signal.
- When you specify `Cyclic Repetition`, the block repeats the signal from the beginning after generating the last frame. When there are not enough samples at the end of the signal to fill the final frame, the block zero-pads the final frame as necessary to ensure that the output for each cycle is identical (for example, the *i*th frame of one cycle contains the same samples as the *i*th frame of any other cycle).

Parameters

Signal

The name of the MATLAB workspace variable from which to import the signal, or a valid MATLAB expression specifying the signal.

Trigger type

The type of event that triggers the block's execution.

Initial output

The value to output until the first trigger event is received.

Samples per frame

The number of samples, M_o, to buffer into each output frame. This value must be `1` when you specify a 3-D array in the **Signal** parameter.

Form output after final data value by

Specifies the output after all of the specified signal samples have been generated. The block can output zeros for the duration of the simulation (`Setting to zero`), repeat the final data sample (`Holding Final Value`) or repeat the entire signal from the beginning (`Cyclic Repetition`).

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

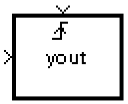
See Also

`Signal From Workspace`	DSP System Toolbox
To Workspace	Simulink
`Triggered To Workspace`	DSP System Toolbox

Introduced before R2006a

Triggered To Workspace

Write input sample to MATLAB workspace when triggered



Library

Sinks

`dspsnks4`

Description

The Triggered To Workspace block creates a matrix or array variable in the MATLAB workspace, where it stores the acquired inputs at the end of a simulation. The block overwrites an existing variable with the same name.

When you set the **Save 2-D signals as** parameter to `2-D array (concatenate along first dimension`, the block saves an *M*-by-*N* input as a *P*-by-*N* matrix, where *P* is the **Maximum number of rows** parameter. When the simulation progresses long enough for the block to acquire more than *P* samples, the block stores only the most recent *P* samples. The **Decimation factor**, *D*, allows you to store only every *D*th input matrix.

When you set the **Save 2-D signals as** parameter to `3-D array (concatenate along third dimension`, the block saves an *M*-by-*N* input as a three-dimensional array in which each *M*-by-*N* page represents a single sample from each of the *M***N* channels (the most recent input matrix occupies the last page). The maximum size of this variable is limited to *M*-by-*N*-by-*P*, where *P* is the **Maximum number of rows** parameter. When the simulation progresses long enough for the block to acquire more than *P* inputs, it stores only the last *P* inputs. The **Decimation factor**, *D*, allows you to store only every *D*th input matrix.

The block acquires and buffers a single frame from input 1 whenever it is triggered by the control signal at input 2 (). At all other times, the block ignores input 1. You specify the triggering event at input 2 in the **Trigger type** pop-up menu:

- `Rising edge` triggers execution of the block when the trigger input rises from a negative value to zero or a positive value, or from zero to a positive value.
- `Falling edge` triggers execution of the block when the trigger input falls from a positive value to zero or a negative value, or from zero to a negative value.
- `Either edge` triggers execution of the block when either a rising or falling edge (as described above) occurs.

To save a record of the sample time corresponding to each sample value, open the Configuration Parameters dialog box. In the **Select** pane, click **Data Import/Export** and select the **Time** check box.

The nontriggered version of this block is the To Workspace block.

Parameters

Trigger type

The type of event that triggers the block's execution.

Variable name

The name of the workspace variable in which to store the data.

Maximum number of rows

The maximum number of rows (one row per time step) to be saved, P.

Decimation

The decimation factor, D.

Save 2-D signals as

Specify whether the block saves 2-D signals as a 2-D or 3-D array in the MATLAB workspace:

- `2-D array (concatenate along first dimension)` — When you select this option, the block vertically concatenates each M-by-N matrix input with the previous input to produce a 2-D output array.
- `3-D array (concatenate along third dimension)` — When you select this option, the block saves an M-by-N input signal as a 3-D array. The maximum size of this 3-D array is limited to M-by-N-by-P, where P is the **Maximum number of rows** parameter. When the simulation progresses long enough for the block to acquire more than P inputs, the block stores only the last P inputs. The **Decimation factor**, D, allows you to store only every Dth input matrix.

Log fixed-point data as a fi object

Select to log fixed-point data to the MATLAB workspace as a Fixed-Point Designer `fi` object. Otherwise, fixed-point data is logged to the workspace as `double`.

Supported Data Types

Port	Supported Data Types
Input	• Any data type supported by the To Workspace block
Trigger	• Any data type supported by the Trigger block

See Also

`Signal From Workspace`	DSP System Toolbox
To Workspace	Simulink
`Triggered Signal From Workspace`	DSP System Toolbox

Introduced before R2006a

Two-Channel Analysis Subband Filter

Decompose signal into high-frequency and low-frequency subbands

Library

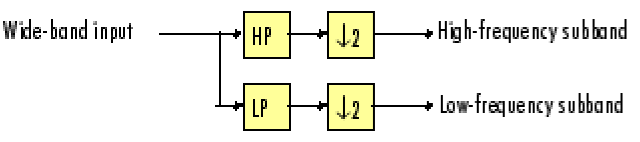
Filtering / Multirate Filters

`dspmlti4`

Description

The Two-Channel Analysis Subband Filter block decomposes the input into high-frequency and low-frequency subbands, each with half the bandwidth and half the sample rate of the input.

The block filters the input with a pair of highpass and lowpass FIR filters, and then downsamples the results by 2, as illustrated in the following figure.



The block implements the FIR filtering and downsampling steps together using a polyphase filter structure, which is more efficient than the straightforward filter-then-decimate algorithm shown in the preceding figure. Each subband is the first phase of the respective polyphase filter. You can implement a multilevel dyadic analysis filter bank by connecting multiple copies of this block or by using the `Dyadic Analysis Filter Bank` block. See “Creating Multilevel Dyadic Analysis Filter Banks” on page 2-1764 for more information.

You must provide a vector of filter coefficients for the lowpass and highpass FIR filters. Each filter should be a half-band filter that passes the frequency band that the other filter stops.

See the following topics for more information about this block:

- "Specifying the FIR Filters" on page 2-1762
- "Frame-Based Processing" on page 2-1762
- "Sample-Based Processing" on page 2-1763
- "Latency" on page 2-1763
- "Creating Multilevel Dyadic Analysis Filter Banks" on page 2-1764

Specifying the FIR Filters

You must provide the vector of numerator coefficients for the lowpass and highpass filters in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.

For example, to specify a filter with the following transfer function, enter the vector `[b(1) b(2) ... b(m)]`.

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

Each filter should be a half-band filter that passes the frequency band that the other filter stops. You can use the `Two-Channel Synthesis Subband Filter` block to reconstruct the input to this block. To do so, you must design perfect reconstruction filters to use in the synthesis subband filter.

The best way to design perfect reconstruction filters is to use the Wavelet Toolbox `wfilters` function in to design both the filters both in this block and in the `Two-Channel Synthesis Subband Filter` block. You can also use other DSP System Toolbox and Signal Processing Toolbox functions.

The Two-Channel Analysis Subband Filter block initializes all filter states to zero.

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block accepts an M-by-N matrix. The block treats each column of the input as the high- or low-frequency subbands of the corresponding output channel. You can use the **Rate options** parameter to specify how the block resamples the input:

- When you set the **Rate options** parameter to `Enforce single-rate processing`, the input to the block can be an M-by-N matrix, where M is a multiple of two. The block treats each column of the input as an independent channel and decomposes each channel over time. The block outputs two matrices, where each column of the output is the high- or low-frequency subband of the corresponding input column. To maintain the input sample rate, the block decreases the output frame size by a factor of two.

- When you set the **Rate options** parameter to `Allow multirate processing`, the block treats an M_i-by-N matrix input as N independent channels and decomposes each channel over time. The block outputs two M-by-N matrices, where each column of the output is the high- or low-frequency subband of the corresponding input column. The input and output frame *sizes* are the same, but the frame *rate* of the output is half that of the input. Thus, the overall sample rate of the output is half that of the input.

  In this mode, the block has one frame of latency, as described in the "Latency" on page 2-1763 section.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an M-by-N matrix input as $M \cdot N$ independent channels. The block decomposes each channel over time and outputs two M-by-N matrices whose sample rates are half the input sample rate. Each element in the output matrix is the high- or low-frequency subband output of the corresponding element of the input matrix.

Depending on the setting of your Simulink configuration parameters, the output may have one sample of latency, as described in the "Latency" on page 2-1763 section.

Latency

When you set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`, the Two-Channel Analysis Subband Filter block always has zero-tasking latency. *Zero-tasking latency* means that the block propagates the first input sample (received at time t=0) as the first output sample.

When you set the **Rate options** parameter to `Allow multirate processing`, the Two-Channel Analysis Subband Filter block may exhibit latency. The amount of latency depends on the setting of the **Input processing** parameter of this block, and the setting of the Simulink **Treat each discrete rate as a separate task** configuration parameter. The following table summarizes the conditions that produce latency when the block is performing multirate processing.

Input processing	Treat each discrete rate as a separate task	Latency
`Elements as channels (sample based)`	`Off`	None.
	`On`	One sample. The first output sample in each channel always has a value of `0`.
`Columns as channels (frame based)`	`On` or `Off`	One frame. All samples in the first output frame have a value of `0`.

Note: For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Time-Based Scheduling and Code Generation" (Simulink Coder).

Creating Multilevel Dyadic Analysis Filter Banks

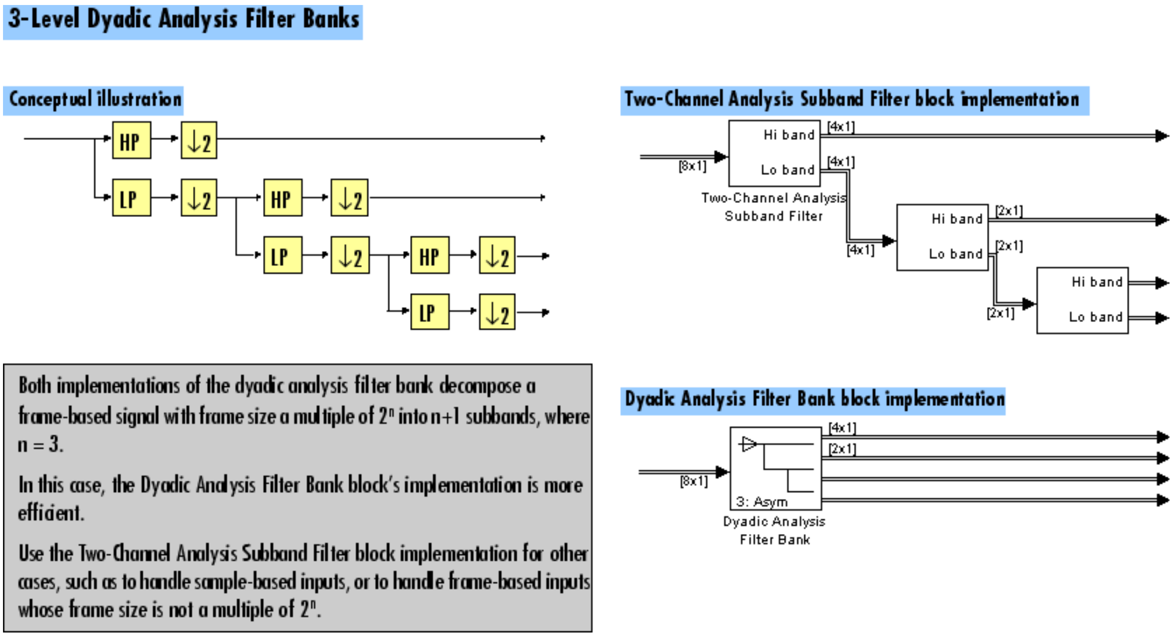
The Two-Channel Analysis Subband Filter block is the basic unit of a dyadic analysis filter bank. You can connect several of these blocks to implement an n-level filter bank, as illustrated in the following figure. For a review of dyadic analysis filter banks, see the `Dyadic Analysis Filter Bank` block reference page.

When you create a filter bank by connecting multiple copies of this block, the output values of the filter bank differ depending on whether there is latency. Though the output values differ, both sets of values are valid; the difference arises from changes in latency. See the "Latency" on page 2-1763 section for more information about when latency can occur in the Two-Channel Analysis Subband Filter block.

In some cases, rather than connecting several Two-Channel Analysis Subband Filter blocks, you can use the `Dyadic Analysis Filter Bank` block, which is faster and requires less memory. In particular, the Dyadic Analysis Filter Bank block is more efficient under the following conditions:

- The frame size of the signal you are decomposing is a multiple of 2^n.
- You are decomposing the signal into n+1 or 2^n subbands.

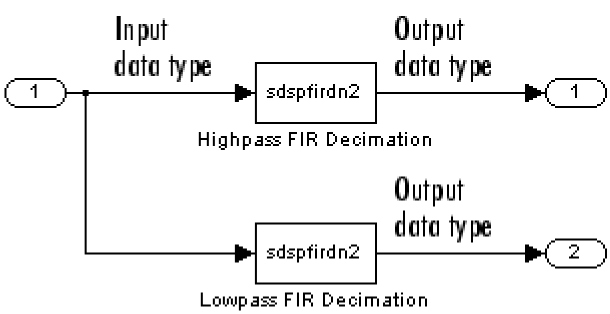
In all other cases, use Two-Channel Analysis Subband Filter blocks to implement your filter banks.



The `Dyadic Analysis Filter Bank` block allows you to specify the filter bank filters by providing vectors of filter coefficients, just as this block does. The Dyadic Analysis Filter Bank block provides an additional option of using wavelet-based filters that the block designs by using a wavelet you specify.

Fixed-Point Data Types

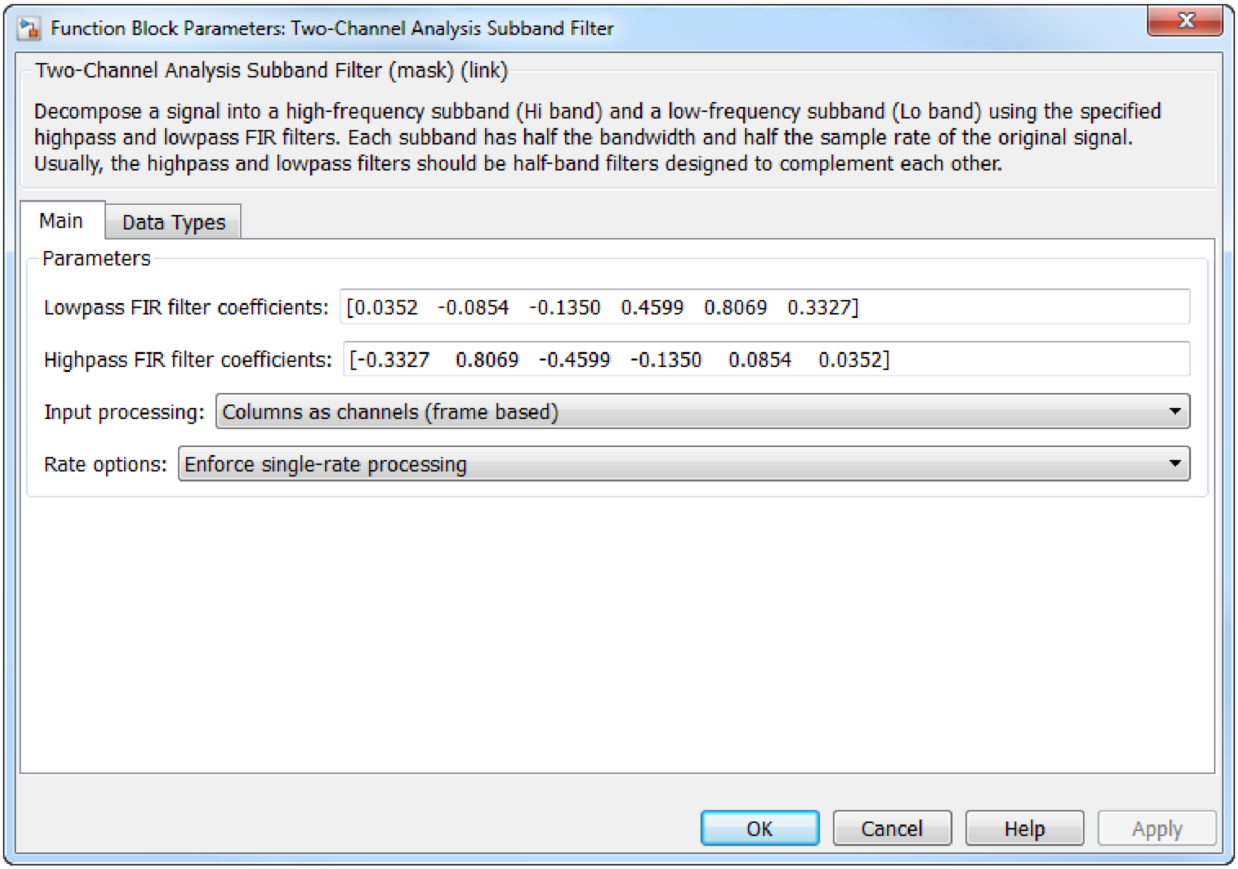
The Two-Channel Analysis Subband Filter Bank block is composed of two FIR Decimation blocks as shown in the following diagram.



For fixed-point signals, you can set the coefficient, product output, accumulator, and output data types of the FIR Decimation blocks as discussed in "Dialog Box" on page 2-1766. For a diagram showing the usage of these data types, see the `FIR Decimation` block reference page.

Dialog Box

The **Main** pane of the Two-Channel Analysis Subband Filter block dialog appears as follows.



Lowpass FIR filter coefficients

Specify a vector of lowpass FIR filter coefficients, in descending powers of *z*. The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. The default values of this parameter specify a filter based on a third-order Daubechies wavelet. When you use the `Two-Channel Synthesis Subband Filter` block to reconstruct the input to this block, you need to design perfect reconstruction filters to use in the synthesis subband filter. For more information, see "Specifying the FIR Filters" on page 2-1762.

Highpass FIR filter coefficients

Specify a vector of highpass FIR filter coefficients, in descending powers of *z*. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. The default values of this parameter specify a filter based on a third-order Daubechies wavelet. When you use the `Two-Channel Synthesis Subband Filter` block to reconstruct the input to this block, you need to design perfect reconstruction filters to use in the synthesis subband filter. For more information, see "Specifying the FIR Filters" on page 2-1762.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

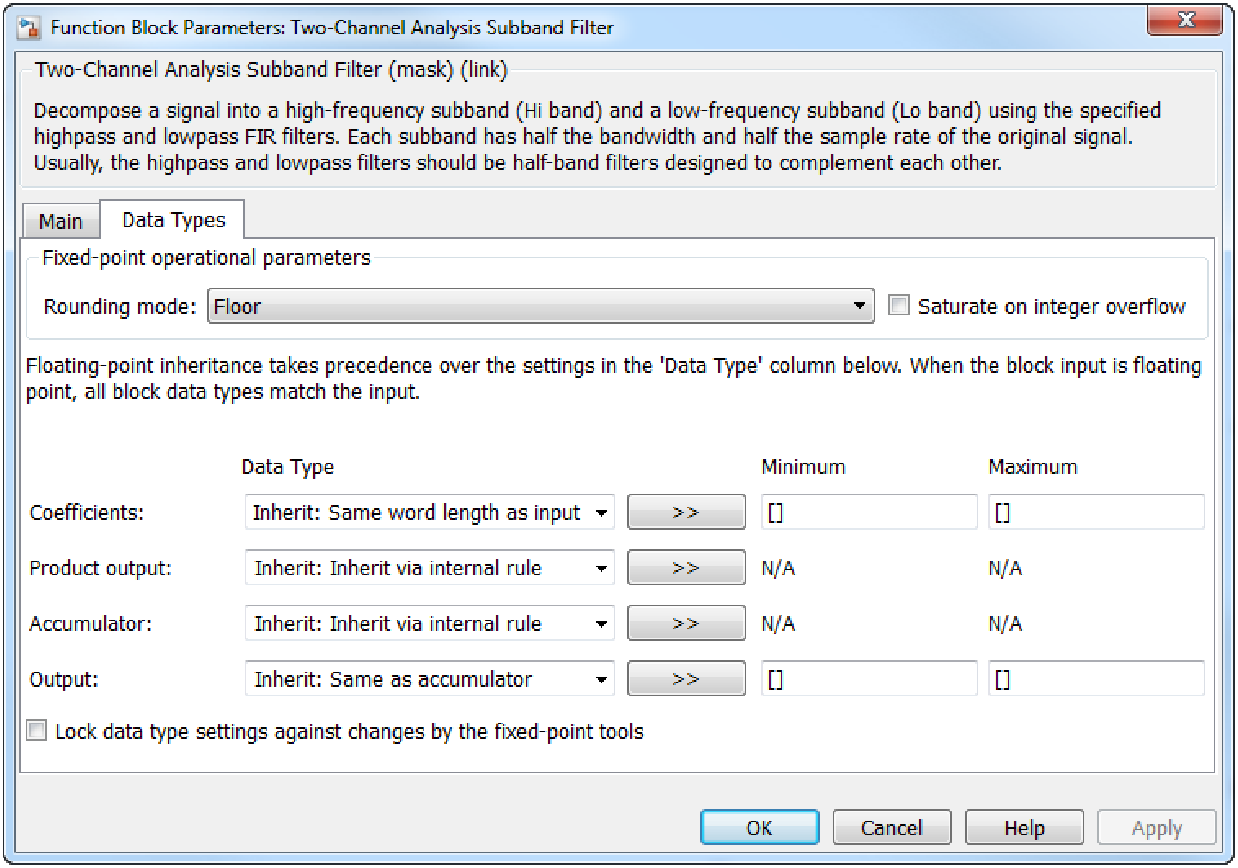
Rate options

Specify the rate processing rule for the block. You can set this parameter to one of the following options:

- `Enforce single-rate processing` — When you select this option, the block treats each column of the input as an independent channel and decomposes each channel over time. The output has the same sample rate as the input, but the output frame size is half that of the input frame size. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the input and output of the block are the same size, but the sample rate of the output is half that of the input.

Some settings of this parameter cause the block to have nonzero latency. See “Latency” on page 2-1763 for more information.

The **Data Types** pane of the Two-Channel Analysis Subband Filter block dialog appears as follows.



Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always rounded to `Nearest`.

Note: The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output data type** is `Inherit: Inherit via internal rule`
- **Accumulator data type** is `Inherit: Inherit via internal rule`
- **Output data type** is `Inherit: Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients data type

Specify the coefficients data type. See "Fixed-Point Data Types" on page 2-1765 and "Multiplication Data Types" for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Coefficients data type** parameter.

See "Specify Data Types Using Data Type Assistant" (Simulink) in Simulink User's Guide (Simulink) for more information.

Coefficients Minimum

Specify the minimum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Coefficients Maximum

Specify the maximum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-1765 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.

Note The actual product output word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-1765 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-1765 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

References

[1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

[2] Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

[3] Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

See Also

See Also

Functions
`fir1` | `fir2` | `firls`

Blocks
`DWT` | `Dyadic Analysis Filter Bank` | `FIR Decimation` | `IDWT` | `Two-Channel Synthesis Subband Filter`

Topics
"Multirate and Multistage Filters"

Introduced before R2006a

Two-Channel Synthesis Subband Filter

Reconstruct signal from high-frequency and low-frequency subbands

Library

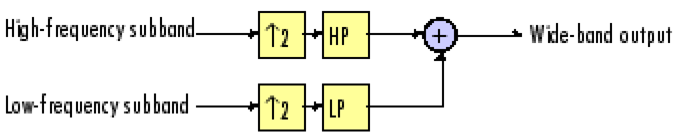
Filtering / Multirate Filters

`dspmlti4`

Description

The Two-Channel Synthesis Subband Filter block reconstructs a signal from its high-frequency and low-frequency subbands, each with half the bandwidth and half the sample rate of the original signal. Use this block to reconstruct signals decomposed by the `Two-Channel Analysis Subband Filter` block.

The block upsamples the high- and low-frequency subbands by 2, and then filters the results with a pair of highpass and lowpass FIR filters, as illustrated in the following figure.



The block implements the FIR filtering and downsampling steps together using a polyphase filter structure, which is more efficient than the straightforward interpolate-then-filter algorithm shown in the preceding figure. You can implement a multilevel dyadic synthesis filter bank by connecting multiple copies of this block or by using the `Dyadic Synthesis Filter Bank` block. For more information, see “Creating Multilevel Dyadic Synthesis Filter Banks” on page 2-1776.

You must provide a vector of filter coefficients for the lowpass and highpass FIR filters. Each filter should be a half-band filter that passes the frequency band that the other filter stops. You can use this block to reconstruct the output of a `Two-Channel Analysis Subband Filter` block. To do so, you must design the filters in this block such that they perfectly reconstruct the outputs of the analysis filters.

See the following topics for more information about this block:

- "Specifying the FIR Filters" on page 2-1774
- "Frame-Based Processing" on page 2-1774
- "Sample-Based Processing" on page 2-1775
- "Latency" on page 2-1776
- "Creating Multilevel Dyadic Synthesis Filter Banks" on page 2-1776

Specifying the FIR Filters

You must provide the vector of numerator coefficients for the lowpass and highpass filters in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.

For example, to specify a filter with the following transfer function, enter the vector `[b(1) b(2) ... b(m)]`.

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

Each filter should be a half-band filter that passes the frequency band that the other filter stops. You can use this block to reconstruct the output of a `Two-Channel Analysis Subband Filter` block. To do so, you must design the filters in this block such that they perfectly reconstruct the outputs of the analysis filters.

The best way to design perfect reconstruction filters is to use the Wavelet Toolbox `wfilters` function for the filters in both this block *and* in the corresponding Two-Channel Analysis Subband Filter block. You can also use DSP System Toolbox and Signal Processing Toolbox functions.

The Two-Channel Synthesis Subband Filter block initializes all filter states to zero.

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block accepts any two M-by-N matrices with the same frame rates. The block treats each column of the input as the high- or low-frequency subbands of the corresponding output channel. You can use the **Rate options** parameter to specify how the block resamples the input:

- When you set the **Rate options** parameter to `Enforce single-rate processing`, the input to the block can be any two *M*-by-*N* matrices with the same frame rate. The block treats each input column as the high- or low-frequency subbands of the corresponding output channel. The input to the topmost input port should contain the high-frequency subbands. The block outputs one matrix, where each column is reconstructed from the corresponding columns of each input matrix. The input and output frame *rates* are the same, but the frame *size* of the output is twice that of the input.
- When you set the **Rate options** parameter to `Allow multirate processing`, the block treats each column of the input as the high- or low-frequency subbands of the corresponding output channel. The input to the topmost input port should contain the high-frequency subbands. The block outputs one matrix, where each column is reconstructed from the corresponding columns of the input matrices. The input and output frame *sizes* are the same, but the frame *rate* of the output is twice that of the input. Thus, the overall sample rate of the output is twice that of the input sample rate.

  In this mode, the block has one frame of latency, as described in the "Latency" on page 2-1776 section.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block accepts any two *M*-by-*N* matrices with the same sample rates. The block treats each *M*-by-*N* matrix as $M \cdot N$ independent subbands. Each element of the input matrices is the high- or low-frequency subband of the corresponding channel in the output matrix. The input to the topmost input port should contain the high-frequency subbands. The block outputs one matrix with the same dimensions as the input matrices, but a sample rate that is twice that of the input. The block reconstructs each element of the output from the corresponding elements in the input matrices.

Depending on the setting of your Simulink configuration parameters, the output may have one sample of latency, as described in the "Latency" on page 2-1776 section.

Latency

When you set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`, the Two-Channel Synthesis Subband Filter block always has zero-tasking latency. *Zero-tasking latency* means that the block propagates the first input sample (received at time t= 0) as the first output sample.

When you set the **Rate options** parameter to `Allow multirate processing`, the Two-Channel Synthesis Subband Filter block may exhibit latency. The amount of latency depends on the setting of the **Input processing** parameter of this block and the setting of the Simulink **Treat each discrete rate as a separate task** configuration parameter. The following table summarizes the conditions that produce latency when the block is performing multirate processing.

Input processing	Treat each discrete rate as a separate task	Latency
`Elements as channels (sample based)`	`Off`	None.
	`On`	One sample. The first output sample in each channel always has a value of `0`.
`Columns as channels (frame based)`	`Off` or `On`	One frame. All samples in the first output frame have a value of `0`.

Note: For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Time-Based Scheduling and Code Generation" (Simulink Coder).

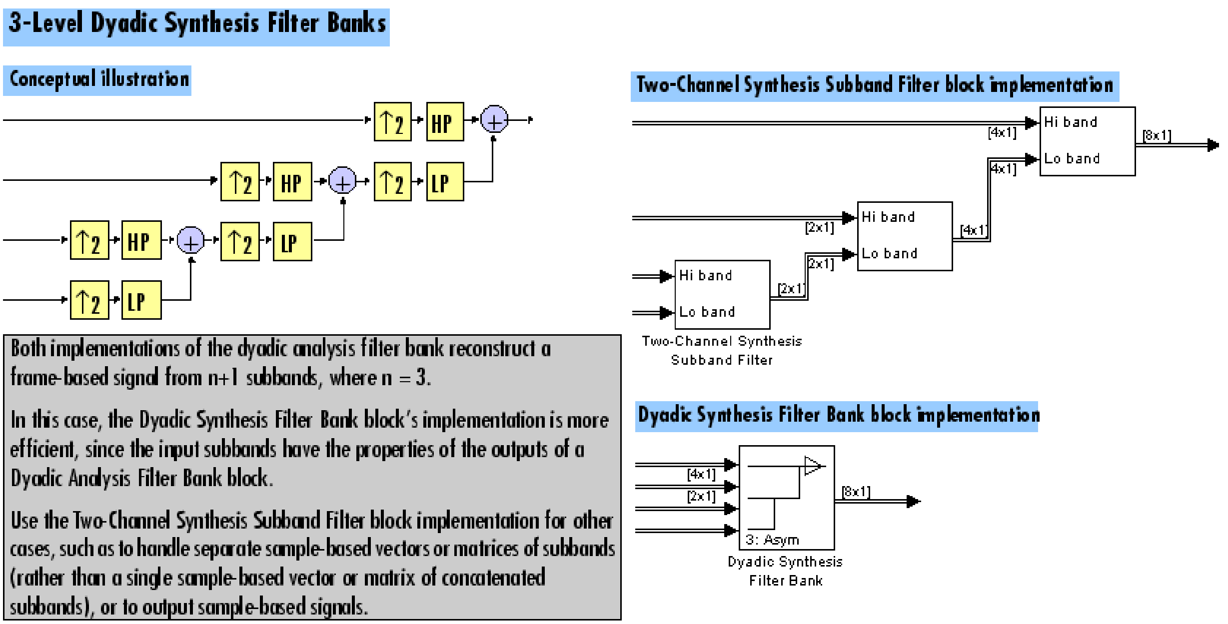
Creating Multilevel Dyadic Synthesis Filter Banks

The Two-Channel Synthesis Subband Filter block is the basic unit of a dyadic synthesis filter bank. You can connect several of these blocks to implement an n-level filter bank, as illustrated in the following figure. For a review of dyadic synthesis filter banks, see the `Dyadic Synthesis Filter Bank` block reference page.

When you create a filter bank by connecting multiple copies of this block, the output values of the filter bank differ depending on whether there is latency. Though the output values differ, both sets of values are valid; the difference arises from changes in latency. See the “Latency” on page 2-1776 section for more information about when latency can occur in the Two-Channel Analysis Subband Filter block.

In some cases, rather than connecting several Two-Channel Analysis Subband Filter blocks, you can use the `Dyadic Analysis Filter Bank` block, which is faster and requires less memory. In particular, the Dyadic Analysis Filter Bank block is more efficient under the following conditions:

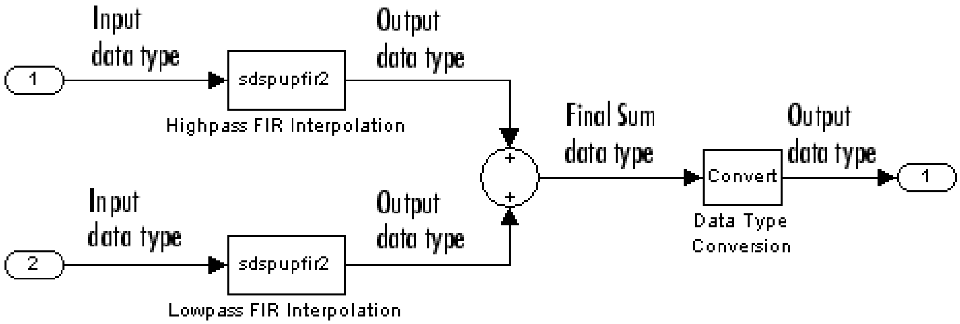
- You are reconstructing a signal from 2^n or $n+1$ subbands.
- The frame size of the signal you are reconstructing is a multiple of 2^n.
- The properties of the subbands you are working with match those of the outputs of the Dyadic Analysis Filter Bank block. These properties are described in the `Dyadic Analysis Filter Bank` reference page.



The `Dyadic Synthesis Filter Bank` block allows you to specify the filter bank filters by providing vectors of filter coefficients, just as this block does. The Dyadic Synthesis Filter Bank block provides an additional option of using wavelet-based filters that the block designs by using a wavelet you specify.

Fixed-Point Data Types

The Two-Channel Synthesis Subband Filter block is composed of two FIR Interpolation blocks as shown in the following diagram.

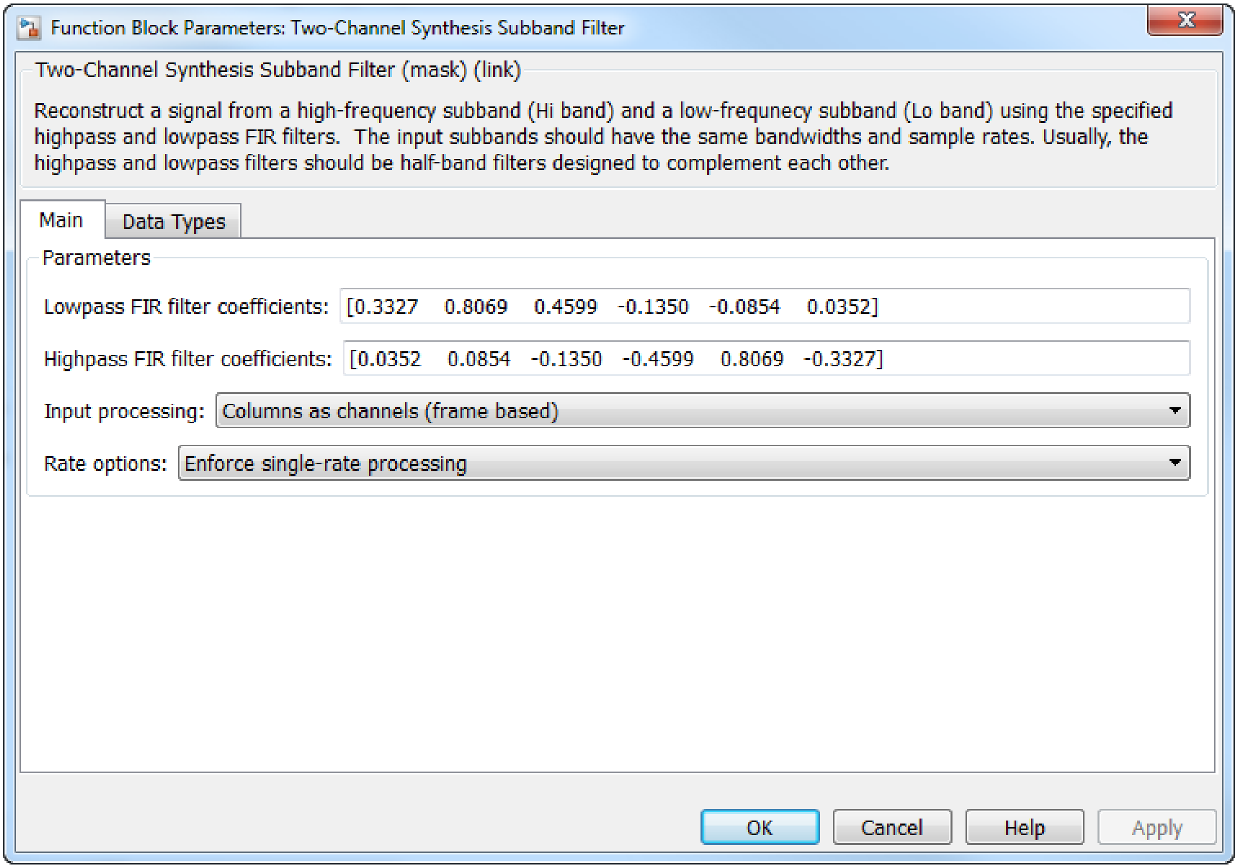


For fixed-point signals, you can set the coefficient, product output, accumulator, and output data types used in the FIR Interpolation blocks as discussed in "Dialog Box" on page 2-1778. For a diagram showing the usage of these data types within the FIR blocks, see the `FIR Interpolation` block reference page.

In addition, the inputs to the Sum block shown in the previous diagram are accumulated using the accumulator data type. The output of the Sum block is then cast from the accumulator data type to the output data type. Therefore the output of the Two-Channel Synthesis Subband Filter block is in the output data type. You also set these data types in the block dialog box as discussed in the "Dialog Box" on page 2-1778 section.

Dialog Box

The **Main** pane of the Two-Channel Synthesis Subband Filter block dialog appears as follows.



Lowpass FIR filter coefficients

A vector of lowpass FIR filter coefficients, in descending powers of z. The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. To use this block to reconstruct the output of a `Two-Channel Analysis Subband Filter` block, you must design the filters in this block to perfectly reconstruct the outputs of the analysis filters. For more information, see "Specifying the FIR Filters" on page 2-1774.

Highpass FIR filter coefficients

A vector of highpass FIR filter coefficients, in descending powers of z. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. To use this block to reconstruct the output of a `Two-Channel Analysis Subband Filter` block, you must design the filters in this block to perfectly reconstruct the outputs of the analysis filters. For more information, see "Specifying the FIR Filters" on page 2-1774.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

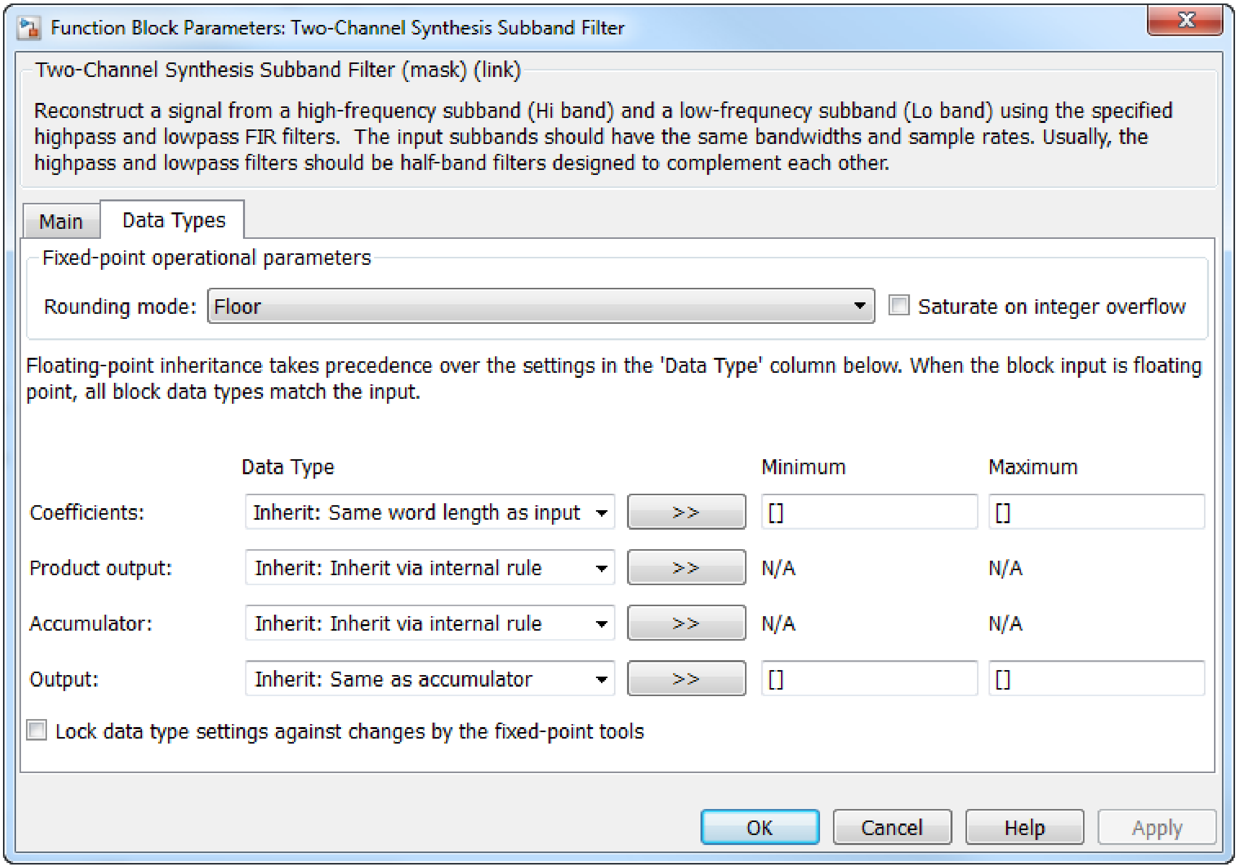
Rate options

Specify the rate processing rule for the block. You can set this parameter to one of the following options:

- `Enforce single-rate processing` — When you select this option, the block treats each column of the input as an independent channel and reconstructs each channel over time. The output has the same sample rate as the input, but the output frame size is twice that of the input frame size. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the input and output of the block are the same size, but the sample rate of the output is twice that of the input.

Some settings of this parameter cause the block to have nonzero latency. See "Latency" on page 2-1776 for more information.

The **Data Types** pane of the Two-Channel Synthesis Subband Filter block dialog appears as follows.



Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Note: The **Rounding mode** and **Saturate on integer overflow** settings have no effect on numerical results when all the following conditions exist:

- **Product output data type** is `Inherit: Inherit via internal rule`
- **Accumulator data type** is `Inherit: Inherit via internal rule`
- **Output data type** is `Inherit: Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.

Saturate on integer overflow

When you select this check box, the block saturates the result of its fixed-point operation. When you clear this check box, the block wraps the result of its fixed-point operation. By default, this check box is cleared. For details on `saturate` and `wrap`, see overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients data type

Specify the coefficients data type. See “Fixed-Point Data Types” on page 2-1778 and “Multiplication Data Types” for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Coefficients data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Coefficients Minimum

Specify the minimum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Coefficients Maximum

Specify the maximum value of the filter coefficients. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Automatic scaling of fixed-point data types

Product output data type

Specify the product output data type. See “Fixed-Point Data Types” on page 2-1778 and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.

Note The actual product output word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Accumulator data type

Specify the accumulator data type. See “Fixed-Point Data Types” on page 2-1778 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`. For more information on this rule, see “Inherit via Internal Rule”.
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button >> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See “Specify Data Types Using Data Type Assistant” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output data type

Specify the output data type. See “Fixed-Point Data Types” on page 2-1778 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

Output Minimum

Specify the minimum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Output Maximum

Specify the maximum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges” (Simulink))
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

References

[1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

[2] Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

[3] Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

See Also

See Also

Functions
`fir1` | `fir2` | `firls`

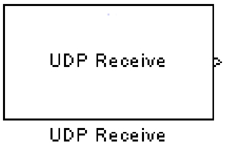
Blocks
`DWT` | `Dyadic Synthesis Filter Bank` | `FIR Interpolation` | `IDWT` | `Two-Channel Analysis Subband Filter`

Topics
"Multirate and Multistage Filters"

Introduced before R2006a

UDP Receive

Receive `uint8` vector as UDP message



Library

Sources

`dspsrcs4`

Description

The UDP Receive block receives UDP packets from an IP network port and saves them to its buffer. With each sample, the block outputs the contents of a single UDP packet as a data vector. The local IP port number on which the block receives the UDP packets is tunable in the C/C++ generated code.

The generated code for this block relies on prebuilt `.dll` files. You can run this code outside the MATLAB environment, or redeploy it, but you must account for these extra `.dll` files when doing so. The `packNGo` function creates a single zip file containing all of the pieces required to run or rebuild this code. For more details, see "How To Run a Generated Executable Outside MATLAB".

Parameters

Local IP port

Specify the IP port number on which to receive UDP packets. This parameter is tunable in the C/C++ generated code but not tunable during simulation. The default is `25000`. The value can be in the range [1 65535].

Note: On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

Remote IP address ('0.0.0.0' to accept all)

Specify the IP address from which to accept packets. Entering a specific IP address blocks UDP packets from other addresses. To accept packets from any IP address, enter `'0.0.0.0'`. This value defaults to `'0.0.0.0'`.

Receive buffer size (bytes)

Make the receive buffer large enough to avoid data loss caused by buffer overflows. This value defaults to `8192`.

Maximum length for Message

Specify the maximum length, in vector elements, of the data output vector. Set this parameter to a value equal or greater than the data size of a UDP packet. The system truncates data that exceeds this length. This value defaults to `255`.

If you disable **Output variable-size signal**, the block outputs a fixed-length output the same length as the **Maximum length for Message**.

Data type for Message

Set the data type of the vector elements in the Message output. Match the data type with the data input used to create the UDP packets. This option defaults to `uint8`.

Output variable-size signal

If your model supports signals of varying length, enable the **Output variable-size signal** parameter. This checkbox defaults to selected (enabled). In that case:

- The output vector varies in length, depending on the amount of data in the UDP packet.
- The block emits the data vector from a single unlabeled output.

If your model does not support signals of varying length, disable the **Output variable-size signal** parameter. In that case:

- The block emits a fixed-length output the same length as the **Maximum length for Message**.
- If the UDP packet contains less data than the fixed-length output, the difference contains invalid data.

- The block emits the data vector from the **Message** output.
- The block emits the length of the valid data from the **Length** output.
- The block dialog box displays the **Data type for Length** parameter.

In both cases, the block truncates data that exceeds the **Maximum length for Message**.

Data type for Length

Set the data type of the Length output. This option defaults to `double`.

Blocking time (seconds)

For each sample, wait this length of time for a UDP packet before returning control to the scheduler. This value defaults to `inf`, which indicates to wait indefinitely.

Note: This parameter appears only in the Embedded Coder® UDP Receive block.

Sample time (seconds)

Specify how often the scheduler runs this block. Enter a value greater than zero. In real-time operation, setting this option to a large value reduces the likelihood of dropped UDP messages. This value defaults to a sample time of 0.01 s.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- The executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- The **Local IP port** parameter is tunable in the generated code, but not tunable during simulation. You can control the parameter tunability in the generated code

through several ways. One of the ways is to configure the parameter as a tunable field of a global structure in the generated code. Other ways include applying a built-in storage class or custom storage class to a `Simulink.Parameter` object and using this object to set the value of the block parameter. For details, see “Block Parameter Representation in the Generated Code” (Simulink Coder).

See Also

See Also

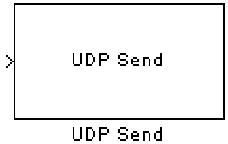
System Objects
dsp.UDPSender | dsp.UDPReceiver

Blocks
`UDP Send`

Introduced in R2010a

UDP Send

Send UDP message



Library

Sinks

`dspsnks4`

Description

The UDP Send block transmits an input vector as a UDP message over an IP network port. The remote IP port number to which the block sends the UDP packets is tunable in the C/C++ generated code.

Note: Some Simulink blocks and `.exe` files built from models that contain those blocks require shared libraries, such as `.dll` files on Windows. The UDP Send block requires the `networkdevice.dll` library file. To meet this requirement, follow the example on the `packNGo` function page to package the code files for your model. The resulting compressed folder contains the `.dll` files that the model requires, including `networkdevice.dll`. To run this type of `.exe` file outside a MATLAB environment, place the required `.dll` files in the same folder as the `.exe` file, or place them in a folder on the Windows system path. For more details, see “How To Run a Generated Executable Outside MATLAB”.

Parameters

IP address ('255.255.255.255' for broadcast)

Specify the IP address or hostname to which the block sends the message. To broadcast the UDP message, retain the default value, `'255.255.255.255'`.

Remote IP port

Specify the port to which the block sends the message. This parameter is tunable in the C/C++ generated code but not tunable during simulation. The default is `25000`. The value can be in the range [1 65535].

Note: On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

Local IP port source

To let the system automatically assign the port number, select `Assign automatically`. To specify the IP port number using the **Local IP port** parameter, select `Specify`.

Local IP port

Specify the IP port number from which the block sends the message.

If the receiving address expects messages from a particular port number, enter that number here.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- The executable generated from this block relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this block and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "How To Run a Generated Executable Outside MATLAB".

- The **Remote IP port** parameter is tunable in the generated code, but not tunable during simulation. You can control the parameter tunability in the generated code through several ways. One of the ways is to configure the parameter as a tunable field of a global structure in the generated code. Other ways include applying a built-in storage class or custom storage class to a `Simulink.Parameter` object and using this object to set the value of the block parameter. For details, see “Block Parameter Representation in the Generated Code” (Simulink Coder).

See Also

See Also

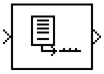
System Objects
dsp.UDPSender | dsp.UDPReceiver

Blocks
`UDP Receive`

Introduced in R2010a

Unbuffer

Unbuffer input frame into sequence of scalar outputs



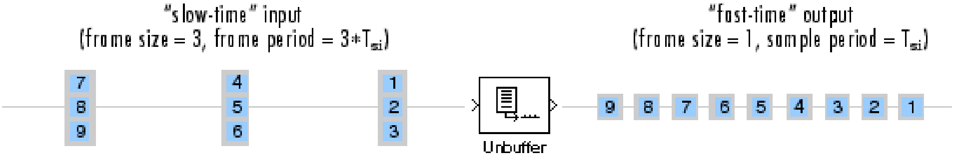
Library

Signal Management / Buffers

`dspbuff3`

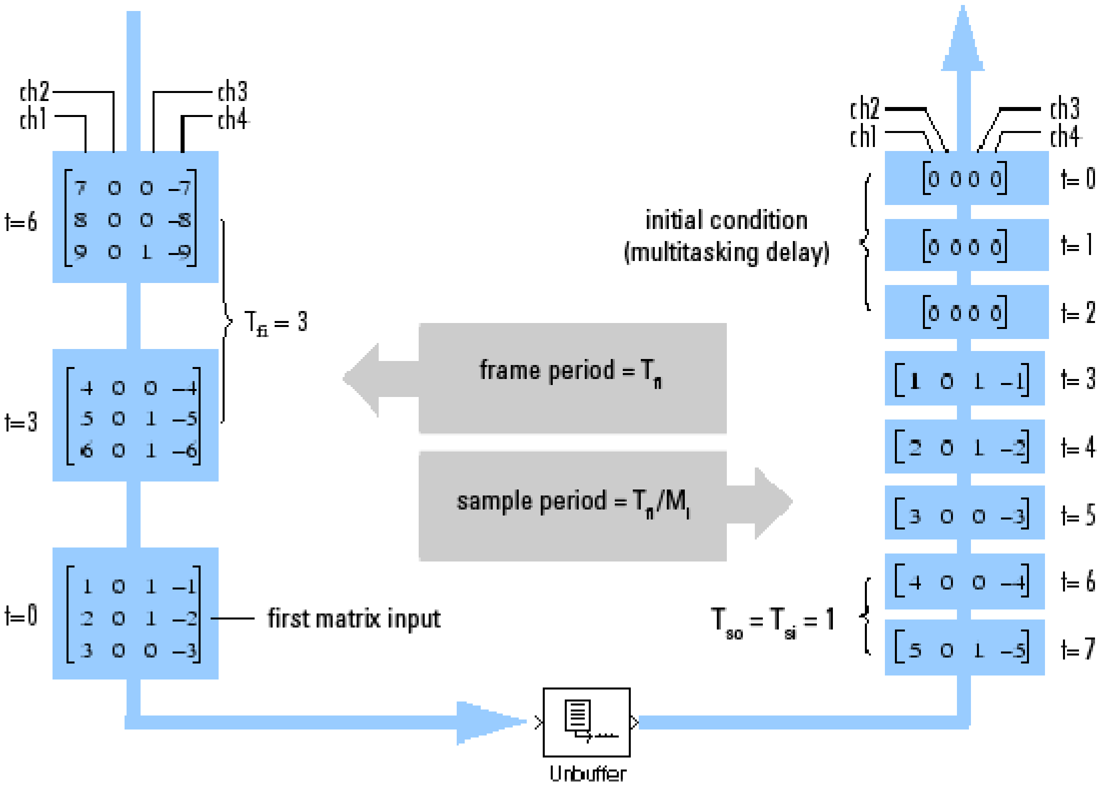
Description

The Unbuffer block unbuffers an M_i-by-N input into a 1-by-N output. That is, inputs are unbuffered *row-wise* so that each matrix row becomes an independent time-sample in the output. The rate at which the block receives inputs is generally less than the rate at which the block produces outputs.



The block adjusts the output rate so that the *sample period* is the same at both the input and output, $T_{so}=T_{si}$. Therefore, the output sample period for an input of frame size M_i and frame period T_{fi} is T_{fi}/M_i, which represents a *rate* M_i times higher than the input frame rate. In the example above, the block receives inputs only once every three sample periods, but produces an output once every sample period. To rebuffer inputs to a larger or smaller frame size, use the `Buffer` block.

In the model below, the block unbuffers a four-channel input with a frame size of three. The **Initial conditions** parameter is set to zero and the tasking mode is set to multitasking, so the first three outputs are zero vectors.



Zero Latency

The Unbuffer block has *zero-tasking latency* in Simulink single-tasking mode. Zero-tasking latency means that the first input sample (received at t=0) appears as the first output sample.

Nonzero Latency

For *multitasking* operation, the Unbuffer block's buffer is initialized with the value specified by the **Initial conditions** parameter, and the block begins unbuffering this frame at the start of the simulation. Inputs to the block are therefore delayed by one buffer length, or M_i samples.

The **Initial conditions** parameter can be one of the following:

- A scalar to be repeated for the first M_i output samples of every channel
- A length-M_i vector containing the values of the first M_i output samples for every channel
- An M_i-by-N matrix containing the values of the first M_i output samples in each of N channels

Note For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

Parameters

Initial conditions

The value of the block's initial output for cases of nonzero latency. You can specify a scalar, vector, or matrix.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

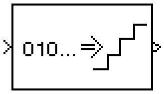
`Buffer`	DSP System Toolbox

See "Unbuffer Frame Signals into Sample Signals" for related information.

Introduced before R2006a

Uniform Decoder

Decode integer input into floating-point output



Library

Quantizers

`dspquant2`

Description

The Uniform Decoder block performs the inverse operation of the Uniform Encoder block, and reconstructs quantized floating-point values from encoded integer input. The block adheres to the definition for uniform decoding specified in ITU-T Recommendation G.701.

Inputs can be real or complex values of the following six integer data types: `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`.

The block first casts the integer input values to floating-point values, and then uniquely maps (decodes) them to one of 2^B uniformly spaced floating-point values in the range [-V, $(1-2^{1-B})$V], where you specify B in the **Bits** parameter (as an integer between 2 and 32) and V is a floating-point value specified by the **Peak** parameter. The smallest input value representable by B bits (0 for an unsigned input data type; -2^{B-1} for a signed input data type) is mapped to the value -V. The largest input value representable by B bits (2^B-1 for an unsigned input data type; 2^{B-1}-1 for a signed input data type) is mapped to the value $(1-2^{1-B})$V. Intermediate input values are linearly mapped to the intermediate values in the range [-V, $(1-2^{1-B})$V].

To correctly decode values encoded by the Uniform Encoder block, the **Bits** and **Peak** parameters of the Uniform Decoder block should be set to the same values as the **Bits** and **Peak** parameters of the Uniform Encoder block. The **Overflow mode** parameter specifies the Uniform Decoder block's behavior when the integer input is outside the range representable by B bits. When you select **Saturate**, *unsigned* input values greater than 2^B-1 saturate at 2^B-1; *signed* input values greater than 2^{B-1}-1 or less than -2^{B-1} saturate at those limits. The real and imaginary components of complex inputs saturate independently.

When you select **Wrap**, *unsigned* input values, `u`, greater than 2^B-1 are wrapped back into the range [0, 2^B-1] using mod-2^B arithmetic.

```
u = mod(u,2^B)
```

Signed input values, `u`, greater than 2^{B-1}-1 or less than -2^{B-1} are wrapped back into that range using mod-2^B arithmetic.

```
u = (mod(u+2^B/2,2^B)-(2^B/2))
```

The real and imaginary components of complex inputs wrap independently.

The **Output type** parameter specifies whether the decoded floating-point output is single or double precision. Either level of output precision can be used with any of the six integer input data types.

Examples

See example model ex_uniform_decoder.

In this example, the input to the block is the `uint8` output of a Uniform Encoder block. This block has comparable settings: **Peak** = `2`, **Bits** = `3`, and **Output type** = `Unsigned`. (Comparable settings ensure that inputs to the Uniform Decoder block do not saturate or wrap. See the example on the Uniform Encoder block reference page for more about these settings.)

The real and complex components of each input are independently mapped to one of 2^3 distinct levels in the range `[-2.0,1.5]`.

```
0  is mapped to  -2.0
1  is mapped to  -1.5
2  is mapped to  -1.0
3  is mapped to  -0.5
4  is mapped to   0.0
5  is mapped to   0.5
```

```
6  is mapped to   1.0
7  is mapped to   1.5
```

Parameters

Peak

Specify the largest amplitude represented in the encoded input. To correctly decode values encoded with the Uniform Encoder block, set the **Peak** parameters in both blocks to the same value.

Bits

Specify the number of input bits, B, used to encode the data. (This can be less than the total number of bits supplied by the input data type.) To correctly decode values encoded with the Uniform Encoder block, set the **Bits** parameters in both blocks to the same value.

Overflow mode

Specify the block's behavior when the integer input is outside the range representable by B bits. Out-of-range inputs can either saturate at the extreme value, or wrap back into range.

Output type

Specify the precision of the floating-point output, `single` or `double`.

References

General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms, International Telecommunication Union, ITU-T Recommendation G.701, March, 1993

Supported Data Types

Port	Supported Data Types
Input	• 8-, 16-, and 32-bit integers
Output	• Double-precision floating point • Single-precision floating point

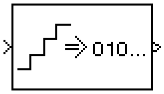
See Also

Data Type Conversion	Simulink
Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
`udecode`	Signal Processing Toolbox
`uencode`	Signal Processing Toolbox

Introduced before R2006a

Uniform Encoder

Quantize and encode floating-point input into integer output



Library

Quantizers

`dspquant2`

Description

The Uniform Encoder block performs the following two operations on each floating-point sample in the input vector or matrix:

1. Quantizes the value using the same precision
2. Encodes the quantized floating-point value to an integer value

In the first step, the block quantizes an input value to one of 2^B uniformly spaced levels in the range $[-V, (1-2^{1-B})V]$, where you specify B in the **Bits** parameter and you specify V in the **Peak** parameter. The quantization process rounds both positive and negative inputs *downward* to the nearest quantization level, with the exception of those that fall exactly on a quantization boundary. The real and imaginary components of complex inputs are quantized independently.

The number of bits, B, can be any integer value between 2 and 32, inclusive. Inputs greater than $(1-2^{1-B})V$ or less than $-V$ saturate at those respective values. The real and imaginary components of complex inputs saturate independently.

In the second step, the quantized floating-point value is uniquely mapped (encoded) to one of 2^B integer values. When the **Output type** is set to `Unsigned integer`, the smallest quantized floating-point value, $-V$, is mapped to the integer 0, and the largest quantized floating-point value, $(1-2^{1-B})V$, is mapped to the integer 2^B-1. Intermediate quantized floating-point values are linearly (uniformly) mapped to the intermediate integers in the range $[0, 2^B\text{-}1]$. For efficiency, the block automatically selects an *unsigned* output data type (`uint8`, `uint16`, or `uint32`) with the minimum number of bits equal to or greater than B.

When the **Output type** is set to `Signed integer`, the smallest quantized floating-point value, $-V$, is mapped to the integer $-2^{B\text{-}1}$, and the largest quantized floating-point value, $(1\text{-}2^{1\text{-}B})V$, is mapped to the integer $2^{B\text{-}1}\text{-}1$. Intermediate quantized floating-point values are linearly mapped to the intermediate integers in the range $[-2^{B\text{-}1}, 2^{B\text{-}1}\text{-}1]$. The block automatically selects a *signed* output data type (`int8`, `int16`, or `int32`) with the minimum number of bits equal to or greater than B.

Inputs can be real or complex, double or single precision. The output data types that the block uses are shown in the table below. Note that most of the DSP System Toolbox blocks accept only double-precision inputs. Use the Simulink Data Type Conversion block to convert integer data types to double precision. See "About Data Types in Simulink" (Simulink) for a complete discussion of data types, as well as a list of Simulink blocks capable of reduced-precision operations.

Bits	Unsigned Integer	Signed Integer
2 to 8	`uint8`	`int8`
9 to 16	`uint16`	`int16`
17 to 32	`uint32`	`int32`

The Uniform Encoder block operations adhere to the definition for uniform encoding specified in ITU-T Recommendation G.701.

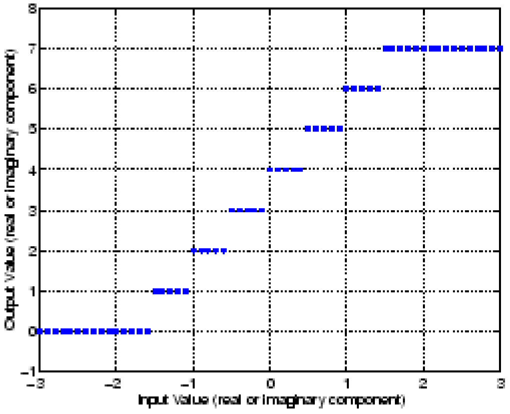
Examples

See example model ex_uniform_encoder.

In this example, the following parameters are set:

- **Peak** = 2
- **Bits** = 3
- **Output type** = `Unsigned`

The following figure illustrates uniform encoding.



The real and complex components of each input (horizontal axis) are independently quantized to one of 2^3 distinct levels in the range `[-2,1.5]`. These components are then mapped to one of 2^3 integer values in the range `[0,7]`.

```
-2.0  is mapped to  0
-1.5  is mapped to  1
-1.0  is mapped to  2
-0.5  is mapped to  3
 0.0  is mapped to  4
 0.5  is mapped to  5
 1.0  is mapped to  6
 1.5  is mapped to  7
```

This table shows the results for a few particular inputs.

Input	Quantized Input	Output	Notes
1.6	1.5+0.0i	7+4i	
-0.4	-0.5+0.0i	3+4i	
-3.2	-2.0+0.0i	4i	Saturation (real)
0.4-1.2i	0.0-1.5i	4+i	
0.4-6.0i	0.0-2.0i	4	Saturation (imaginary)

Input	Quantized Input	Output	Notes
`-4.2+3.5i`	`-2.0+2.0i`	`7i`	Saturation (real and imaginary)

The output data type is automatically set to `uint8`, the most efficient format for this input range.

Parameters

Peak

The largest input amplitude to be encoded, V. Real or imaginary input values greater than $(1\text{-}2^{1-B})V$ or less than $-V$ saturate (independently for complex inputs) at those limits.

Bits

Specify the number of bits, B, needed to represent the integer output. The number of levels at which the block quantizes the floating-point input is 2^B.

Output type

The data type of the block's output, `Unsigned integer` or `Signed integer`. Unsigned outputs are `uint8`, `uint16`, or `uint32`, while signed outputs are `int8`, `int16`, or `int32`.

References

General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms, International Telecommunication Union, ITU-T Recommendation G.701, March, 1993

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
Output	• 8-, 16-, and 32-bit integers

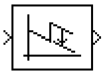
See Also

Data Type Conversion	Simulink
Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Uniform Decoder	DSP System Toolbox
`udecode`	Signal Processing Toolbox
`uencode`	Signal Processing Toolbox

Introduced before R2006a

Unwrap

Unwrap signal phase



Library

Signal Operations

`dspsigops`

Description

The Unwrap block unwraps each channel of the input by adding or subtracting appropriate multiples of 2π to each channel element. The input can be a vector or matrix, and must have radian phase entries. The block recognizes phase discontinuities larger than the **Tolerance** parameter setting. For more information about phase unwrapping, see the “Definition of Phase Unwrap” on page 2-1810.

The block preserves the input size and dimension, and the output port rate equals the input port rate.

Unwrap Method

The Unwrap block unwraps each channel of its input matrix or input vector by adding 2π*k* to each successive channel element, and updating *k* at each *phase jump*. A phase jump occurs when the difference between two adjacent phase value entries exceeds the value of the **Tolerance** parameter.

The following code illustrates how the block unwraps the data in a given input channel `u`.

```
k=0;    % initialize k to 0
i=1;    % initialize the counter to 1
alpha=pi;   % set alpha to the desired Tolerance. In this case, pi
```

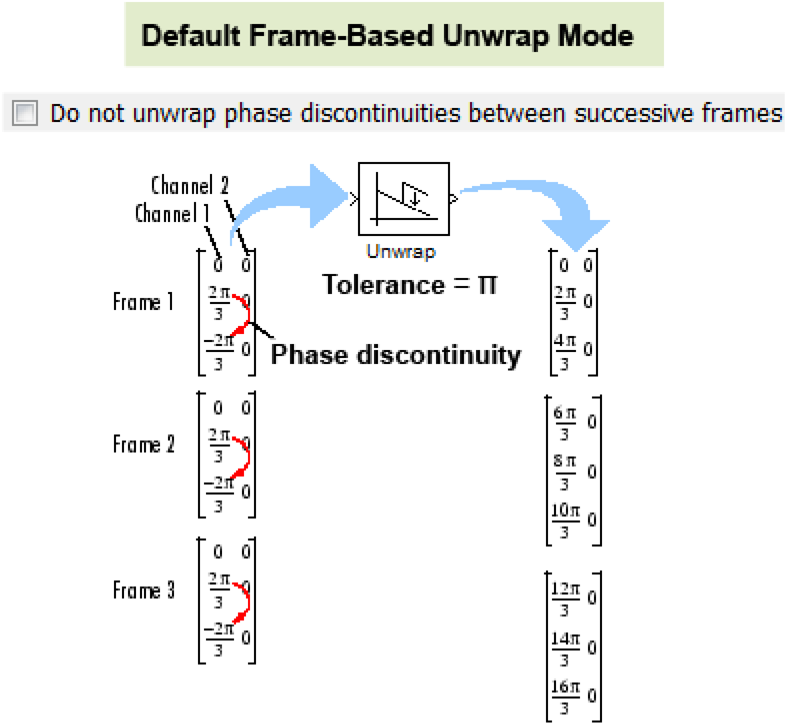
```
for i = 1:(size(u)-1)
    yout(i,:)=u(i)+(2*pi*k); % add 2*pi*k to ui
    if((abs(u(i+1)-u(i)))>(abs(alpha)))  %if diff is greater than alpha, increment or d

        if u(i+1)<u(i)   % if the phase jump is negative, increment k
            k=k+1;
        else             % if the phase jump is positive, decrement k
            k=k-1;
        end
    end
end
yout((i+1),:)=u(i+1)+(2*pi*k); % add 2*pi*k to the last element of the input
```

Frame-Based Processing

When you configure the block to perform frame-based processing, the block supports two different unwrap modes. In both modes, the block adds $2\pi k$ to each input channel's elements, and updates k at each phase discontinuity. The difference between the two modes is how often the block resets the initial phase value (k) to zero. You can choose to unwrap data across frame boundaries (default), or to unwrap only within input frames, by resetting the initial phase value each time a new input frame is received.

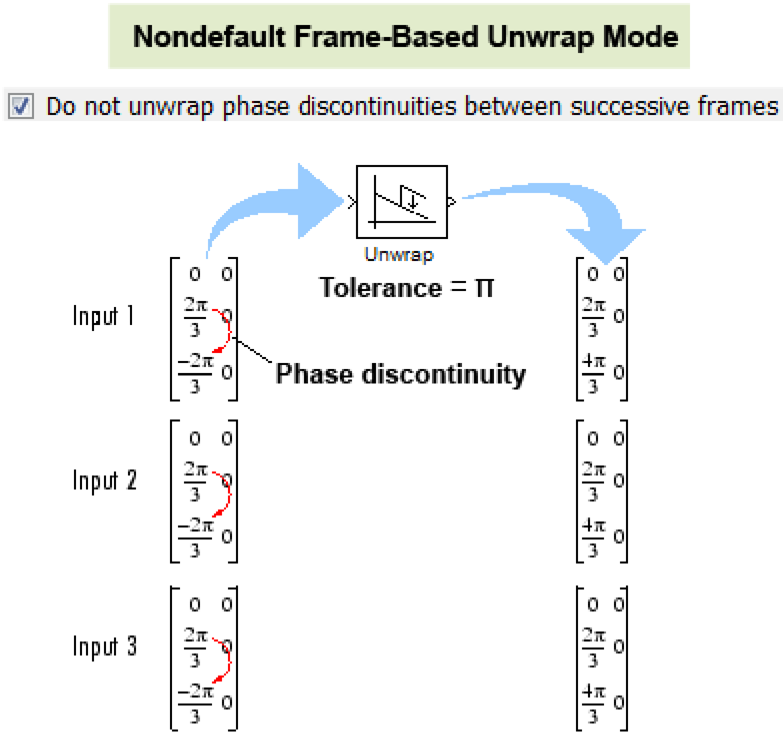
Unwrapping Across Frame Boundaries

In the default mode, the block ignores boundaries between input frames, and continues to unwrap the data in each channel without resetting the initial phase value to zero. To specify this mode, clear the **Do not unwrap phase discontinuities between successive frames** check box. The following figure illustrates how the block unwraps data in this mode.



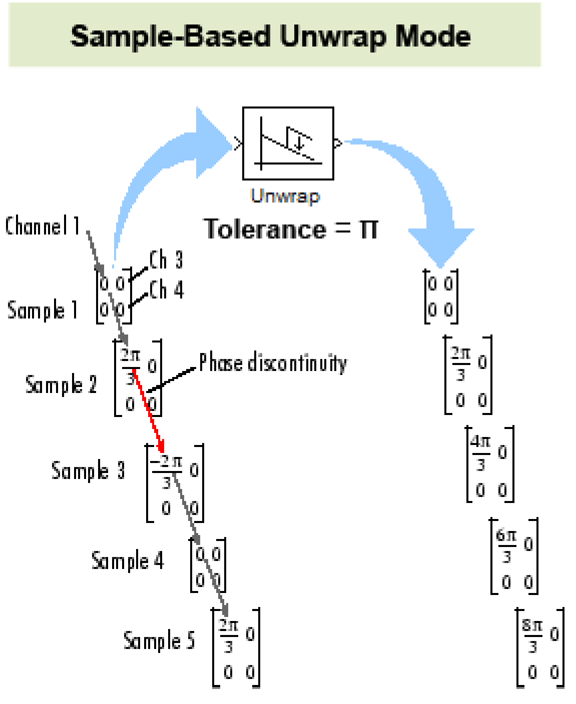
Unwrapping Within Frames

When you select the **Do not unwrap phase discontinuities between successive frames** check box, the block treats each frame of input data independently. In this mode, the block resets the initial phase value to zero each time a new input frame is received. The following figure illustrates how the block unwraps data in this mode.



Sample-Based Processing

When you configure the block to perform sample-based processing, the block treats each element of the input as an individual channel. The block unwraps the data in each channel of the input, and does not reset the initial phase to zero each time a new input is received. The following figure illustrates how the block unwraps data when performing sample-based processing.

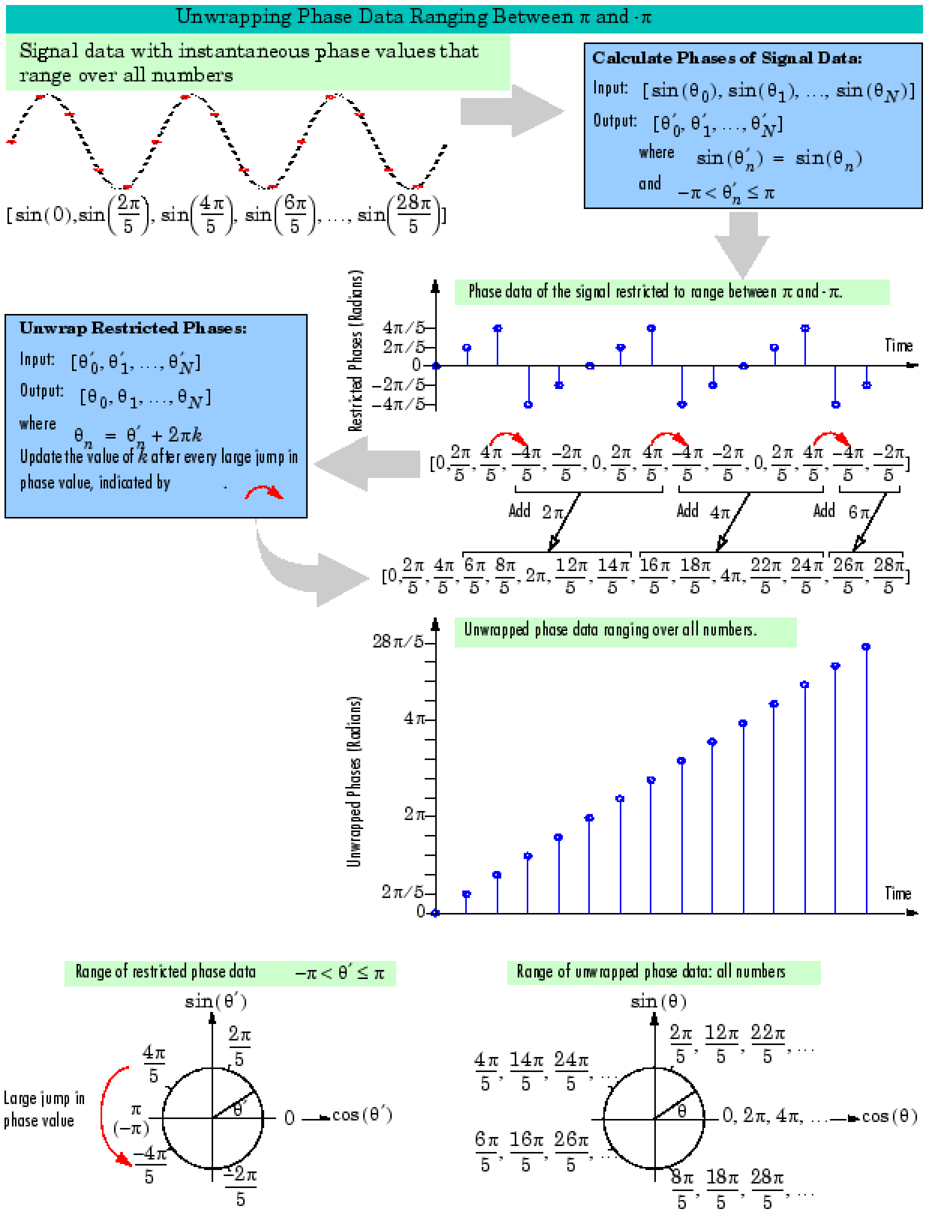


Definition of Phase Unwrap

Algorithms that compute the phase of a signal often only output phases between –π and π. For instance, such algorithms compute the phase of sin(2π + 3) to be 3, since sin(3) = sin(2π + 3), and since the actual phase, 2π + 3, is not between –π and π. Such algorithms compute the phases of sin(–4π + 3) and sin(16π + 3) to be 3 as well.

Phase unwrap or unwrap is a process often used to reconstruct a signal's original phase. Unwrap algorithms add appropriate multiples of 2π to each phase input to restore original phase values, as illustrated in the following diagram. See “Unwrap Method” on page 2-1806 for more information on the unwrap algorithm used by this block.

The following figure illustrates the concept of phase unwrapping.



Unwrapping Phase Data Ranging Between π and -π
Signal data with instantaneous phase values that range over all numbers
[sin(0), sin(2π/5), sin(4π/5), sin(6π/5), ..., sin(28π/5)]
Calculate Phases of Signal Data:
Input: [sin(θ0), sin(θ1), ..., sin(θN)]
Output: [θ'0, θ'1, ..., θ'N]
where sin(θ'n) = sin(θn)
and −π < θ'n ≤ π
Phase data of the signal restricted to range between π and -π.
Restricted Phases (Radians)
Time
Unwrap Restricted Phases:
Input: [θ'0, θ'1, ..., θ'N]
Output: [θ0, θ1, ..., θN]
where θn = θ'n + 2πk
Update the value of k after every large jump in phase value, indicated by
Add 2π
Add 4π
Add 6π
Unwrapped phase data ranging over all numbers.
Unwrapped Phases (Radians)
Range of restricted phase data −π < θ' ≤ π
Large jump in phase value
Range of unwrapped phase data: all numbers

Parameters

Tolerance

The jump size that the block recognizes as a true phase discontinuity. The default is set to π (rather than a smaller value) to avoid altering legitimate signal features. To increase the block's sensitivity, set the **Tolerance** to a value slightly less than π.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Do not unwrap phase discontinuities between successive frames

When you clear this check box, the block ignores boundaries between input frames and does not reset the initial phase value to zero each time a new input is received. In this mode, the block continuously unwraps the data in each column of the input. When you select this check box, the block treats each frame of input data independently, and resets the initial phase value for each new input frame. See the "Frame-Based Processing" on page 2-1807 section for more information.

This parameter is available only when you configure the block to perform frame-based processing. In sample-based processing mode, the block does not reset the initial phase value to zero for each new input. See "Sample-Based Processing" on page 2-1809 for more information.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

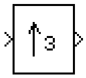
See Also

`unwrap`	MATLAB

Introduced before R2006a

Upsample

Resample input at higher rate by inserting zeros



Library

Signal Operations

`dspsigops`

Description

The Upsample block resamples each channel of the M_i-by-N input at a rate L times higher than the input sample rate by inserting L-1 zeros between consecutive samples. You specify the integer L in the **Upsample factor** parameter. The **Sample offset** parameter, D, allows you to delay the output samples by an integer number of sample periods. Doing so enables you to select any of the L possible output phases. The value you specify for the **Sample offset** parameter must be in the range $0 \leq D < (L-1)$.

You can use this block inside of triggered subsystems when you set the **Rate options** parameter to `Enforce single-rate processing`.

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block upsamples each column of the input over time. In this mode, the block can perform either single-rate or multirate processing. You can use the **Rate options** parameter to specify how the block upsamples the input:

- When you set the **Rate options** parameter to `Enforce single-rate processing`, the input and output of the block have the same sample rate. In this mode, the block outputs a signal with a proportionally larger frame *size* than the input. For upsampling by a factor of L, the output frame size is L times larger than the input frame size ($M_o = M_i*L$), but the input and output frame rates are equal.

For an example of single-rate upsampling, see Example: Single-Rate Processing.

- When you set the **Rate options** parameter to `Allow multirate processing`, the block treats an M_i-by-N matrix input as N independent channels. The block upsamples each column of the input over time by keeping the frame size constant (M_i=M_o), and making the output frame period (T_{fo}) L times shorter than the input frame period ($T_{fo} = T_{fi}/L$).

  See Example: Multirate, Frame-Based Processing for an example that uses the Upsample block in this mode.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an M-by-N matrix input as $M*N$ independent channels, and upsamples each channel over time. In this mode, the block always performs multirate processing. The output sample rate is L times higher than the input sample rate ($T_{so} = T_{si}/L$), and the input and output sizes are identical.

Zero Latency

The Upsample block has *zero-tasking latency* for all single-rate operations. The block is in a single-rate mode if you set the **Upsample factor** parameter to `1` or if you set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`.

The Upsample block also has zero-tasking latency for multirate operations if you run your model in Simulink single-tasking mode.

Zero-tasking latency means that the block propagates the first input (received at t=0) immediately following the D consecutive zeros specified by the **Sample offset** parameter. This output (D+1) is followed in turn by the L-1 inserted zeros and the next input sample.

Nonzero Latency

The Upsample block has tasking latency for multirate, multitasking operation:

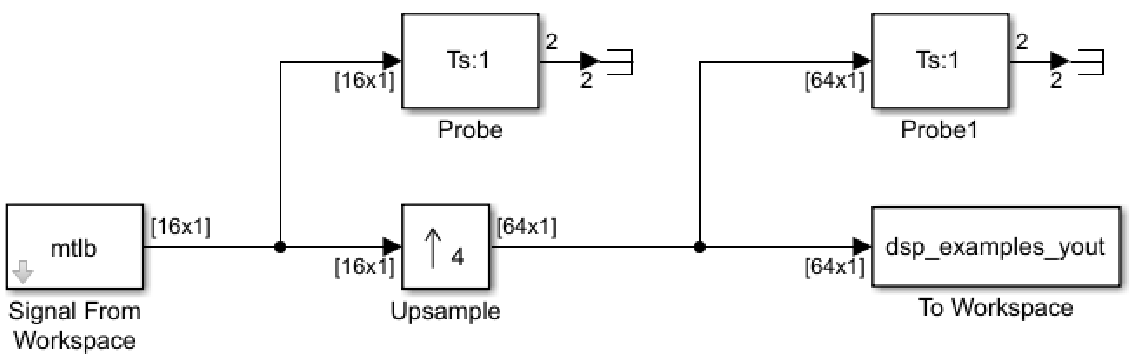
- In multirate, sample-based processing mode, the initial condition for each channel appears as output sample D+1, and is followed by L-1 inserted zeros. The channel's first input appears as output sample D+L+1. The **Initial conditions** parameter can be an M_i-by-N matrix containing one value for each channel, or a scalar to be applied to all signal channels.
- In multirate, frame-based processing mode, the first row of the initial condition matrix appears as output sample D+1, and is followed by L-1 inserted rows of zeros, the second row of the initial condition matrix, and so on. The first row of the first input matrix appears in the output as sample M_iL+D+1. The **Initial conditions** parameter can be an M_i-by-N matrix, or a scalar to be repeated across all elements of the input matrix.

Note: For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Time-Based Scheduling and Code Generation" (Simulink Coder).

Examples

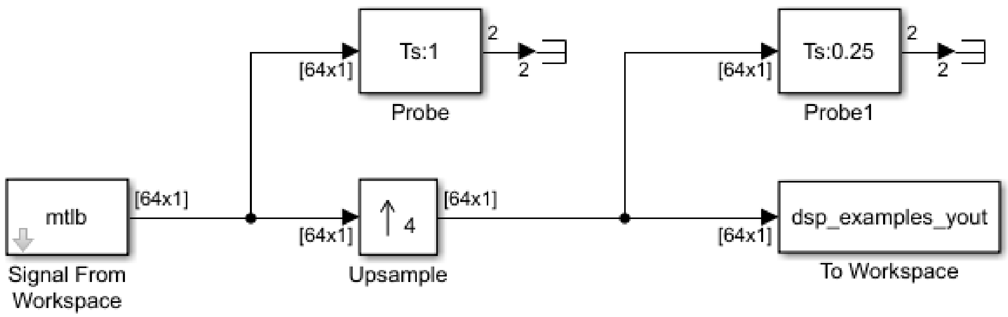
Example: Single-Rate Processing

In the ex_upsample_ref2 model, the Upsample block resamples a single-channel input with a frame size of 16. The block upsamples the input by a factor of 4. Thus, the output of the block has a frame size of 64. Because the block is in single-rate processing mode, the input and output frame rates are identical.



Example: Multirate, Frame-Based Processing

In the ex_upsample_ref1 model, the Upsample block resamples a single-channel input with a frame period of 1 second. The block upsamples the input by a factor of 4. Thus, the output of the block has a frame period of 0.25 seconds. Because the block is in multirate processing mode, the input and output frame sizes are identical.



Parameters

Upsample factor

The integer factor, L, by which to increase the input sample rate.

Sample offset

The sample offset, D, which must be an integer in the range [0,L-1].

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel. In this mode, the block can perform single-rate or multirate processing.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel. In this mode, the block always performs multirate processing.

Rate options

Specify the method by which the block upsamples the input. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate by increasing the output frame size by a factor of *L*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block resamples the signal such that the output sample rate is *L* times faster than the input sample rate.

Initial conditions

The value with which the block is initialized for cases of nonzero latency, a scalar or matrix. This value appears in the output as sample *D*+1. This parameter appears only when you configure the block to perform multirate processing.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information on implementations, properties, and restrictions for HDL code generation, see Upsample.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Port	Supported Data Types
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

`Downsample`	DSP System Toolbox
`FIR Interpolation`	DSP System Toolbox
`FIR Rate Conversion`	DSP System Toolbox
`Repeat`	DSP System Toolbox

Introduced before R2006a

Variable Bandwidth FIR Filter

Design tunable bandwidth FIR filter

Library

Filtering / Filter Designs

`dspfdesign`

Description

The Variable Bandwidth FIR Filter block filters each channel of the input signal over time using specified FIR filter specifications. This block offers tunable filter design parameters, which enable you to tune the filter characteristics while the simulation is running.

The block designs the FIR filter according to the filter parameters set in the block dialog box. The output port properties, such as datatype, complexity, and dimension, are identical to the input port properties.

Each column of the input signal is treated as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then it is interpreted as a single channel.

This block supports variable-size input, enabling you to change the channel length during simulation. To enable variable-size input, clear the **Inherit sample rate from input** check box. The number of channels must remain constant.

Algorithms

This block brings the capabilities of dsp.VariableBandwidthFIRFilter System object to the Simulink environment.

The FIR filter is designed using the window method. For information on the algorithms used by the Variable Bandwidth FIR Filter block, see the "Algorithms" on page 4-1853 section of `dsp.VariableBandwidthFIRFilter`.

Examples

- "System Identification Using RLS Adaptive Filtering"

Parameters

FIR filter order

Order of the FIR filter, specified as a positive integer scalar. The default is `30`. This parameter is nontunable.

Filter type

Type of FIR filter. You can set this parameter to:

- `Lowpass` (default)
- `Highpass`
- `Bandpass`
- `Bandstop`

This parameter is nontunable.

Filter Cutoff frequency (Hz)

Cutoff frequency of the FIR filter, specified as a real positive scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter type** to `Lowpass` or `Highpass`. The default is `1000`. This parameter is tunable.

Filter center frequency (Hz)

Center frequency of the FIR filter, specified as a real positive scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter type** to `Bandpass` or `Bandstop`. The default is `10000`. This parameter is tunable.

Filter bandwidth (Hz)

Bandwidth of the FIR filter, specified as a real positive scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter type** to `Bandpass` or `Bandstop`. The default is `2000`. This parameter is tunable.

Window function

Window function used to design the FIR filter. You can set this parameter to:

- `Hann` (default)
- `Hamming`
- `Chebyshev`
- `Kaiser`

This parameter is nontunable.

Chebyshev window sidelobe attenuation (dB)

Sidelobe attenuation of chebyshev window, specified as a real positive scalar. This parameter applies when you set **Window function** to `Chebyshev`. The default is `60`. This parameter is nontunable.

Kaiser window parameter

Kaiser window parameter, specified as a real scalar. This parameter applies when you set **Window function** to `Kaiser`. The default is `0.5`. This parameter is nontunable.

Inherit sample rate from input

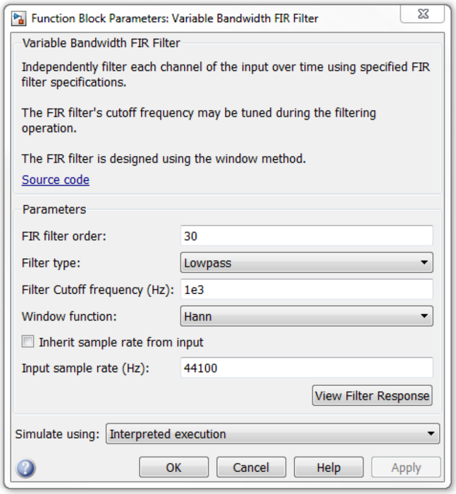
When you select this check box, the block's sample rate is computed as N / Ts, where N is the frame size of the input signal and Ts is the sample time of the input signal. When you clear this check box, the block's sample rate is the value specified in **Input sample rate (Hz)**. By default, this check box is selected.

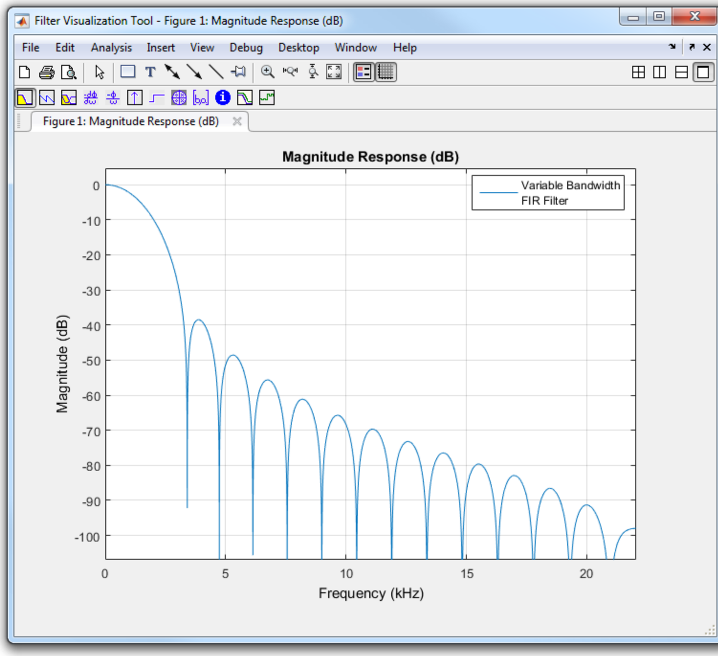
Input sample rate (Hz)

Sample rate of the input signal, specified as a positive scalar. The default is `44100`. This parameter applies when you clear the **Inherit sample rate from input** check box. This parameter is nontunable.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the Variable Bandwidth FIR Filter. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.





To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

- `Interpreted execution` (default)

  Simulate model using the MATLAB interpreter. This option has faster simulation speed compared to `Code generation`.

- `Code generation`

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time compared to `Interpreted execution`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

References

[1] Jarske, P.,Y. Neuvo, and S. K. Mitra. *A simple approach to the design of linear phase FIR digital filters with variable characteristics* Signal Processing. Vol. 14, Issue 4, June 1988, pp. 313-326.

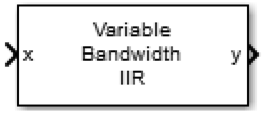
See Also

`Biquad Filter`	DSP System Toolbox
`Variable Bandwidth IIR Filter`	DSP System Toolbox
dsp.VariableBandwidthFIRFilter	DSP System Toolbox
dsp.VariableBandwidthIIRFilter	DSP System Toolbox

Introduced in R2015a

Variable Bandwidth IIR Filter

Design tunable bandwidth IIR filter



Library

Filtering / Filter Designs

`dspfdesign`

Description

The Variable Bandwidth IIR Filter block filters each channel of the input signal over time using specified IIR filter specifications. This block offers tunable filter design parameters, which enable you to tune the filter characteristics while the simulation is running.

The block designs the IIR filter according to the filter parameters set in the block dialog box. The output port properties, such as datatype, complexity, and dimension, are identical to the input port properties.

Each column of the input signal is treated as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size) and the second dimension represents the number of channels. If the input is a one-dimensional signal, then it is interpreted as a single channel.

This block supports variable-size input, enabling you to change the channel length during simulation. To enable variable-size input, clear the **Inherit sample rate from input** check box. The number of channels must remain constant.

Algorithms

This block brings the capabilities of dsp.VariableBandwidthIIRFilter System object to the Simulink environment.

The IIR filter is designed using the elliptical method. The IIR filter is tuned using IIR spectral transformations based on allpass filters. For more information on the algorithms used by the Variable Bandwidth IIR Filter block, see the “Algorithms” on page 4-1863 section of `dsp.VariableBandwidthIIRFilter`.

Examples

- “Tunable Lowpass Filtering of Noisy Input in Simulink”

Parameters

Filter type

Type of IIR filter. You can set this parameter to:

- `Lowpass` (default)
- `Highpass`
- `Bandpass`
- `Bandstop`

This parameter is nontunable.

IIR filter order

Order of the IIR filter, specified as a positive integer scalar. The default is `8`. This parameter is nontunable.

Filter passband frequency (Hz)

Passband frequency of the IIR filter, specified as a real positive scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter type** to `Lowpass` or `Highpass`. The default is `1000`. This parameter is tunable.

Filter center frequency (Hz)

Center frequency of the IIR filter, specified as a real positive scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter type** to `Bandpass` or `Bandstop`. The default is `10000`. This parameter is tunable.

Filter bandwidth (Hz)

Bandwidth of the IIR filter, specified as a real positive scalar that is less than half the sample rate of the input signal. This parameter applies when you set **Filter type** to `Bandpass` or `Bandstop`. The default is `2000`. This parameter is tunable.

Filter passband ripple (dB)

Passband ripple of the IIR filter, specified as a real positive scalar. The default is `1`. This parameter is nontunable.

Filter Stopband attenuation (dB)

Stopband attenuation of the IIR filter, specified as a real positive scalar. The default is `60`. This parameter is nontunable.

Inherit sample rate from input

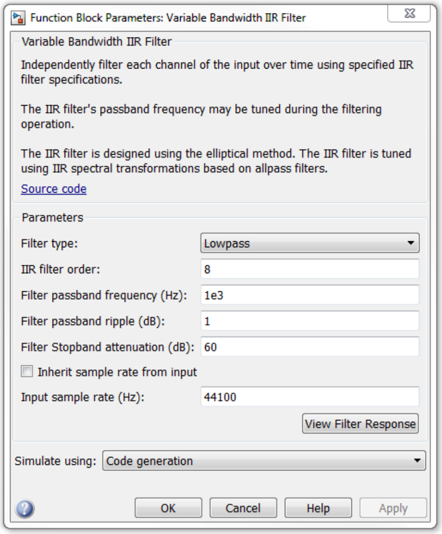
When you select this check box, the block's sample rate is computed as N / Ts, where N is the frame size of the input signal and Ts is the sample time of the input signal. When you clear this check box, the block's sample rate is the value specified in **Input sample rate (Hz)**. By default, this check box is selected.

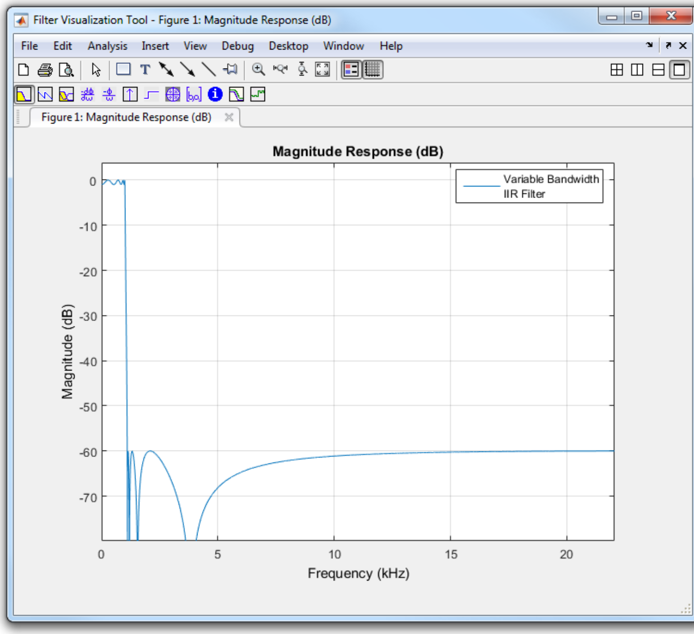
Input sample rate (Hz)

Sample rate of the input signal, specified as a positive scalar. The default is `44100`. This parameter applies when you clear the **Inherit sample rate from input** check box. This parameter is nontunable.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the Variable Bandwidth IIR Filter. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.





To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

Simulate using

Type of simulation to run. You can set this parameter to:

- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

- `Interpreted execution`

  Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

References

[1] A. G. Constantinides. *Spectral transformations for digital filters*, Proc. Inst. Elect. Eng. Vol. 117, No. 8, 1970, pp. 1585-1590.

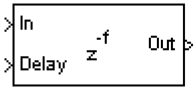
See Also

`Biquad Filter`	DSP System Toolbox
`Variable Bandwidth FIR Filter`	DSP System Toolbox
dsp.VariableBandwidthFIRFilter	DSP System Toolbox
dsp.VariableBandwidthIIRFilter	DSP System Toolbox

Introduced in R2015a

Variable Fractional Delay

Delay input by time-varying fractional number of sample periods



Library

Signal Operations

`dspsigops`

Description

The Variable Fractional Delay block delays each element of the discrete-time N-D input array, u, by a variable number of sample intervals. The input delay values can be integer or noninteger values. The block provides three different interpolation modes: `Linear`, `FIR`, and `Farrow`.

The block computes the value for each channel of the output based on the stored samples in memory most closely indexed by the `Delay` input, v, and the interpolation method specified by the **Interpolation mode** parameter.

- In `Linear` interpolation mode, the block stores the D_{max}+1 most recent samples received at the In port for each channel, where D_{max} is the value you specify for the **Maximum delay (Dmax) in samples** parameter.
- In `FIR` interpolation mode, the block stores the D_{max}+P+1 most recent samples received at the In port for each channel, where P is the value you specify for the **Interpolation filter half-length (P)** parameter.
- In `Farrow` interpolation mode, the block stores the $D_{max}+\frac{N}{2}+1$ most recent samples received at the In port for each channel, where N is the value you specify for the **Farrow filter length (N)** parameter.

The Variable Fractional Delay block assumes that the input values at the `Delay` port are between D_{min} and D_{max}, where D_{min} appears in the `Valid delay range` section on the

Main pane of the block mask, and D_{max} is the value of the **Maximum delay (Dmax) in samples** parameter. The block clips delay values less than D_{min} to D_{min} and delay values greater than D_{max} to D_{max}.

You must consider additional factors when selecting valid `Delay` values for the `FIR` and `Farrow` interpolation modes. For more information about these considerations, refer to or , respectively.

The Variable Fractional Delay block is similar to the Variable Integer Delay block, in that they both store a minimum of D_{max}+1 past samples in memory. The Variable Fractional Delay block differs only in the way that these stored samples are accessed; a fractional delay requires the computation of a value by interpolation from the nearby samples in memory.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats each element of the N-D input array, u, as an independent channel. The input to the Delay port, v, must either be an N-D array of the same size and dimension as the input u, or be a scalar value, such that $D_{min} \leq v \leq D_{max}$.

For example, consider an M-by-N input matrix. The block treats each of the M*N matrix elements as independent channels. The input to the Delay port can be an M-by-N matrix of floating-point values in the range $D_{min} \leq v \leq D_{max}$ that specifies the number of sample intervals to delay each channel of the input, or it can be a scalar floating-point value, $D_{min} \leq v \leq D_{max}$, by which to equally delay all channels.

In sample-based processing mode, the block treats an unoriented vector input as an M-by-1 matrix. In this mode, the output is also an unoriented vector.

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation in the same manner as the Variable Integer Delay block. See the Variable Integer Delay block reference page for more information.

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each of the N input columns as an independent channel containing M_i sequential time samples.

The input to the Delay port, v, contains floating-point values that specify the number of sample intervals to delay the current input.

The input to the Delay port can be a scalar value to uniformly delay every sample in every channel. It can also be a length-M column vector, containing one delay for each sample in the input frame. The block applies the set of delays contained in the vector identically to every channel of a multichannel input. The Delay port entry can also be a length-N row vector, containing one delay for each channel. Finally, the Delay port entry can be an M-by-N matrix, containing a different delay for each corresponding element of the input.

For example, if v is the M_i-by-1 matrix `[v(1) v(2) ... v(Mi)]'`, the earliest sample in the current frame is delayed by `v(1)` fractional sample intervals, the following sample in the frame is delayed by `v(2)` fractional sample intervals, and so on. The block applies the set of fractional delays contained in v identically to every channel of a multichannel input.

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation in the same manner as the Variable Integer Delay block. See the Variable Integer Delay block reference page for more information.

Interpolation Modes

The delay value specified at the Delay port serves as an index into the block's memory, `U`, which stores, at a minimum, the D_{max}+1 most recent samples received at the In port for each channel. For example, an integer delay of `5` on a scalar input sequence retrieves and outputs the fifth most recent input sample from the block's memory, `U(6)`. The block computes fractional delays by interpolating between stored samples; the three available interpolation modes are `Linear`, `FIR` and `Farrow`.

Linear Interpolation Mode

For noninteger delays, at each sample time, the `Linear Interpolation` mode uses the two samples in memory nearest to the specified delay to compute a value for the sample at that time. If v is the specified fractional delay for a scalar input, the output sample, y, is computed as follows.

```
vi = floor(v)     % vi = integer delay
vf = v-vi     % vf = fractional delay
y = (1-vf)*U(vi+1) + vf*U(vi)
```

FIR Interpolation Mode

In `FIR Interpolation` mode, the block provides a discrete set of fractional delays described by:

$$v + \frac{i}{L}, \qquad v \geq P - 1, \;\; i = 0, 1, \ldots, L-1$$

If v is less than P-1, the block's behavior depends on the setting of the **For small input delay values** parameter. You can specify the block's behavior when the input delay value is too small to center the kernel (less than P-1), by setting the **For small input delay values** parameter:

- If you select `Clip to the minimum value necessary for centered kernel`, the block remains in `FIR` interpolation mode by clipping small input delay values to the smallest value necessary to center the kernel.

  To determine the minimum delay value, select `Clip to the minimum value necessary for centered kernel`, and click **Apply** on the block mask. All input delay values less than the value displayed for D_{min} will be clipped to D_{min}.
- If you select `Switch to linear interpolation if kernel cannot be centered`, the block computes fractional delays using linear interpolation when the input delay value is less than P-1.

To add an extra delay to the minimum possible delay value, select the **Disable direct feedthrough by increasing minimum possible delay by one** check box. Checking this box prevents algebraic loops from occurring when you use the block inside a feedback loop.

In `FIR Interpolation` mode, the block implements a polyphase structure to compute a value for each sample at the desired delay. Each arm of the structure corresponds to a different delay value and the output computed for each sample corresponds to the output of the arm with a delay value nearest to the desired input delay. Thus, only a discrete set of delays is actually possible. The number of coefficients in each of the L filter arms of the polyphase structure is $2P$. In most cases, using values of P between 4 and 6 will provide you with reasonably accurate interpolation values.

In this mode, the Signal Processing Toolbox `intfilt` function computes an FIR filter for interpolation.

For example, when you set the parameters on the block mask to the following values:

- Interpolation filter half-length (P): 4
- Interpolation points per input sample: 10
- Normalized input bandwidth: 1

The filter coefficients are given by:

```
b = intfilt(10,4,1);
```

The block then implements this filter as a polyphase structure, as described previously.

Increasing the **Interpolation filter half length (P)** increases the accuracy of the interpolation, but also increases the number of computations performed per input sample, as well as the amount of memory needed to store the filter coefficients. Increasing the **Interpolation points per input sample** (L) increases the number of representable discrete delay points, but also increases the simulation's memory requirements and does not affect the computational load per sample.

The **Normalized input bandwidth (0 to 1)** parameter allows you to take advantage of the bandlimited frequency content of the input. For example, if you know that the input signal does not have frequency content above $F_s/4$, you can specify a value of `0.5` for the **Normalized input bandwidth (0 to 1)** to constrain the frequency content of the output to that range.

Note You can consider each of the L interpolation filters to correspond to one output phase of an "upsample-by-L" FIR filter. Thus, the **Normalized input bandwidth (0 to 1)** value improves the stopband in critical regions, and relaxes the stopband requirements in frequency regions where there is no signal energy.

Farrow Interpolation Mode

In `Farrow` interpolation mode, the block uses the LaGrange method to interpolate values.

To increase the minimum possible delay value, select the **Disable direct feedthrough by increasing minimum possible delay by one** check box. Checking this box prevents algebraic loops from occurring when you use the block inside a feedback loop.

To specify the block's behavior when the input delay value is too small to center the kernel (less than $\frac{N}{2}$-1), set the **For small input delay values** parameter:

- If you select `Clip to the minimum value necessary for centered kernel`, the block clips small input delay values to the smallest value necessary to keep the kernel centered. This increases D_{min} but yields more accurate interpolation values.

  To determine the minimum delay value, select `Clip to the minimum value necessary for centered kernel`, and click **Apply** on the block mask. All input delay values less than the value displayed for D_{min} will be clipped to D_{min}.
- If you select `Use off-centered kernel`, the block computes fractional delays using a Farrow filter with an off-centered kernel. This mode does not increase D_{min}, but if there are input delay values less than $\frac{N}{2}$-1, the results are less accurate than the results achieved by keeping the kernel centered.

Fixed-Point Data Types

The diagrams in the following sections show the data types used within the Variable Fractional Delay block for fixed-point signals.

Although you can specify most of these data types on the **Data Types** pane of the block mask, the following data types are computed internally by the block and cannot be directly specified on the block mask.

Data Type	Word Length	Fraction Length
vf data type	Same word length as the **Coefficients**	Same as the word length
HoldInteger data type	Same word length as the input delay value	0 bits
Integer data type	32 bits	0 bits

Note: When the block input is fixed point, all internal data types are signed fixed point.

To compute the integer (v_i) and fractional (v_f) parts of the input delay value (v), the Variable Fractional Delay block uses the following equations:

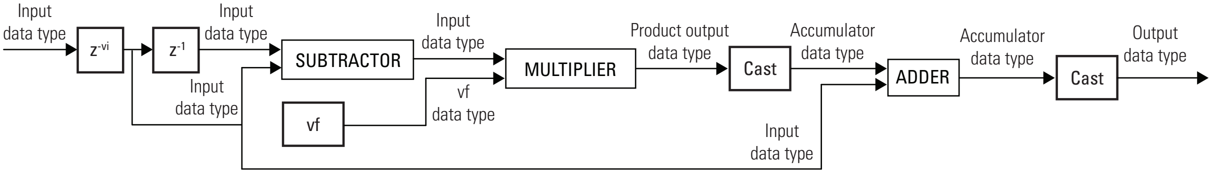
$$D_{min} < v < D_{max} \Rightarrow \begin{cases} v_i = \text{floor}(v) \\ v_f = v - vi \end{cases}$$

$$v \leq D_{min} \Rightarrow \begin{cases} v_i = D_{min} \\ v_f = 0 \end{cases}$$

$$v \geq D_{max} \Rightarrow \begin{cases} v_i = D_{max} \\ v_f = 0 \end{cases}$$

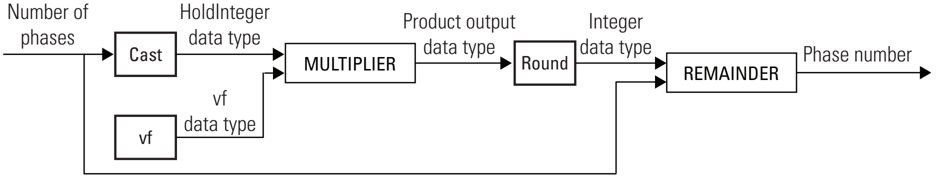
Linear Interpolation Mode

The following diagram shows the fixed-point data types used by the `Linear` interpolation mode of the Variable Fractional Delay block.



FIR Interpolation Mode

The following diagram illustrates how the Variable Fractional Delay block selects the arm of the polyphase filter structure that most closely matches the fractional delay value (v_f).

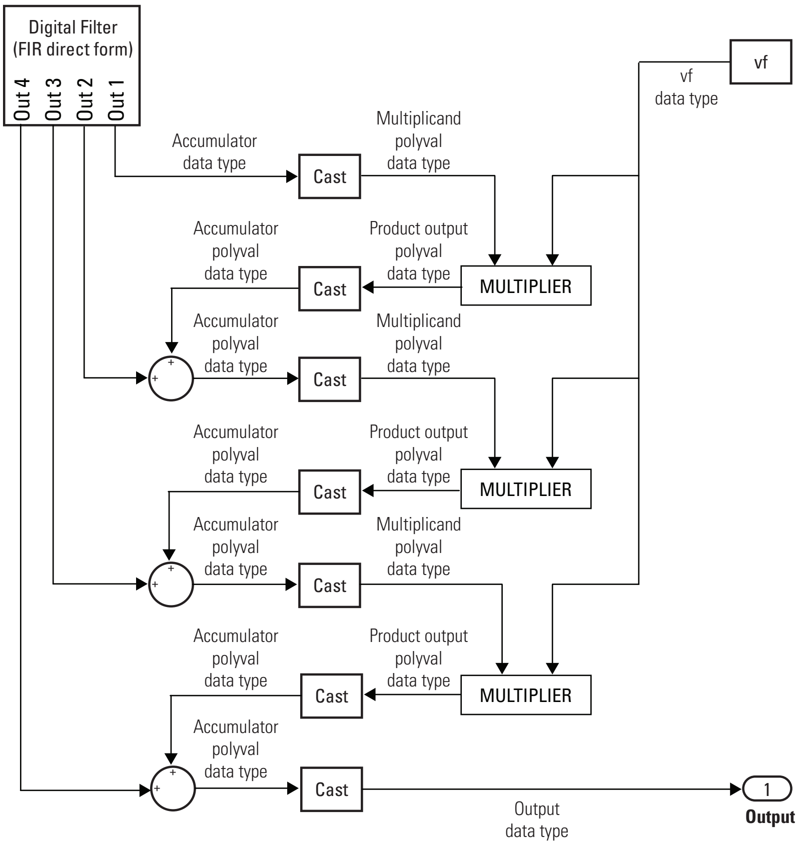


See the "Fixed-Point Data Types" on page 2-723 section of the FIR Interpolation reference page for a diagram showing the fixed-point data types used by the Variable Fractional Delay block in `FIR` interpolation mode.

Farrow Interpolation Mode

The following diagram shows the fixed-point data types used by the `Farrow` interpolation mode of the Variable Fractional Delay block where:

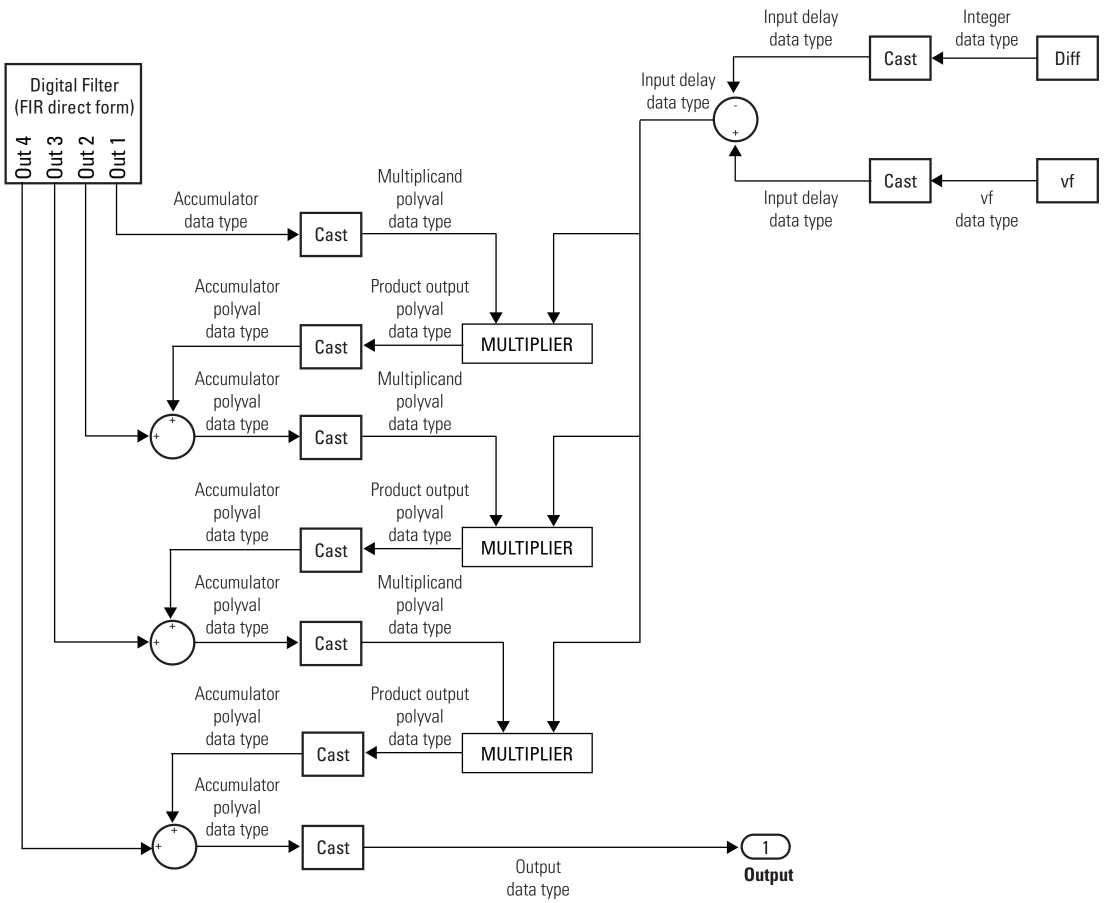
- **Farrow filter length (N)** = 4
- **For small input delay values** = `Clip to the minimum value necessary to for centered kernel`



The following diagram shows the fixed-point data types used by the `Farrow` interpolation mode of the Variable Fractional Delay block where:

- **Farrow filter length (N)** = 4

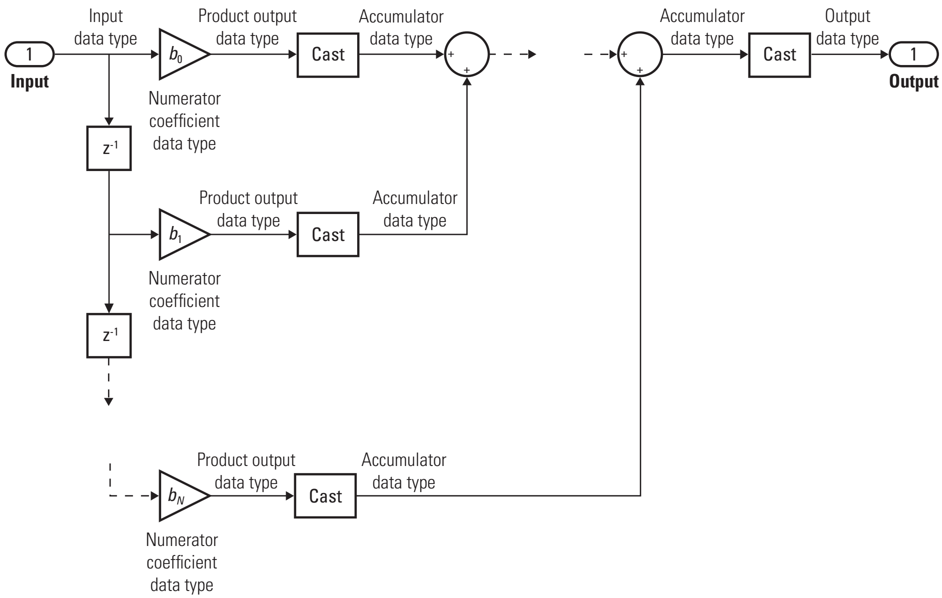
- **For small input delay values** = `Use off-centered kernel`



`Diff` is computed from the integer part of the delay value (v_i) and the **Farrow filter length (N)** according to the following equation:

$$Diff = v_i - \left(\frac{N-1}{2}\right)$$
$$Diff \geq 0 \Rightarrow Diff = 0$$
$$Diff < 0 \Rightarrow Diff = -Diff$$

The following diagram shows the fixed-point data types used by the Digital Filter block's FIR direct form filter.

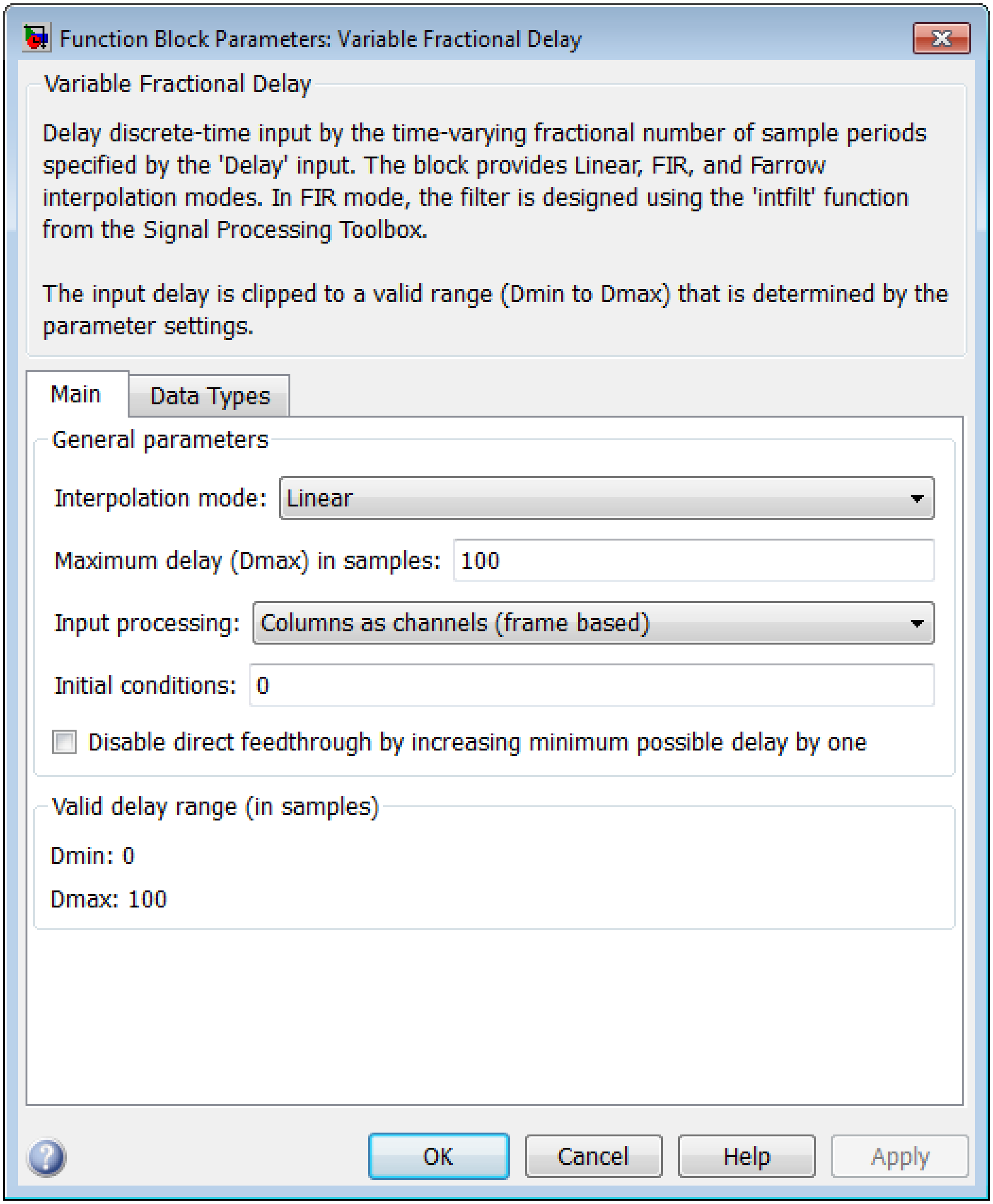


Examples

- "Fractional Delay Filters Using Farrow Structures"

Dialog Box

The **Main** pane of the Variable Fractional Delay block dialog appears as follows.


Function Block Parameters: Variable Fractional Delay
Variable Fractional Delay
Delay discrete-time input by the time-varying fractional number of sample periods specified by the 'Delay' input. The block provides Linear, FIR, and Farrow interpolation modes. In FIR mode, the filter is designed using the 'intfilt' function from the Signal Processing Toolbox.
The input delay is clipped to a valid range (Dmin to Dmax) that is determined by the parameter settings.
Main
Data Types
General parameters
Interpolation mode:
Linear
Maximum delay (Dmax) in samples:
100
Input processing:
Columns as channels (frame based)
Initial conditions:
0
Disable direct feedthrough by increasing minimum possible delay by one
Valid delay range (in samples)
Dmin: 0
Dmax: 100
OK
Cancel
Help
Apply

Interpolation mode

The method by which to interpolate between adjacent stored samples to obtain a value for the sample indexed by the input at the Delay port.

Interpolation filter half-length (P)

Half the number of input samples to use in the FIR interpolation filter. This parameter is only visible when the **Interpolation mode** is set to `FIR`.

Farrow filter length (N)

The number of input samples to use in the Farrow interpolation filter. This parameter is only visible when the **Interpolation mode** is set to `Farrow`.

Interpolation points per input sample

The number of points per input sample, L, at which a unique FIR interpolation filter is computed. This parameter is only visible when the **Interpolation mode** is set to `FIR`.

Normalized input bandwidth (0 to 1)

The bandwidth to which the interpolated output samples should be constrained. The value must be a real scalar between `0` and `1`. A value of `1` specifies half the sample frequency. This parameter is only visible when the **Interpolation mode** is set to `FIR`.

Maximum delay (Dmax) in samples

The maximum delay that the block can produce, D_{max}. Input delay values exceeding this maximum are clipped to D_{max}.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Initial conditions

The values with which the block's memory is initialized. See the Variable Integer Delay block for more information.

Disable direct feedthrough by increasing minimum possible delay by one

Select this box to disable direct feedthrough by increasing the minimum possible delay value. When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block increases the minimum possible delay value by *`frame size`* -1. Similarly, when you set the **Input processing** parameter to `Elements as channels (sample based)`, the block increases the minimum possible delay value by one sample.

Checking this box allows you to use the Variable Fractional Delay block in feedback loops.

For small input delay values

Specify the block's behavior when the input delay values are too small to center the kernel. This parameter is only visible when the **Interpolation mode** is set to `FIR` or `Farrow`.

You can specify how the block handles input delay values that are too small for the kernel to be centered using one of the following choices:

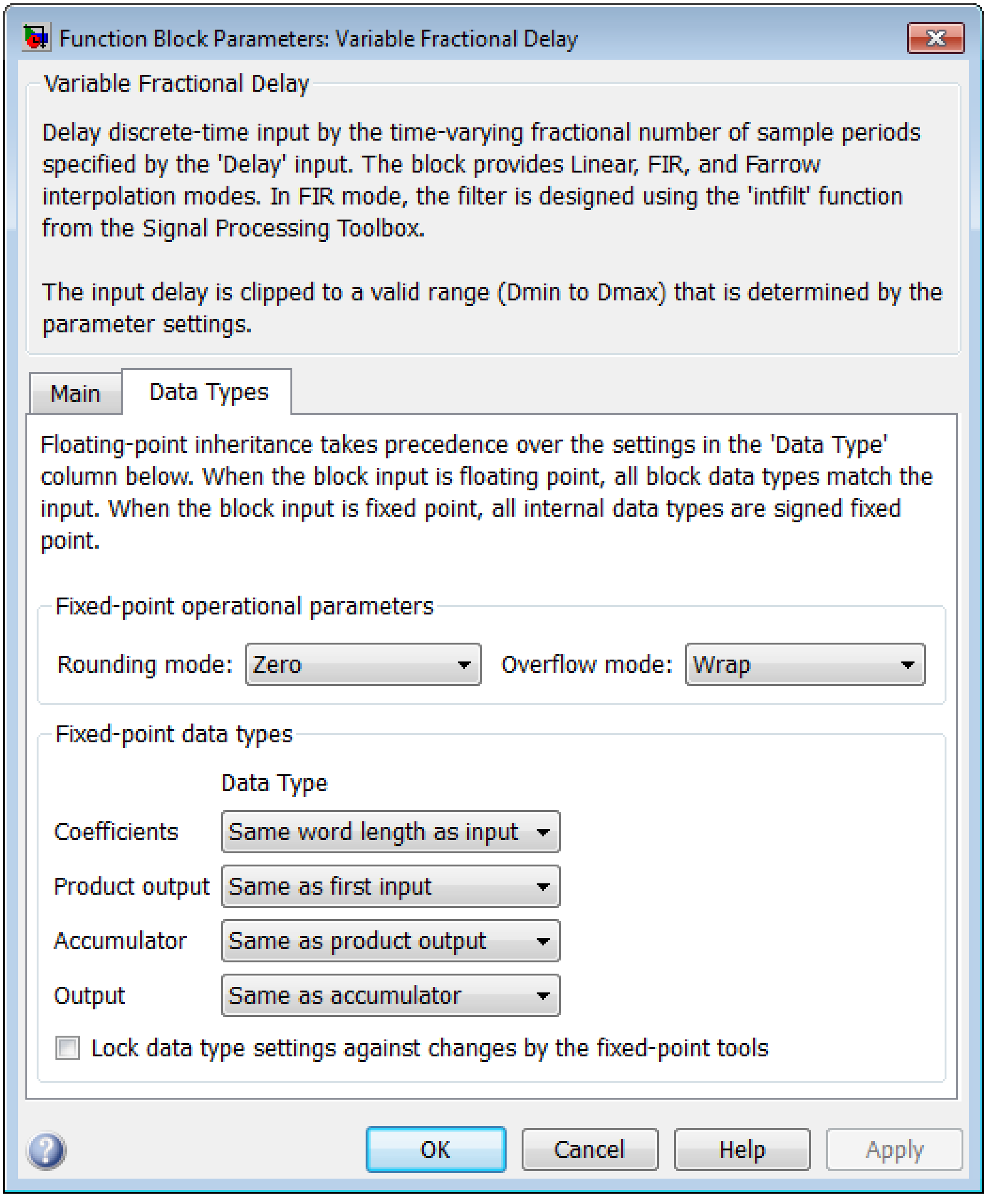
- In both `FIR` and `Farrow` interpolation modes, you can select `Clip to the minimum value necessary for centered kernel`. This option forces the block to increase D_{min} to the smallest value necessary to keep the kernel centered.
- In `FIR` interpolation mode, you can select `Switch to linear interpolation if kernel cannot be centered`. This option forces the block to preserve the value of D_{min} and compute all interpolated values using `Linear` interpolation.
- In `Farrow` interpolation mode, you can select `Use off-centered kernel`. This option forces the block to preserve the value of D_{min} and compute the interpolated values using a farrow filter with an off-centered kernel.

Valid delay range

The values displayed in this section of the **Main** pane are calculated (in samples) by the block based on the current parameter settings.

- `Dmin` is the smallest possible valid delay value (in samples) based on the current settings of the block parameters. The block clips all input delay values less than `Dmin` to `Dmin`.
- `Dmax` is the maximum valid delay value (in samples) based on the current settings of the block parameters. The block clips all input delay values greater than `Dmax` to `Dmax`.

The **Data Types** pane of the Variable Fractional Delay block dialog appears as follows.


Function Block Parameters: Variable Fractional Delay
Variable Fractional Delay
Delay discrete-time input by the time-varying fractional number of sample periods specified by the 'Delay' input. The block provides Linear, FIR, and Farrow interpolation modes. In FIR mode, the filter is designed using the 'intfilt' function from the Signal Processing Toolbox.
The input delay is clipped to a valid range (Dmin to Dmax) that is determined by the parameter settings.
Main
Data Types
Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input. When the block input is fixed point, all internal data types are signed fixed point.
Fixed-point operational parameters
Rounding mode:
Zero
Overflow mode:
Wrap
Fixed-point data types
Data Type
Coefficients
Same word length as input
Product output
Same as first input
Accumulator
Same as product output
Output
Same as accumulator
Lock data type settings against changes by the fixed-point tools
OK
Cancel
Help
Apply

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Coefficients

Choose how you specify the word length and fraction length of the filter coefficients.

- When you select `Same word length as input`, the word length of the filter coefficients match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Specify word length`, you can enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See "Fixed-Point Data Types" on page 2-1835 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block.

- When you select `Same as first input`, these characteristics match those of the first input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the product output, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See "Fixed-Point Data Types" on page 2-1835 and "Multiplication Data Types" for illustrations depicting the use of the accumulator data type in this block:

- When you select `Same as product output`, these characteristics match those of the product output.

- When you select `Same as first input`, these characteristics match those of the first input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Product output polyval

Choose how you specify the word length and fraction length of the product output polyval data type. This parameter is only visible when the **Interpolation mode** is set to `Farrow`.

- When you select `Same as first input`, these characteristics match those of the first input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the product output polyval in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the product output polyval. This block requires power-of-two slope and a bias of zero.

Accumulator polyval

Choose how you specify the word length and fraction length of the accumulator polyval data type. This parameter is only visible when the **Interpolation mode** is set to `Farrow`.

- When you select `Same as first input`, these characteristics match those of the first input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the accumulator polyval in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the accumulator polyval. This block requires power-of-two slope and a bias of zero.

Multiplicand polyval

Choose how you specify the word length and fraction length of the multiplicand polyval data type. This parameter is only visible when the **Interpolation mode** is set to `Farrow`.

- When you select `Same as first input`, these characteristics match those of the first input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the multiplicand polyval, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the multiplicand polyval. This block requires power-of-two slope and a bias of zero.

Output

Choose how you specify the output word length and fraction length:

- When you select `Same as accumulator`, these characteristics match those of the accumulator.
- When you select `Same as first input`, these characteristics match those of the first input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the output, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Delay	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
	• Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

Delay	Simulink
Unit Delay	Simulink
Variable Integer Delay	Simulink

Introduced before R2006a

Variable Integer Delay

Delay input by time-varying integer number of sample periods

Library

`dspsigops`

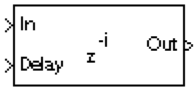
Description

The Variable Integer Delay block is an implementation of the Simulink Delay block. See Delay for more information.

Introduced before R2006a

Variable Integer Delay (Obsolete)

Delay input by time-varying integer number of sample periods



Library

`dspobslib`

Description

Note: The Variable Integer Delay block has been replaced with the Simulink Variable Integer Delay block. Existing instances of the DSP block will continue to operate, but certain functionality will be disabled in future releases. See "Functionality being removed or replaced for blocks and System objects".

The Variable Integer Delay block delays the discrete-time input at the In port by the integer number of sample intervals specified by the input to the Delay port. The sample rate of the input signal at the Delay port must be the same as the sample rate of the input signal at the In port. When these sample rates are not the same, you need to insert a Zero-Order Hold or Rate Transition block in order to make the sample rates identical. When you set the **Input processing** parameter to `Elements as channels (sample based)`, the delay for an N-D input can be a scalar value to uniformly delay every sample in every channel, or a matrix containing one delay value for each channel of the input. When you set the **Input processing** parameter to `Columns as channels (frame based)`, the delay can be a scalar value to uniformly delay every sample in every channel, a vector containing one delay value for each sample in the input frame, or a vector containing one delay value for each channel in the input frame.

The delay values should be in the range of 0 to *D*, where *D* is the **Maximum delay**. Delay values greater than *D* or less than 0 are clipped to those respective values and noninteger delays are rounded to the nearest integer value.

The Variable Integer Delay block differs from the Delay block in the following ways.

Variable Integer Delay Block	Delay Block
The delay is provided as an input to the Delay port.	You specify the delay as a parameter setting in the dialog box.
Delay can vary with time; for example, when the block performs frame-based processing, the *n*th element's delay in the first input frame can differ from the *n*th element's delay in the second input frame.	Delay cannot vary with time; for example, when the block performs frame-based processing, the *n*th element's delay is the same for every input frame.
When you use the Variable Integer Delay block in a feedback loop, you must check the **Disable direct feedthrough by increasing minimum possible delay by one** check box. This prevents the occurrence of an algebraic loop when the delay of the Variable Integer Delay block is driven to zero.	You can use the Delay block to break an algebraic loop.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the Variable Integer Delay block supports N-D input arrays. When the input is an *M*-by-*N*-by-*P* array, the block treats each of the *M***N***P* elements as independent channels, and applies the delay at the Delay port to each channel.

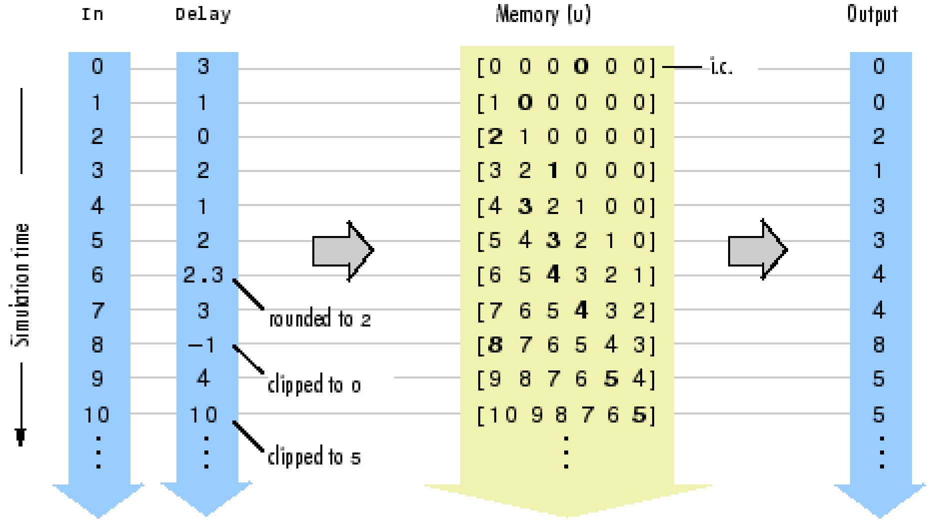
The Variable Integer Delay block stores the D+1 most recent samples received at the In port for each channel. At each sample time the block outputs the stored sample(s) indexed by the input to the Delay port.

For example, when the input to the In port, `u`, is a scalar signal, the block stores a vector, `U`, of the D+1 most recent signal samples. When the current input sample is `U(1)`, the previous input sample is `U(2)`, and so on, then the block's output is

```
y = U(v+1);     % Equivalent MATLAB code
```

where `v` is the input to the Delay port. A delay value of 0 (`v`=0) causes the block to pass through the sample at the In port in the same simulation step that it is received. The block's memory is initialized to the **Initial conditions** value at the start of the simulation (see below).

The next figure shows the block output for a scalar ramp sequence at the In port, a **Maximum delay** of 5, an **Initial conditions** of 0, and a variety of different delays at the Delay port.



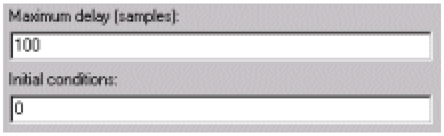
The current input at each time step is immediately stored in memory as `U(1)`. This allows the current input to be available at the output for a delay of 0 (v=0).

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation. Unlike the Delay block, the Variable Integer Delay block does not have a fixed initial delay period during which the initial conditions appear at the output. Instead, the initial conditions are propagated to the output only when they are indexed in memory by the value at the Delay port. Both fixed and time-varying initial conditions can be specified in a variety of ways to suit the dimensions of the input sequence.

Fixed Initial Conditions

The settings in this section specify fixed initial conditions. For a fixed initial condition, the block initializes each of D samples in memory to the value entered in the **Initial conditions** parameter. A fixed initial condition in sample-based mode can be specified in one of the following ways:

- Scalar value with which to initialize every sample of every channel in memory. For a general M-by-N input and the parameter settings in this figure,



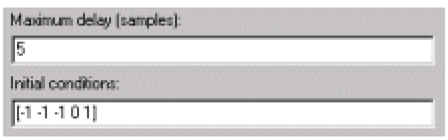
the block initializes 100 *M*-by-*N* matrices in memory with zeros.

- Array of size *M*-by-*N*-by-*D*. In this case, you can specify different fixed initial conditions for each channel. See the Array bullet in "Time-Varying Initial Conditions" on page 2-1852 below for details.

Time-Varying Initial Conditions

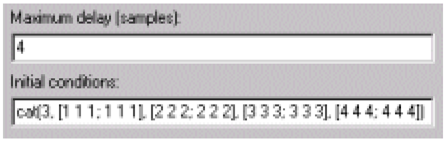
The following settings specify time-varying initial conditions. For a time-varying initial condition, the block initializes each of *D* samples in memory to one of the values entered in the **Initial conditions** parameter. This allows you to specify a unique output value for each sample in memory. A time-varying initial condition in sample-based mode can be specified in one of the following ways:

- Vector containing *D* elements with which to initialize memory samples `U(2:D+1)`, where *D* is the **Maximum delay**. For a scalar input and the parameters in the next figure, the block initializes `U(2:6)` with values `[-1, -1, -1, 0, 1]`.



- Array of dimension *M*-by-*N*-by-*D* with which to initialize memory samples `U(2:D+1)`, where *D* is the **Maximum delay** and *M* and *N* are the number of rows and columns, respectively, in the input matrix. For a 2-by-3 input and the following parameters, the block initializes memory locations `U(2:5)` with values

$$\mathbf{U}(2)=\begin{bmatrix}1&1&1\\1&1&1\end{bmatrix},\ \mathbf{U}(3)=\begin{bmatrix}2&2&2\\2&2&2\end{bmatrix},\ \mathbf{U}(4)=\begin{bmatrix}3&3&3\\3&3&3\end{bmatrix},\ \mathbf{U}(5)=\begin{bmatrix}4&4&4\\4&4&4\end{bmatrix}$$



An *M*-by-*N*-by-*P*-by-*D* array can be entered for the **Initial Conditions** parameter when the input is an *M*-by-*N*-by-*P* array. The (*M*,*N*,*P*,*T*)th sample of the **Initial Conditions** matrix provides the initial condition value for the (*M*,*N*,*P*)th channel of the input matrix at delay = *D*–*t*+1 samples.

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the input can be an *M*-by-*N* matrix. The block treats each of the *N* input columns as independent channels containing *M* sequential time samples.

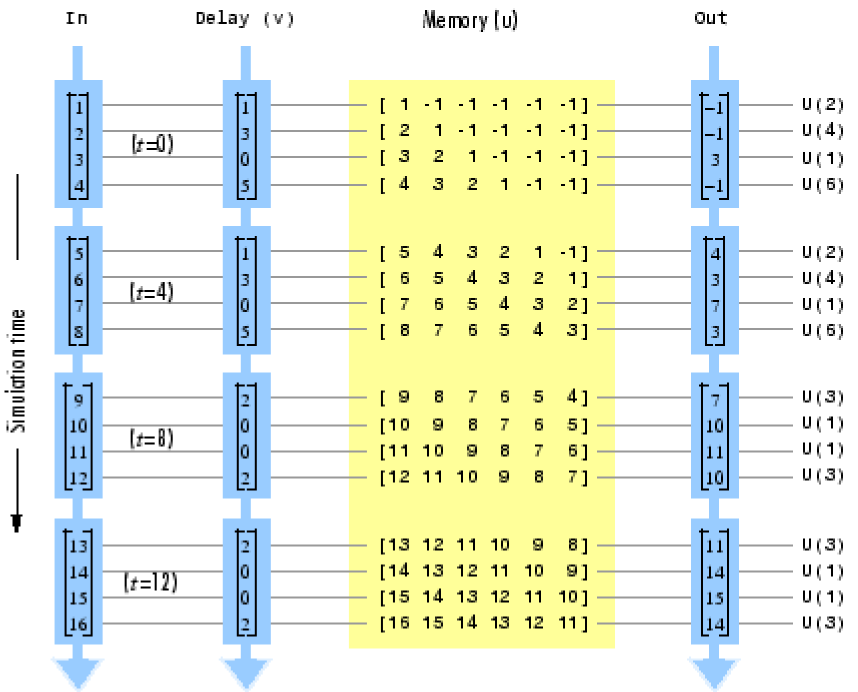
In this mode, the input at the Delay port can be a scalar value to uniformly delay every sample in every channel. It can also be a length-*M* column vector containing one delay value for each sample in the input frame(s). The set of delays contained in the vector is applied identically to every channel of a multichannel input. The Delay port entry can also be a length-*N* row vector, containing one delay for each channel. Finally, the Delay port entry can be an *M*-by-*N* matrix, containing a different delay for each corresponding element of the input.

Vector `v` does not specify when the samples in the current input frame will appear in the output. Rather, `v` indicates which previous input samples (stored in memory) should be included in the current output frame. The first sample in the current output frame is the input sample `v(1)` intervals earlier in the sequence, the second sample in the current output frame is the input sample `v(2)` intervals earlier in the sequence, and so on.

The illustration below shows how this works for an input with a sample period of 1 and frame size of 4. The **Maximum delay** (`Dmax`) is `5`, and the **Initial conditions** parameter is set to `-1`. The delay input changes from `[1 3 0 5]` to `[2 0 0 2]` after the second input frame. The samples in each output frame are the values in memory indexed by the elements of `v`:

```
y(1) = U(v(1)+1)
y(2) = U(v(2)+1)
y(3) = U(v(3)+1)
```

```
y(4) = U(v(4)+1)
```

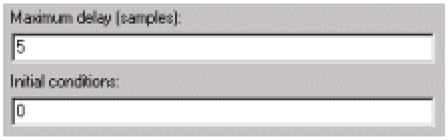


The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation. Both fixed and time-varying initial conditions can be specified.

Fixed Initial Conditions

The settings shown in this section specify fixed initial conditions. For a fixed initial condition, the block initializes each of D samples in memory to the value entered in the **Initial conditions** parameter. A fixed initial condition in frame-based mode can be one of the following:

- Scalar value with which to initialize every sample of every channel in memory. For a general M-by-N input with the parameter settings below, the block initializes five samples in memory with zeros.



- Array of size 1-by-N-by-D. In this case, you can specify different fixed initial conditions for each channel. See the Array bullet in "Time-Varying Initial Conditions" on page 2-1855 below for details.

Time-Varying Initial Conditions

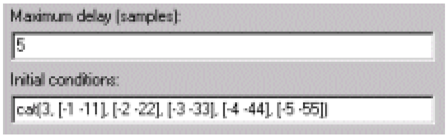
The following setting specifies a time-varying initial condition. For a time-varying initial condition, the block initializes each of D samples in memory to one of the values entered in the **Initial conditions** parameter. This allows you to specify a unique output value for each sample in memory. When the block is performing frame-based processing, you can specify a time-varying initial condition in the following ways:

- Vector containing D elements. In this case, all channels have the same set of time-varying initial conditions specified by the entries of the vector. For the ramp input `[1:100; 1:100]'` with a frame size of `4`, delay of `5`, and the following parameter settings, the block outputs the following sequence of frames at the start of the simulation:

$$\begin{bmatrix} -1 & -1 \\ -2 & -2 \\ -3 & -3 \\ -4 & -4 \end{bmatrix}, \begin{bmatrix} -5 & -5 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \ldots$$

- Array of size 1-by-N-by-D. In this case, you can specify different time-varying initial conditions for each channel. For the ramp input `[1:100; 1:100]'` with a frame size of `4`, delay of `5`, and the following parameter settings, the block outputs the following sequence of frames at the start of the simulation:

$$\begin{bmatrix} -1 & -11 \\ -2 & -22 \\ -3 & -33 \\ -4 & -44 \end{bmatrix}, \begin{bmatrix} -5 & -55 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \ldots$$



By specifying a 1-by-*N*-by-*D* initial condition array such that each 1-by-*N* vector entry is identical, you can implement different fixed initial conditions for each channel.

Examples

See “Basic Algorithmic Delay” in the *DSP System Toolbox User's Guide*.

Parameters

Maximum delay

The maximum delay that the block can produce for any sample. Delay input values exceeding this maximum are clipped at the maximum.

Initial conditions

The values with which the block's memory is initialized.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note: The `Inherited (this choice will be removed - see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Disable direct feedthrough by increasing minimum possible delay by one

Select this box to disable direct feedthrough by adding one to the minimum possible delay value. When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block increases the minimum possible delay value by *`frame size`* – 1. Similarly, when you set the **Input processing** parameter to `Elements as channels (sample based)`, the block increases the minimum possible delay value by one sample.

Checking this box allows you to use the Variable Integer Delay block in feedback loops.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Delay	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
	• 8-, 16-, and 32-bit unsigned integers

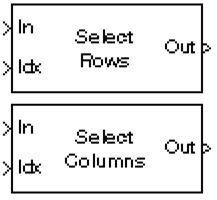
See Also

Delay	DSP System Toolbox
Variable Fractional Delay	DSP System Toolbox

Introduced in R2014b

Variable Selector

Select subset of rows or columns from input



Library

Signal Management / Indexing

`dspindex`

Description

The Variable Selector block extracts a subset of rows or columns from the M-by-N input matrix `u` at each input port. You specify the number of input and output ports in the **Number of input signals** parameter.

When the **Select** parameter is set to `Rows`, the Variable Selector block extracts rows from each input matrix, while if the **Select** parameter is set to `Columns`, the block extracts columns.

When the **Selector mode** parameter is set to `Variable`, the length-L vector input to the Idx port selects L rows or columns of each input to pass through to the output. The elements of the indexing vector can be updated at each sample time, but the vector length must remain the same throughout the simulation.

When the **Selector mode** parameter is set to `Fixed`, the Idx port is disabled, and the length-L vector specified in the **Elements** parameter selects L rows or columns of each input to pass through to the output. The **Elements** parameter is tunable, so you can change the values of the indexing vector elements at any time during the simulation; however, the vector length must remain the same.

For both variable and fixed indexing modes, the row selection operation is equivalent to

```
y = u(idx,:)      % Equivalent MATLAB code
```

and the column selection operation is equivalent to

```
y = u(:,idx)      % Equivalent MATLAB code
```

where `idx` is the length-L indexing vector. The row selection output size is L-by-N and the column selection output size is M-by-L. Input rows or columns can appear any number of times in the output, or not at all.

When the input is an unoriented vector, the **Select** parameter is ignored; the output is a unoriented vector of length L containing those elements specified by the length-L indexing vector.

When an element of the indexing vector references a nonexistent row or column of the input, the block reacts with the action you specify using the **Invalid index** parameter.

When the indexing vector elements are of Boolean data type, the block performs logical indexing. Select **Fill empty spaces in outputs (for logical indexing)** to access the **Fill values** parameter. These values are appended to the output to make it as long as the input elements.

Note The Variable Selector block always copies the selected input rows to a contiguous block of memory (unlike the Simulink Selector block).

Parameters

Number of input signals

Specify the number of input signals. An input port is created on the block for each input signal.

Select

Specify the dimension of the input to select, `Rows` or `Columns`.

Selector mode

Specify the type of indexing operation to perform, `Variable` or `Fixed`. Variable indexing uses the input at the Idx port to select rows or columns from the input at the `In` port. Fixed indexing uses the **Elements** parameter value to select rows from the input at the In port, and disables the Idx port.

Elements

Specify a vector containing the indices of the input rows or columns that will appear in the output matrix. This parameter appears only when you set the **Selector mode** to `Fixed`.

Index mode

When set to `One-based`, an index value of `1` refers to the first row or column of the input. When set to `Zero-based`, an index value of `0` refers to the first row or column of the input.

Invalid index

Specify how the block handles an invalid index value. You can select one of the following options:

- `Clip index` — Clip the index to the nearest valid value, and do not issue an alert.

  For example, if the block receives a `64`-by-`4` input and the **Select** parameter is set to `Rows`, the block clips an index of `72` to `64`. For the same input, if the **Select** parameter is set to `Columns`, the block clips an index of `72` to `4`. In both cases, the block clips an index of `-2` to `1`.
- `Clip and warn` — Clip the index to the nearest valid value and display a warning message at the MATLAB command line.
- `Generate error` — Display an error dialog box and terminate the simulation.

This parameter is tunable (Simulink).

Fill empty spaces in outputs (for logical indexing)

When the indexing vector elements are of Boolean data type, the block performs logical indexing. This can cause empty spaces in the output. Select this parameter to designate values to be appended to the output in the **Fill values** parameter.

Fill values

Specify the fill values when the block performs logical indexing. This parameter appears only when you select the **Fill empty spaces in outputs (for logical indexing)** check box.

HDL Code Generation

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

For more information on implementations, properties, and restrictions for HDL code generation, see Variable Selector.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated
Idx	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated
Out	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated

See Also

`Multiport Selector`	DSP System Toolbox

`Permute Matrix`	DSP System Toolbox
Selector	Simulink
`Submatrix`	DSP System Toolbox

Introduced before R2006a

Variance

Variance of input or sequence of inputs
Library: DSP System Toolbox / Statistics

Description

The Variance block computes the unbiased variance of each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the variance of the entire input. You can specify the dimension using the **Find the variance value over** parameter. The Variance block can also track the variance in a sequence of inputs over a period of time. To track the variance in a sequence of inputs, select the **Running variance** parameter.

Note: The **Running** mode in the Variance block will be removed in a future release. To compute the running variance in Simulink, use the `Moving Variance` block instead.

Ports

Input

`In` — Data input

vector | matrix | *N*-D array

The block accepts real-valued or complex-valued multichannel and multidimensional inputs.

This port is unnamed until you select the **Running variance** parameter and set the **Reset port** parameter to any option other than `None`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed_point`
Complex Number Support: Yes

`Rst` — Reset port

scalar

Specify the reset event that causes the block to reset the running variance. The sample time of the **Rst** input must be a positive integer multiple of the input sample time.

Dependencies

To enable this port, select the **Running variance** parameter and set the **Reset port** parameter to any option other than `None`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Port_1` — Variance along the specified dimension

scalar | vector | matrix | *N*-D array

The data type of the output matches the data type of the input.

When you do not select the **Running variance** parameter, the block computes the variance in each row or column of the input, or along vectors of a specified dimension of the input. It can also compute the variance of the entire input at each individual sample time. Each element in the output array `y` is the variance of the corresponding column, row, or entire input. The output array `y` depends on the setting of the **Find the variance value over** parameter. Consider a three-dimensional input signal of size *M*-by-*N*-by-*P*. When you set **Find the variance value over** to:

- `Entire input` — The output at each sample time is a scalar that contains the variance of the *M*-by-*N*-by-*P* input matrix.
- `Each row` — The output at each sample time consists of an *M*-by-1-by-*P* array, where each element contains the variance of each vector over the second dimension of the input. For an *M*-by-*N* matrix input, the output at each sample time is an *M*-by-1 column vector.
- `Each column` — The output at each sample time consists of a 1-by-*N*-by-*P* array, where each element contains the variance of each vector over the first dimension of the input. For an *M*-by-*N* matrix input, the output at each sample time is a 1-by-*N* row vector.

In this mode, the block treats length-M unoriented vector inputs as M-by-1 column vectors.

- `Specified dimension` — The output at each sample time depends on the value of the **Dimension** parameter. If you set the **Dimension** to `1`, the output is the same as when you select `Each column`. If you set the **Dimension** to `2`, the output is the same as when you select `Each row`. If you set the **Dimension** to `3`, the output at each sample time is an M-by-N matrix containing the variance of each vector over the third dimension of the input.

When you select **Running variance**, the block tracks the variance of each channel in a time sequence of inputs. In this mode, you must also specify a value for the **Input processing** parameter.

- `Elements as channels (sample based)` — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size M-by-N-by-P, the block outputs an M-by-N-by-P array. Each element y_{ijk} of the output contains the variance of the element u_{ijk} for all inputs since the last reset.

  When a reset event occurs, the running variance y_{ijk} in the current frame is reset to the element u_{ijk}.
- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel. This option does not support input signals with more than two dimensions. For a two-dimensional input signal of size M-by-N, the block outputs an M-by-N matrix. Each element y_{ij} of the output contains the variance of the elements in the jth column of all inputs since the last reset, up to and including the element u_{ij} of the current input.

  When a reset event occurs, the running variance for each channel becomes the variance of all the samples in the current input frame, up to and including the current input sample.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed_point`

Parameters

Main Tab

`Running variance` — Option to select running variance

off (default) | on

When you select the **Running variance** parameter, the block tracks the variance value of each channel in a time sequence of inputs.

`Find the variance value over` — Dimension over which the block computes the variance

`Each column` (default) | `Entire input` | `Each row` | `Specified dimension`

- `Each column` — The block outputs the variance over each column.
- `Each row` — The block outputs the variance over each row.
- `Entire input` — The block outputs the variance over the entire input.
- `Specified dimension` — The block outputs the variance over the dimension specified in the **Dimension** parameter.

Dependencies

To enable this parameter, clear the **Running variance** parameter.

`Dimension` — Custom dimension

`1` (default) | scalar

Specify the dimension (one-based value) of the input signal over which the variance is computed. The value of this parameter must be greater than 0 and less than the number of dimensions in the input signal.

Dependencies

To enable this parameter, set **Find the variance value over** to `Specified dimension`.

`Input processing` — Method to process the input in running mode
`Columns as channels (frame based)` (default) | `Elements as channels (sample based)`

- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel. This option does not support input signals with more than two dimensions. For a two-dimensional input signal of size *M*-by-*N*, the block outputs an *M*-by-*N* matrix. Each element y_{ij} of the output contains the variance of the elements in the *j*th column of all inputs since the last reset, up to and including the element u_{ij} of the current input.

  When a reset event occurs, the running variance for each channel becomes the variance of all the samples in the current input frame, up to and including the current input sample.
- `Elements as channels (sample based)` — The block treats each element of the input as a separate channel. For a three-dimensional input signal of size *M*-by-*N*-by-*P*, the block outputs an *M*-by-*N*-by-*P* array. Each element y_{ijk} of the output contains the variance of the element u_{ijk} for all inputs since the last reset.

  When a reset event occurs, the running variance y_{ijk} in the current frame is reset to the element u_{ijk}.

  Variable-Size Inputs

  When your inputs are of variable size, and you select the **Running variance** parameter, then:

  - If you set the **Input processing** parameter to `Elements as channels (sample based)`, the state is reset.
  - If you set the **Input processing** parameter to `Columns as channels (frame based)`, then:

    - When the input size difference is in the number of channels (number of columns), the state is reset.
    - When the input size difference is in the length of channels (number of rows), no reset occurs and the running operation is carried out as usual.

Dependencies

To enable this parameter, select the **Running variance** parameter.

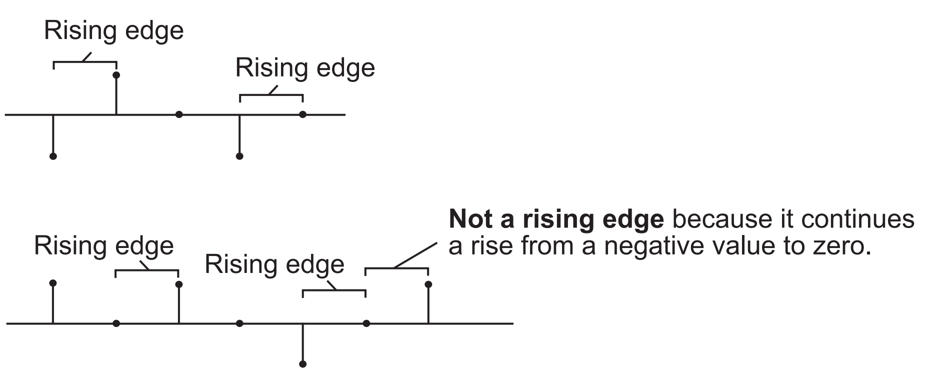
`Reset port` — Reset event
`None` (default) | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero sample`

The block resets the running variance whenever a reset event is detected at the optional **Rst** port. The reset sample time must be a positive integer multiple of the input sample time.

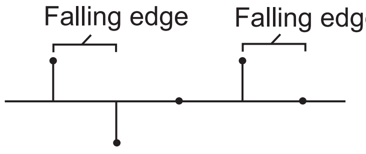
When a reset event occurs while the **Input processing** parameter is set to `Elements as channels (sample based)`, the running variance for each channel is initialized to the value in the corresponding channel of the current input. Similarly, when the **Input processing** parameter is set to `Columns as channels (frame based)`, the running variance for each channel becomes the variance of all the samples in the current input frame, up to and including the current input sample.

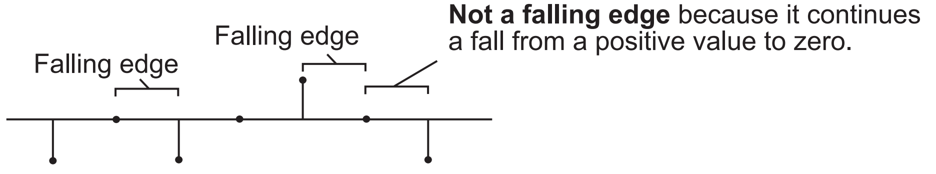
Use this parameter to specify the reset event.

- `None` — Disables the **Rst** port.
- `Rising edge` — Triggers a reset operation when the **Rst** input does one of the following:
  - Rises from a negative value to either a positive value or zero.
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.



- `Falling edge` — Triggers a reset operation when the **Rst** input does one of the following:
  - Falls from a positive value to a negative value or zero.
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.





- `Either edge` — Triggers a reset operation when the **Rst** input is a `Rising edge` or `Falling edge`.
- `Non-zero sample` — Triggers a reset operation at each sample time, when the **Rst** input is not zero.

Note: When running simulations in the Simulink multitasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

Dependencies

To enable this parameter, select the **Running variance** parameter.

Data Types Tab

Note: To use these parameters, the data input must be fixed point. For all other inputs, the parameters on the **Data Types** tab are ignored.

`Rounding mode` — Method of rounding operation

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Simplest` | `Zero`

Select the rounding mode for fixed-point operations.

`Overflow mode` — Method of overflow action

`Wrap` (default) | `Saturate`

Select the overflow mode for fixed-point operations.

`Input-squared product` — Data type that stores the input-squared term

`Same as input` (default) | `Binary point scaling` | `Slope and bias scaling`

The squares of the input elements are stored in the **Input-squared product** data type. If the input is complex, the squares of the real and imaginary parts of the input are stored in this data type. For more details, see “Fixed Point” on page 2- .

You can set this parameter to:

- `Inherit: Same as input` — The block specifies this data type to be the same as the input data type.
- `Binary point scaling` — The **Input-squared product** data type uses binary point scaling. If you select this option, the block displays the fields to specify the **Word length** and **Fraction length**. The **Signedness** is inherited from the input.
- `Slope and bias scaling` — The **Input-squared product** data type uses slope and bias scaling. If you select this option, the block displays the fields to specify the **Word length** and **Slope**. The **Signedness** is inherited from the input and **Bias** is specified to be `0`.

`Input-sum-squared product` — Data type that stores the input-sum-squared term

`Same as input-squared product` (default) | `Binary point scaling` | `Slope and bias scaling`

The squares of the sum of the input elements are stored in the **Input-sum-squared product** data type. If the input is complex, the squares of the sum of the real parts and the squares of the sum of the imaginary parts are stored in this data type. For more details, see “Fixed Point” on page 2- .

You can set this parameter to:

- `Same as input-squared product` — The block specifies this data type to be the same as the input squared-product data type.
- `Binary point scaling` — The **Input-sum-squared product** data type uses binary point scaling. If you select this option, the block displays the fields to specify the **Word length** and **Fraction length**. The **Signedness** is inherited from the input.
- `Slope and bias scaling` — The **Input-sum-squared product** data type uses slope and bias scaling. If you select this option, the block displays the fields to specify the **Word length** and **Slope**. The **Signedness** is inherited from the input and **Bias** is specified to be `0`.

`Accumulator` — Accumulator data type

`Same as input-squared product` (default) | `Same as input` | `Binary point scaling` | `Slope and bias scaling`

Accumulator specifies the data type of the output of an accumulation operation in the Variance block. See “Fixed Point” on page 2- for illustrations depicting the use of the accumulator data type in this block.

You can set this parameter to:

- `Same as input-squared product` — The block specifies the accumulator data type to be the same as the input-squared product data type.
- `Same as input` — The block specifies the accumulator data type to be the same as the input data type.
- `Binary point scaling` — The **Accumulator** data type uses binary point scaling. If you select this option, the block displays the fields to specify the **Word length** and **Fraction length**. The **Signedness** is inherited from the input.
- `Slope and bias scaling` — The **Accumulator** data type uses slope and bias scaling. If you select this option, the block displays the fields to specify the **Word length** and **Slope**. The **Signedness** is inherited from the input and **Bias** is specified to be `0`.

`Output` — Output data type

`Same as input-squared product` (default) | `Same as accumulator` | `Same as input` | `Binary point scaling` | `Slope and bias scaling`

Output specifies the data type of the output of the Variance block. See "Fixed Point" on page 2- for illustrations depicting the use of the output data type in this block. You can set it to:

- `Same as input-squared product` — The block specifies the output data type to be the same as the input-squared product data type.
- `Same as accumulator` — The block specifies the output data type to be the same as the accumulator data type.
- `Same as input` — The block specifies the output data type to be the same as the input data type.
- `Binary point scaling` — The **Output** data type uses binary point scaling. If you select this option, the block displays the fields to specify the **Word length** and **Fraction length**. The **Signedness** is inherited from the input.
- `Slope and bias scaling` — The **Output** data type uses slope and bias scaling. If you select this option, the block displays the fields to specify the **Word length** and **Slope**. The **Signedness** is inherited from the input and **Bias** is specified to be `0`.

`Lock data type settings against changes by the fixed-point tools` — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block.

Definitions

Variance

The variance of a discrete-time signal is the square of the standard deviation of the signal.

Variance gives a measure of deviation of the signal from its mean value.

For purely real or imaginary input, u, of size M-by-N, the variance is given by the following equation:

$$y = \sigma^2 = \frac{\sum_{i=1}^{M}\sum_{j=1}^{N}\left|u_{ij}\right|^2 - \frac{\left|\sum_{i=1}^{M}\sum_{j=1}^{N}u_{ij}\right|^2}{M*N}}{M*N-1}$$

- u_{ij} is the input data element at indices i, j.
- M is the length of the jth column.
- N is the number of columns.

For complex inputs, the variance is given by the following equation:

$$\sigma^2 = \sigma_{\mathrm{Re}}{}^2 + \sigma_{\mathrm{Im}}{}^2$$

- $\sigma_{Re}{}^2$ is the variance of the real part of the complex input.
- $\sigma_{Im}{}^2$ is the variance of the imaginary part of the complex input.

Algorithms

Variance

When you clear the **Running variance** parameter in the block and specify a dimension, the block produces results identical to the MATLAB `var` function, when it is called as `y = var(u,0,D)`.

- `u` is the data input.
- `D` is the dimension.
- `y` is the variance along the specified dimension.

The variance along the entire input is identical to calling the `var` function as `y = var(u(:))`.

For a complex input signal, the variance is the sum of the variances of the real and imaginary parts.

$$\sigma^2 = \sigma_{\mathrm{Re}}{}^2 + \sigma_{\mathrm{Im}}{}^2$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

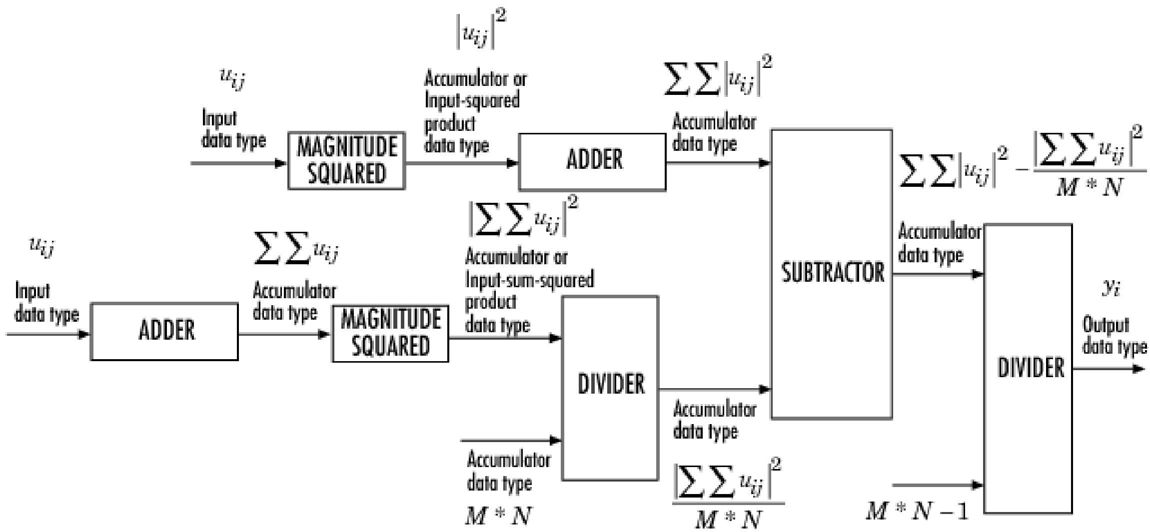
Fixed-Point Conversion

Design and simulate fixed-point algorithms using Fixed-Point Designer™.

For purely real or imaginary input, *u* of size *M*-by-*N*, the variance is given by the following equation.

$$y = \sigma^2 = \frac{\sum_{i=1}^{M}\sum_{j=1}^{N}\left|u_{ij}\right|^2 - \frac{\left|\sum_{i=1}^{M}\sum_{j=1}^{N}u_{ij}\right|^2}{M*N}}{M*N-1}$$

The following diagram shows the data types used within the Variance block when the input is fixed-point.



For complex inputs, the variance is given by the following equation:

$$\sigma^2 = \sigma_{\mathrm{Re}}{}^2 + \sigma_{\mathrm{Im}}{}^2$$

See Also

See Also

Functions
`var`

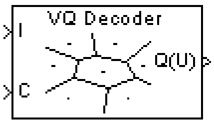
System Objects
dsp.Variance | dsp.MovingVariance

Blocks
`Moving Variance`

Introduced before R2006a

Vector Quantizer Decoder

Find vector quantizer codeword that corresponds to given, zero-based index value



Vector Quantizer
Decoder

Library

Quantizers

`dspquant2`

Description

The Vector Quantizer Decoder block associates each input index value with a codeword, a column vector of quantized output values defined in the **Codebook values** parameter. When you input multiple index values into this block, the block outputs a matrix of quantized output vectors. This matrix is created by horizontally concatenating the codeword vectors that correspond to each index value.

You can select how you want to enter the codebook values using the **Source of codebook** parameter. When you select `Specify via dialog`, you can type the codebook values into the block parameters dialog box. Select `Input port` and port C appears on the block. The block uses the input to port C as the **Codebook values** parameter.

The **Codebook values** parameter is a k-by-N matrix of values, where $k \geq 1$ and $N \geq 1$. Each column of this matrix is a codeword vector, and each codeword vector corresponds to an index value. The index values are zero based; therefore, the first codeword vector corresponds to an index value of 0, the second codeword vector corresponds to an index value of 1, and so on.

The input to this block is a vector of index values, where $0 \leq index < N$ and N is the number of columns of the codebook matrix. Use the **Action for out of range index**

value parameter to determine how the block behaves when an input index value is out of this range. When you want any index values less than 0 to be set to 0 and any index values greater than or equal to *N* to be set to *N*-1, select `Clip`. When you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to *N* are set to *N*-1, select `Clip and warn`. When you want the simulation to stop and display an error when the index values are out of range, select `Error`.

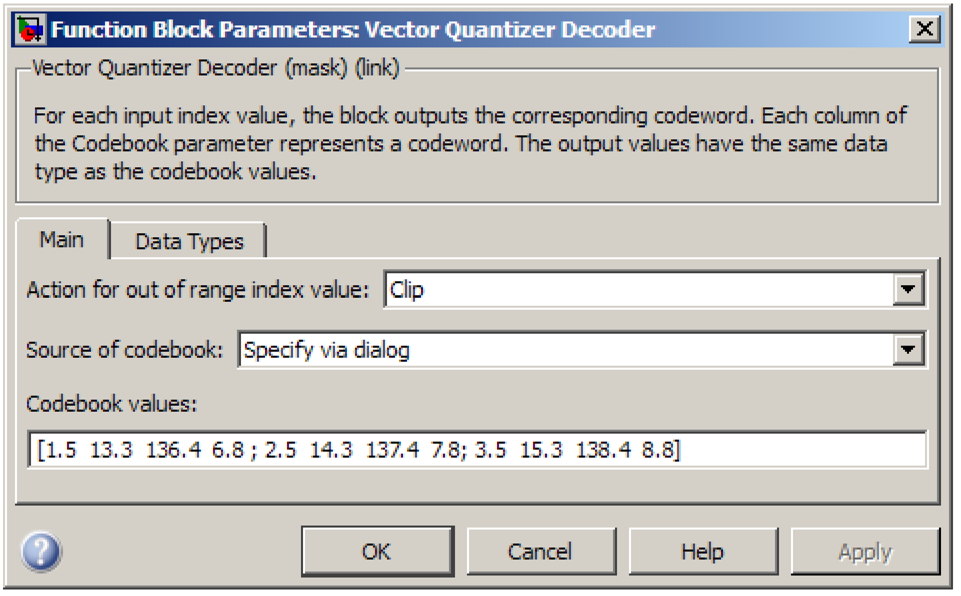
Data Type Support

The input to the block can be the index values and the codebook values. The data type of the index input to the block at port I can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. The data type of the codebook values can be `double`, `single`, or Fixed-point.

The output of the block is the quantized output values. These quantized output values always have the same data type as the codebook values. When the codebook values are specified via an input port, the block assigns the same data type to the Q(U) output port. When the codebook values are specified via the dialog, use the **Codebook and output data type** parameter to specify the data type of the Q(U) output port. The data type of the codebook and quantized output can be `Same as input`, `double`, `single`, `Fixed-point`, `User-defined`, or `Inherit via back propagation`.

Dialog Box

The **Main** pane of the Vector Quantizer Decoder block dialog appears as follows.



Action for out of range index value

Choose the behavior of the block when an input index value is out of range, where $0 \leq index < N$ and N is the length of the codebook vector. Select `Clip` when you want any index values less than 0 to be set to 0 and any index values greater than or equal to N to be set to N-1. Select `Clip and warn` when you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to N are set to N-1. Select `Error` when you want the simulation to stop and display an error when the index values are out of range.

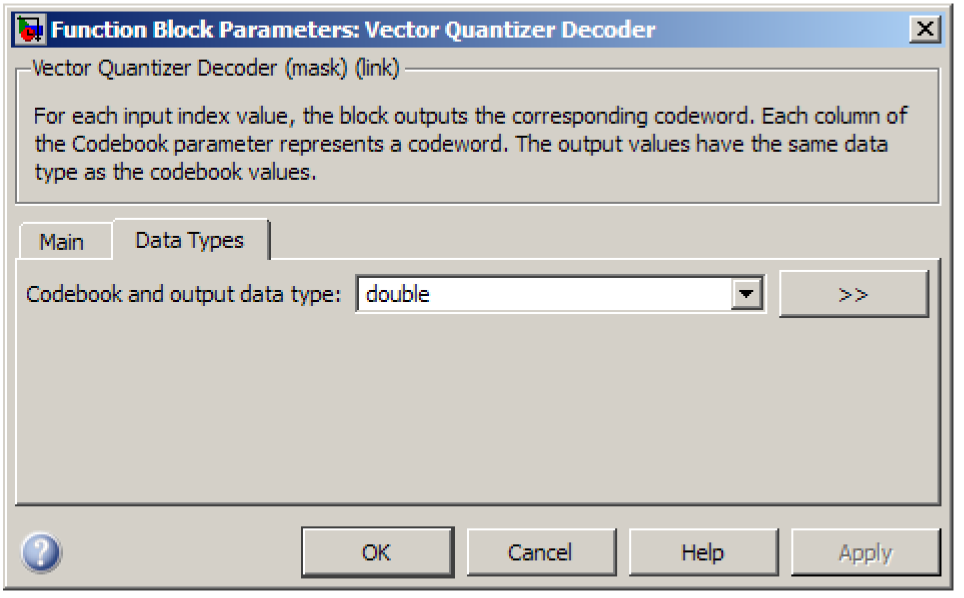
Source of codebook

Choose `Specify via dialog` to type the codebook values into the block parameters dialog box. Select `Input port` to specify the codebook values using the block's input port, C.

Codebook values

Enter a k-by-N matrix of quantized output values, where $1 \leq k$ and $1 \leq N$. Each column of your matrix corresponds to an index value. This parameter is visible if, from the **Source of codebook** list, you select `Specify via dialog`.

The **Data Types** pane of the Vector Quantizer Decoder block dialog appears as follows.



Codebook and output data type

Specify the data type of the codebook and quantized output values. You can select one of the following:

- A rule that inherits a data type, for example, `Inherit: Same as input`.
- A built in data type, such as `double`
- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

Click the **Show data type assistant** button 

to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” (Simulink) in Simulink User's Guide (Simulink) for more information.

This parameter is available only when you set the **Source of codebook** parameter to `Specify via dialog`. If you set the **Source of codebook** parameter to `Input port`, the output values have the same data type as the input codebook values.

References

Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

Port	Supported Data Types
I	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
C	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers
Q(U)	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

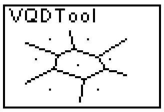
See Also

Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Scalar Quantizer Design	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
Uniform Decoder	DSP System Toolbox
Vector Quantizer Encoder	DSP System Toolbox

Introduced before R2006a

Vector Quantizer Design

Design vector quantizer using Vector Quantizer Design Tool (VQDTool)



Vector Quantizer
Design

Library

Quantizers

`dspquant2`

Description

Double-click on the Vector Quantizer Design block to start VQDTool, a GUI that allows you to design and implement a vector quantizer. You can also start VQDTool by typing `vqdtool` at the MATLAB command prompt. Based on your specifications, VQDTool iteratively calculates the codebook values that minimize the mean squared error between the training set and the codebook until the stopping criteria for the design process is satisfied. The block uses the resulting codebook values to implement your vector quantizer.

For the **Training Set** parameter, enter a k-by-M matrix of values you want to use to train the quantizer codebook. The variable k, where $k \geq 1$, is the length of each training vector. It also represents the dimension of your quantizer. The variable M, where $M \geq 2$, is the number of training vectors. This data can be created using a MATLAB function, such as the default value `randn(10,1000)`, or it can be any variable defined in the MATLAB workspace.

You have two choices for the **Source of initial codebook** parameter. Select `Auto-generate` to have the block choose the values of the initial codebook. In this case, the block picks N random training vectors as the initial codebook, where N is the **Number of levels** parameter and $N \geq 2$. When you select `User defined`, enter the initial codebook values in the **Initial codebook** field. The initial codebook matrix must have the same number of rows as the training set. Each column of the codebook is a codeword, and your codebook must have at least two codewords.

For the given training set and initial codebook, the block performs an iterative process, using the Generalized Lloyd Algorithm (GLA), to design a final codebook. For each iteration of the GLA, the block first associates each training vector with its nearest codeword by calculating the distortion. You can specify one of the two possible methods for calculating distortion using the **Distortion measure** parameter.

When you select `Squared error` for the **Distortion measure** parameter, the block finds the nearest codeword by calculating the squared error (unweighted). Consider the codebook $CB = [CW_1 \quad CW_2 \quad ... \quad CW_N]$. This codebook has N codewords; each codeword has k elements. The i-th codeword is defined as $CW_i = [a_{1i} \quad a_{2i} \quad \cdots \quad a_{ki}]$. The training set has M columns and is defined as $U = [U_1 \quad U_2 \quad ... \quad U_M]$, where the p-th training vector is $U_p = [u_{1p} \quad u_{2p} \quad \cdots \quad u_{kp}]'$. The squared error (unweighted) is calculated using the equation

$$D = \sum_{j=1}^{k} \left(a_{ji} - u_{jp}\right)^2$$

When you select `Weighted squared error` for the **Distortion measure** parameter, enter a vector or matrix for the **Weighting factor** parameter. When the weighting factor is a vector, its length must be equal to the number of rows in the training set. This weighting factor is used for each training vector. When the weighting factor is a matrix, it must be the same size as the training set matrix. The block finds the nearest codeword by calculating the weighted squared error. If the weighting factor for the p-th column of the training vector, U_p, is defined as $Wp = [w_{1p} \quad w_{2p} \quad \cdots \quad w_{kp}]'$, then the weighted squared error is defined by the equation

$$D = \sum_{j=1}^{k} w_{jp} \left(a_{ji} - u_{jp}\right)^2$$

Once the block has associated all the training vectors with their nearest codeword vectors, the block calculates the mean squared error for the codebook and checks to see if the stopping criteria for the process has been satisfied.

The two possible options for the **Stopping criteria** parameter are `Relative threshold` and `Maximum iteration`. When you want the design process to stop when the fractional drop in the squared error is below a certain value, select `Relative threshold`. Then, type the maximum acceptable fractional drop in the **Relative threshold** field. The fraction drop in the squared error is defined as

$$\frac{\text{error at previous iteration} - \text{error at current iteration}}{\text{error at previous iteration}}$$

When you want the design process to stop after a certain number of iterations, choose `Maximum iteration`. Then, enter the maximum number of iterations you want the block to perform in the **Maximum iteration** field. For **Stopping criteria**, you can also choose `Whichever comes first` and enter **Relative threshold** and **Maximum iteration** values. The block stops iterating as soon as one of these conditions is satisfied.

When a training vector has the same distortion for two different codeword vectors, the algorithm uses the **Tie-breaking rule** parameter to determine which codeword vector the training vector is associated with. When you want the training vector to be associated with the lower indexed codeword, select `Lower indexed codeword`. To associate the training vector with the higher indexed codeword, select `Higher indexed codeword`.

With each iteration, the block updates the codeword values in order to minimize the distortion. The **Codebook update method** parameter defines the way the block calculates these new codebook values.

Note If, for the **Distortion measure** parameter, you choose `Squared error`, the **Codebook update method** parameter is set to `Mean`.

If, for the **Distortion measure** parameter, you choose `Weighted squared error` and you choose `Mean` for the **Codebook update method** parameter, the new codeword vector is found as follows. Suppose there are three training vectors associated with one codeword vector. The training vectors are

$$TS_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix},\ TS_3 = \begin{bmatrix} 10 \\ 12 \end{bmatrix},\ \text{and } TS_7 = \begin{bmatrix} 11 \\ 12 \end{bmatrix}.$$

The new codeword vector is calculated as

$$CW_{new} = \begin{bmatrix} \frac{1+10+11}{3} \\ \frac{2+12+12}{3} \end{bmatrix}$$

where the denominator is the number of training vectors associated with this codeword. If, for the **Codebook update method** parameter, you choose `Centroid` and you specify the weighting factors $W_1 = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$, $W_3 = \begin{bmatrix} 1 \\ 0.6 \end{bmatrix}$, and $W_7 = \begin{bmatrix} 0.3 \\ 0.4 \end{bmatrix}$, the new codeword vector is calculated as

$$CW_{new} = \begin{bmatrix} \frac{(0.1)(1)+(1)(10)+(0.3)(11)}{0.1+1+0.3} \\ \frac{(0.2)(2)+(0.6)(12)+(0.4)(12)}{0.2+0.6+0.4} \end{bmatrix}$$

Click **Design and Plot** to design the quantizer with the parameter values specified on the left side of the GUI. The performance curve and the entropy of the quantizer are updated and displayed in the figures on the right side of the GUI.

Note You must click **Design and Plot** to apply any changes you make to the parameter values in the VQDTool dialog box.

The following is an example of how the block calculates the entropy of the quantizer at each iteration. Suppose you have a codebook with four codewords and a training set with 200 training vectors. Also suppose that, at the i-th iteration, 40 training vectors are associated with the first codeword, 60 training vectors are associated with the second codeword, 20 training vectors are associated with the third codeword, and 80 training vectors are associated with the fourth codeword. The probability that a training vector is associated with the first codeword is $\frac{40}{200}$. The probabilities that training vectors are associated with the second, third, and fourth codewords are $\frac{60}{200}$, $\frac{20}{200}$, and $\frac{80}{200}$, respectively. The GUI uses these probabilities to calculate the entropy according to the equation

$$H = \sum_{i=1}^{N} -p_i \log_2 p_i$$

where N is the number of codewords. Based on these probabilities, the GUI calculates the entropy of the quantizer at the i-th iteration as

$$H = -\left(\frac{40}{200}\log_2 \frac{40}{200} + \frac{60}{200}\log_2 \frac{60}{200} + \frac{20}{200}\log_2 \frac{20}{200} + \frac{80}{200}\log_2 \frac{80}{200} \right)$$
$$H = 1.8464$$

VQDTool can export parameter values that correspond to the figures displayed in the GUI. Click the **Export Outputs** button, or press **Ctrl+E**, to export the **Final Codebook**, **Mean Square Error**, and **Entropy** values to the workspace, a text file, or a MAT-file.

In the **Model** section of the GUI, specify the destination of the block that will contain the parameters of your quantizer. For **Destination**, select `Current model` to create a block with your parameters in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Select `New model` to create a block in a new model file.

From the **Block type** list, select `Encoder` to design a Vector Quantizer Encoder block. Select `Decoder` to design a Vector Quantizer Decoder block. Select `Both` to design a Vector Quantizer Encoder block and a Vector Quantizer Decoder block.

In the **Encoder block name** field, enter a name for the Vector Quantizer Encoder block. In the **Decoder block name** field, enter a name for the Vector Quantizer Decoder block. When you have a Vector Quantizer Encoder and/or Decoder block in your destination model with the same name, select the **Overwrite target block** check box to replace the block's parameters with the current parameters. When you do not select this check box, a new Vector Quantizer Encoder and/or Decoder block is created in your destination model.

Click **Generate Model**. VQDTool uses the parameters that correspond to the current plots to set the parameters of the Vector Quantizer Encoder and/or Decoder blocks.

Parameters

Training Set

Enter the samples of the signal you would like to quantize. This data set can be a MATLAB function or a variable defined in the MATLAB workspace. The typical length of this data vector is `1e5`.

Source of initial codebook

Select `Auto-generate` to have the block choose the initial codebook values. Choose `User defined` to enter your own initial codebook values.

Number of levels

Enter the number of codeword vectors, N, in your codebook matrix, where $N \geq 2$.

Initial codebook

Enter your initial codebook values. From the **Source of initial codebook** list, select `User defined` in order to activate this parameter. The codebook must have the same number of rows as the training set. You must provide at least two codeword vectors.

Distortion measure

When you select `Squared error`, the block finds the nearest codeword by calculating the squared error (unweighted). When you select `Weighted squared error`, the block finds the nearest codeword by calculating the weighted squared error.

Weighting factor

Enter a vector or matrix. The block uses these values to compute the weighted squared error. When the weighting factor is a vector, its length must be equal to the number of rows in the training set. This weighting factor is used for each training vector. When the weighting factor is a matrix, it must be the same size as the training set matrix. The individual weighting factors cannot be negative. The weighting factor vector or matrix cannot contain all zeros.

Stopping criteria

Choose `Relative threshold` to enter the maximum acceptable fractional drop in the squared quantization error. Choose `Maximum iteration` to specify the number of iterations at which to stop. Choose `Whichever comes first` and the block stops the iteration process as soon as the relative threshold or maximum iteration value is attained.

Relative threshold

This parameter is available when you choose `Relative threshold` or `Whichever comes first` for the **Stopping criteria** parameter. Enter the value that is the maximum acceptable fractional drop in the squared quantization error.

Maximum iteration

This parameter is available when you choose `Maximum iteration` or `Whichever comes first` for the **Stopping criteria** parameter. Enter the maximum number of iterations you want the block to perform.

Tie-breaking rules

When a training vector has the same distortion for two different codeword vectors, select `Lower indexed codeword` to associate the training vector with the lower indexed codeword. Select `Higher indexed codeword` to associate the training vector with the lower indexed codeword.

Codebook update method

When you choose `Mean`, the new codeword vector is calculated by taking the average of all the training vector values that were associated with the original codeword vector. When you choose `Centroid`, the block calculates the new codeword vector by taking the weighted average of all the training vector values that were associated with the original codeword vector Note that if, for the **Distortion measure** parameter, you choose `Squared error`, the **Codebook update method** parameter is set to `Mean`.

Destination

Choose `Current model` to create a Vector Quantizer block in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Choose `New model` to create a block in a new model file.

Block type

Select `Encoder` to design a Vector Quantizer Encoder block. Select `Decoder` to design a Vector Quantizer Decoder block. Select `Both` to design a Vector Quantizer Encoder block and a Vector Quantizer Decoder block.

Encoder block name

Enter a name for the Vector Quantizer Encoder block.

Decoder block name

Enter a name for the Vector Quantizer Decoder block.

Overwrite target block

When you do not select this check box and a Vector Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, a new Vector Quantizer Encoder and/or Decoder block is created in the destination model. When you select this check box and a Vector Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, the parameters of these blocks are overwritten by new parameters.

Generate Model

Click this button and VQDTool uses the parameters that correspond to the current plots to set the parameters of the Vector Quantizer Encoder and/or Decoder blocks.

Design and Plot

Click this button to design a quantizer using the parameters on the left side of the GUI and to update the performance curve and entropy plots on the right side of the GUI.

You must click **Design and Plot** to apply any changes you make to the parameter values in the VQDTool GUI.

Export Outputs

Click this button, or press **Ctrl+E**, to export the **Final Codebook**, **Mean Squared Error**, and **Entropy** values to the workspace, a text file, or a MAT-file.

Supported Data Types

- Double-precision floating point

References

[1] Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

See Also

See Also

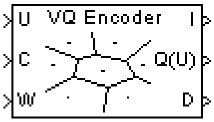
Blocks

Quantizer | `Scalar Quantizer Decoder` | `Scalar Quantizer Design` | `Uniform Decoder` | `Uniform Encoder` | `Vector Quantizer Decoder` | `Vector Quantizer Encoder`

Introduced before R2006a

Vector Quantizer Encoder

For given input, find index of nearest codeword based on Euclidean or weighted Euclidean distance measure



Vector Quantizer Encoder

Library

Quantizers

`dspquant2`

Description

The Vector Quantizer Encoder block compares each input column vector to the codeword vectors in the codebook matrix. Each column of this codebook matrix is a codeword. The block finds the codeword vector nearest to the input column vector and returns its zero-based index. This block supports real floating-point and fixed-point signals on all input ports.

The block finds the nearest codeword by calculating the distortion. The block uses two methods for calculating distortion: Euclidean squared error (unweighted) and weighted Euclidean squared error. Consider the codebook, $CB = [CW_1 \quad CW_2 \quad \ldots \quad CW_N]$. This codebook has N codewords; each codeword has k elements. The i-th codeword is defined as a column vector, $CW_i = [a_{1i} \quad a_{2i} \quad \ldots \quad a_{ki}]$. The multichannel input has M columns and is defined as $U = [U_1 \quad U_2 \quad \ldots \quad U_M]$, where the p-th input column vector is $U_p = [u_{1p} \quad u_{2p} \quad \ldots \quad u_{kp}]'$. The squared error (unweighted) is calculated using the equation

$$D = \sum_{j=1}^{k} \left(a_{ji} - u_{jp}\right)^2$$

The weighted squared error is calculated using the equation

$$D = \sum_{j=1}^{k} w_j \left(a_{ji} - u_{jp}\right)^2$$

where the weighting factor is defined as $W = [w_1 \quad w_2 \quad \ldots \quad w_k]$. The index of the codeword that is associated with the minimum distortion is assigned to the input column vector.

You can select how you want to enter the codebook values using the **Source of codebook** parameter. When you select `Specify via dialog`, you can type the codebook values into the block parameters dialog box. Select `Input port` and port C appears on the block. The block uses the input to port C as the **Codebook** parameter.

The **Codebook** parameter is an k-by-N matrix of values, where $k \geq 1$ and $N \geq 1$. Each input column vector is compared to this codebook. Each column of the codebook matrix is a codeword, and each codeword has an index value. The first codeword vector corresponds to an index value of 0, the second codeword vector corresponds to an index value of 1, and so on. The codeword vectors must have the same number of rows as the input, U.

For the **Distortion measure** parameter, select `Squared error` when you want the block to calculate the distortion by evaluating the Euclidean distance between the input column vector and each codeword in the codebook. Select `Weighted squared error` when you want to use a weighting factor to emphasize or deemphasize certain input values.

For the **Source of weighting factor** parameter, select `Specify via dialog` to enter a weighting factor vector in the dialog box. Choose `Input port` to specify the weighting factor using port W.

Use the **Weighting factor** parameter to emphasize or deemphasize certain input values when calculating the distortion measure. For example, consider the p-th input column vector, U_p, as previously defined. When you want to neglect the effect of the first element of this vector, enter `[0 1 1 ... 1]` as the **Weighting factor** parameter. This weighting factor is used to calculate the weighted squared error using the equation

$$D = \sum_{j=1}^{k} w_j \left(a_{ji} - u_{jp}\right)^2$$

Because of the weighting factor used in this example, the weighted squared error is not affected by the first element of the input matrix. Therefore, the first element of the input column vector no longer impacts the choice of index value output by the Vector Quantizer Encoder block.

Use the **Index output data type** parameter to specify the data type of the index values output at port I. The data type of the index values can be `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`.

When an input vector is equidistant from two codewords, the block uses the **Tie-breaking rule** parameter to determine which index value the block chooses. When you want the input vector to be represented by the lower index valued codeword, select `Choose the lower index`. To represent the input column vector by the higher index valued codeword, select `Choose the higher index`.

Select the **Output codeword** check box to output at port Q(U) the codeword vectors that correspond to each index value. When the input is a matrix, the corresponding codeword vectors are horizontally concatenated into a matrix.

Select the **Output quantization error** check box to output at port D the quantization error that results when the block represents the input column vector by its nearest codeword. When the input is a matrix, the quantization error values are horizontally concatenated.

The Vector Quantizer Encoder block accepts real floating-point and fixed-point inputs. For more information on the data types accepted by each port, see “Data Type Support” on page 2-1892 or “Supported Data Types” on page 2-1898.

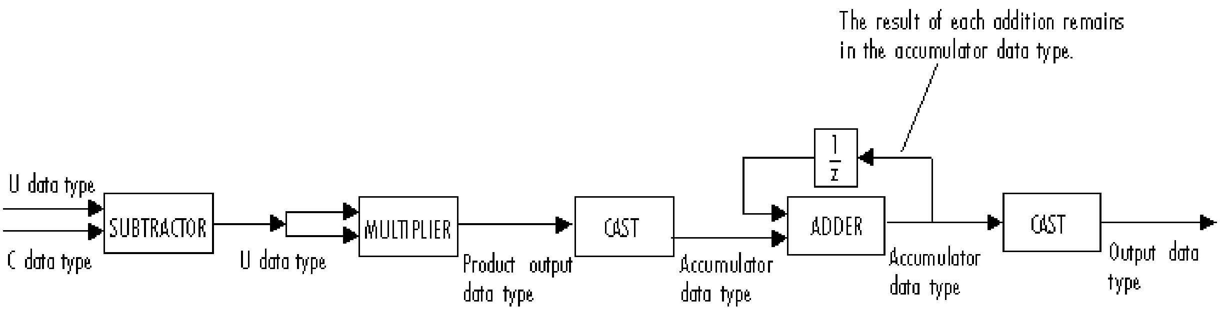
Data Type Support

The input data values, codebook values, and weighting factor values are input to the block at ports U, C, and W, respectively. The data type of the input data values, codebook values, and weighting factor values can be `double`, `single`, or Fixed-point. The input data, codebook values, and weighting factor must be the same data type.

The outputs of the block are the index values, output codewords, and quantization error. Use the **Index output data type** parameter to specify the data type of the index output from the block at port I. The data type of the index can be `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`. The data type of the output codewords and the quantization error can be `double`, `single`, or Fixed-point. The block assigns the data type of the output codewords and the quantization error based on the data type of the input data.

Fixed-Point Data Types

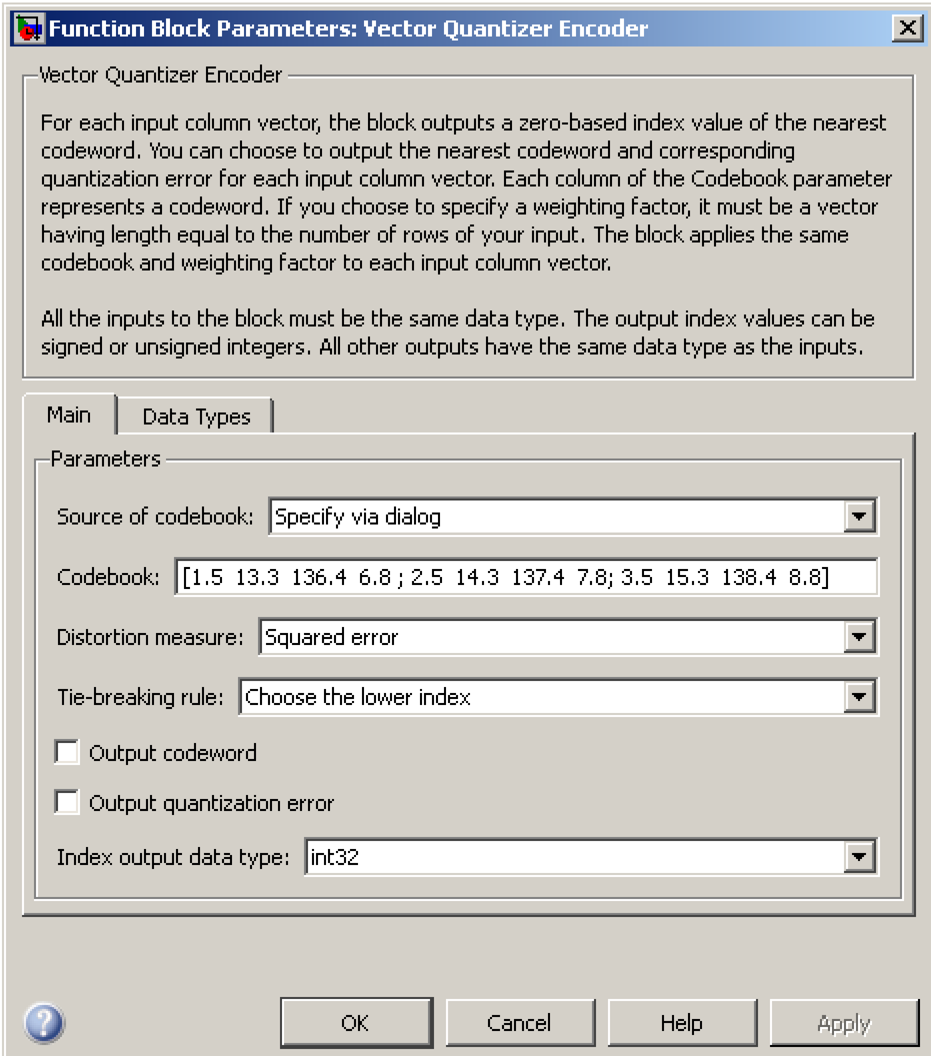
The following diagram shows the data types used within the Vector Quantizer Encoder block for fixed-point signals.



You can set the product output, accumulator, and index output data types in the block dialog as discussed below.

Dialog Box

The **Main** pane of the Vector Quantizer Encoder block dialog appears as follows.



Source of codebook

Choose `Specify via dialog` to type the codebook values into the block parameters dialog box. Select `Input port` to specify the codebook values using the block's input port, C.

Codebook

Enter a k-by-N matrix of values, where $1 \leq k$ and $1 \leq N$, to which your input column vector or matrix is compared. This parameter is visible if, from the **Source of codebook** list, you select `Specify via dialog`.

Distortion measure

Select `Squared error` when you want the block to calculate the distortion by evaluating the Euclidean distance between the input column vector and each codeword in the codebook. Select `Weighted squared error` when you want the block to calculate the distortion by evaluating a weighted Euclidean distance using a weighting factor to emphasize or deemphasize certain input values.

Source of weighting factor

Select `Specify via dialog` to enter a value for the weighting factor in the dialog box. Choose `Input port` and specify the weighting factor using port W on the block. This parameter is visible if, for the **Distortion measure** parameter, you select `Weighted squared error`.

Weighting factor

Enter a vector of values. This vector must have length equal to the number of rows of the input, U. This parameter is visible if, for the **Source of weighting factor** parameter, you select `Specify via dialog`.

Tie-breaking rule

Set this parameter to determine the behavior of the block when an input column vector is equidistant from two codewords. When you want the input column vector to be represented by the lower index valued codeword, select `Choose the lower index`. To represent the input column vector by the higher index valued codeword, select `Choose the higher index`.

Output codeword

Select this check box to output the codeword vectors nearest to the input column vectors.

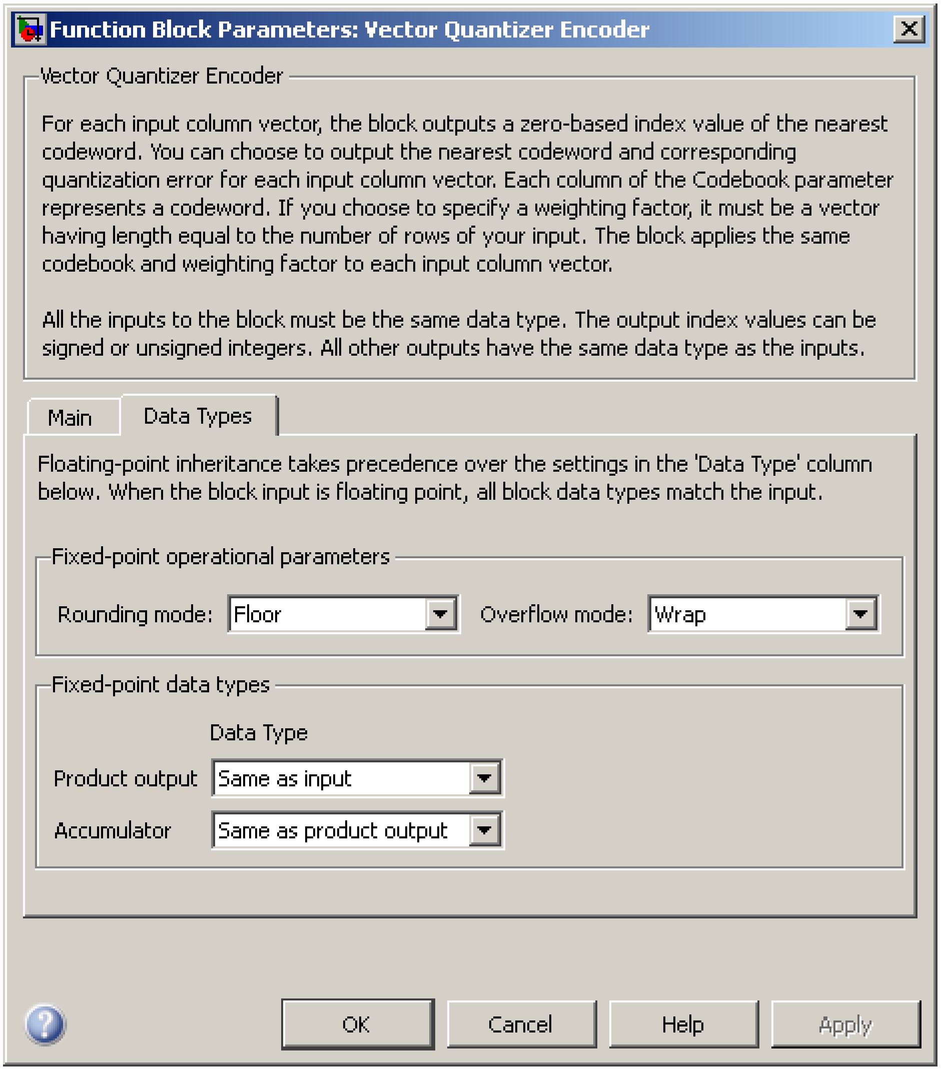
Output quantization error

Select this check box to output the quantization error value that results when the block represents the input column vector by the nearest codeword.

Index output data type

Select `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32` as the data type of the index output at port I.

The **Data Types** pane of the Vector Quantizer Encoder block dialog appears as follows.



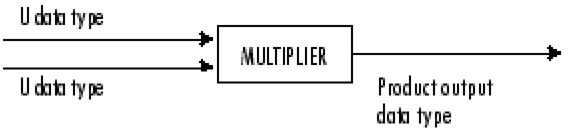
Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode to be used when block inputs are fixed point.

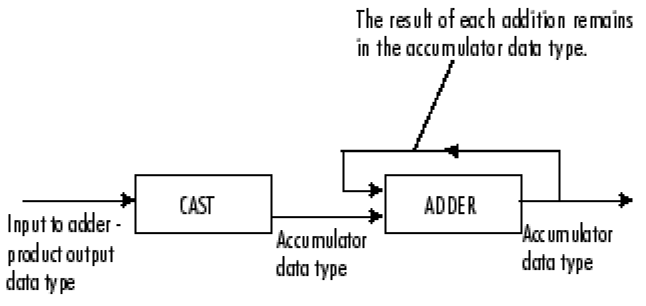
Product output



As depicted above, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how you would like to designate this product output word and fraction lengths.

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the product output, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and zero bias.

Accumulator



As depicted above, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how you would like to designate the accumulator word and fraction lengths.

- When you select `Same as product output`, these characteristics match those of the product output.
- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and zero bias.

References

Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

Port	Supported Data Types
U	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
C	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
W	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
I	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Q(U)	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
D	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

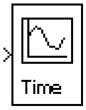
See Also

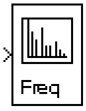
Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Scalar Quantizer Design	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
Uniform Decoder	DSP System Toolbox
Vector Quantizer Decoder	DSP System Toolbox

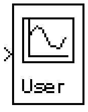
Introduced before R2006a

Vector Scope

Display vector or matrix of time-domain, frequency-domain, or user-defined data







Library

Sinks

`dspsnks4`

Description

Note: Consider using the `Array Plot` block, which offers more features than the Vector Scope. Additional features supported by the Array Plot are:

- Multiple inputs
- Cursors, Signal Statistics, and Peak Finder measurements
- Magnitude-phase plot
- Log *x*-axis scaling
- Advanced panning, zooming, and autoscaling
- Ability to customize the plot appearance, such as, figure and axes colors, line width, and legend text

---

The Vector Scope block is a comprehensive display tool similar to a digital oscilloscope. The block can display time-domain, frequency-domain, or user-defined signals. You can use the Vector Scope block to plot consecutive time samples from a vector, or to plot vectors containing data such as filter coefficients or spectral magnitudes. To compute and plot the periodogram of a signal with a single block, use the `Spectrum Analyzer` block.

The input to the Vector Scope block can be any real-valued matrix or vector. The block treats each column of an *M*-by-*N* matrix input as an independent channel of data with *M* consecutive samples.

The block plots each sample of each input channel sequentially across the horizontal axis of the plot.

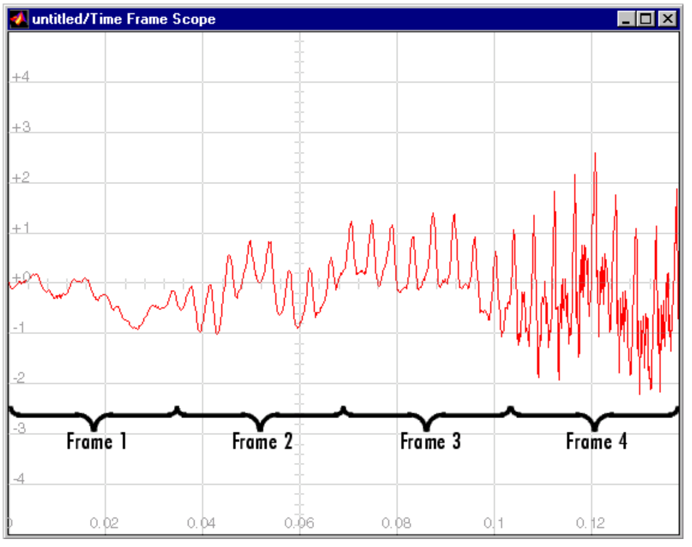
Scope Properties Pane

The **Scope Properties** pane enables you to plot time-domain, frequency-domain, or user-defined data, and adjust the horizontal display span of the plot. The scope displays frames of data, and updates the display for each new input frame.

The **Input domain** parameter specifies the domain of the input data. If you select `Time`, for *M*-by-*N* inputs containing time-domain data, the block treats each of the *N* input frames (columns) as a succession of *M* consecutive samples taken from a time series. That is, each data point in the input frame is assumed to correspond to a unique time value. Also, the **Time display span (number of frames)** parameter appears on the pane. Enter a scalar value greater than or equal to one that corresponds to the number of frames to be displayed across the width of the scope window.

If you select `Frequency` for the **Input domain** parameter, for *M*-by-*N* inputs containing frequency-domain data, the block treats each of the *N* input frames (columns) as a vector of spectral magnitude data corresponding to *M* consecutive ascending frequency indices. That is, when the input is a single column vector, u, each value in the input frame, `u(i)`, is assumed to correspond to a unique frequency value, `f(i)`, where `f(i+1)>f(i)`.

If you select `User-defined` for the **Input domain** parameter, the block does not assume that the input frame data is time-domain or frequency-domain data. You can plot the data in the appropriate manner. Also, the **Horizontal display span (number of frames)** parameter appears on the pane. Enter a scalar value greater than or equal to one that corresponds to the number of frames to be displayed across the width of the scope window.



Display Properties Pane

The **Display Properties** pane enables you to control how the block displays your data.

The **Show grid** parameter toggles the background grid on and off.

If you select the **Frame number** check box, the block displays the number of the current frame in the input sequence on the scope window, and the block increments the count as each new input is received. Counting starts at 1 with the first input frame, and continues until the simulation stops.

If you select the **Channel legend** check box, a legend indicating the line color, style, and marker of each channel's data is added. When the input signal is labeled, that label appears in the channel legend. When the input signal is not labeled, but comes from a Concatenate block or a Mux block with labeled inputs, those labels appear in the channel legend. Otherwise, each channel in the legend is labeled with the channel number (`CH 1`, `CH 2`, etc.). Click and drag the legend to reposition it in the scope window; double-click on the line label to edit the text. If you rerun the simulation, the labels revert to the defaults.

If you select the **Compact display** check box, the scope completely fills the figure window. The scope does not display menus and axis titles, but it does show the numerical axis labels within the axes. If you clear the **Compact display** check box, the scope displays the axis labels and titles in a gray border surrounding the scope axes, and the window's menus and toolbar become visible.

If you select the **Open scope at start of simulation** check box, the scope opens at the start of the simulation. If you clear this parameter, the scope does not open automatically during the simulation. You can use this feature when you have several scope blocks in a model, and you do not want to view all the associated scopes during the simulation.

Note: Before running a model that contains a Vector Scope block in Accelerator, Rapid Accelerator, or External mode, you must select the **Open scope at start of simulation** check box. If you do not select this check box before running your model for the first time, the scope will not display your simulation data.

If you want to view a scope window that is not open during simulation, click **Open scope immediately** on the **Display Properties** pane of the desired Scope block.

The **Scope position** parameter specifies a four-element vector of the form

```
[left bottom width height]
```

specifying the position of the scope window on the screen, where (0,0) is the lower-left corner of the display. See the MATLAB `figure` function for more information.

Axis Properties Pane

The parameters that are available on the **Axis Properties** pane depend on the setting of the **Input domain** parameter on the **Scope Properties** pane.

Time Domain Inputs

When **Time display limits** is set to `Auto`, the block scales the horizontal axis of time-domain signals automatically. The range of the time axis is $[0, S * T_{fi}]$, where T_{fi} is the input frame period, and S is the **Time display span (number of frames)** parameter on the **Scope Properties** pane. The spacing between time points is $T_{fi}/(M-1)$, where M is the number of samples in each consecutive input frame.

When **Time display limits** is set to `User-defined`, the **Minimum X-limit (s)** and **Maximum X-limit (s)** parameters set the range of the horizontal axis.

Minimum Y-limit and **Maximum Y-limit** parameters set the range of the vertical axis. Setting these parameters is analogous to setting the `ymin` and `ymax` values of the MATLAB axis function.

The **Y-axis label** is the text displayed to the left of the *y*-axis.

Frequency Domain Inputs

The **Frequency units** parameter specifies whether the frequency axis values should be in units of Hertz or rad/sec. When the **Frequency units** parameter is set to `Hertz`, the spacing between frequency points is $1/(M * T_s)$, where T_s is the sample time of the original time-domain signal. When the **Frequency units** parameter is set to `rad/sec`, the spacing between frequency points is $2\pi/(M * T_s)$.

The **Frequency range** parameter specifies the range of frequencies over which the magnitudes in the input should be plotted. The available options are `[0..Fs/2]`, `[-Fs/2..Fs/2]`, and `[0..Fs]`, where `Fs` is the original time-domain signal's sample frequency. The Vector Scope block assumes that the input data spans the range $[0, F_s)$, which is the same as the output from an FFT. To plot over the range `[0..Fs/2]` the scope truncates the input vector, leaving only the first half of the data, then plots these remaining samples over half the frequency range. To plot over the range `[-Fs/2..Fs/2]`, the scope reorders the input vector elements such that the last half of the data becomes the first half, and vice versa; then it relabels the *x*-axis accordingly.

If you select the **Inherit sample time from input** check box for frequency domain inputs, the block scales the frequency axis by reconstructing the frequency data from the frame-period of the frequency-domain input. This is valid when the following conditions hold:

- Each frame of frequency-domain data shares the same length as the frame of time-domain data from which it was generated; for example, when the FFT is computed on the same number of points as are contained in the time-domain input.
- The sample period of the time-domain signal in the simulation is equal to the period with which the physical signal was originally sampled.
- Consecutive frames containing the time-domain signal do not overlap each other; that is, a particular signal sample does not appear in more than one sequential frame.

In cases where not all of these conditions hold, specify the appropriate value for the **Sample time of original time series** parameter.

When **Frequency display limits** is set to `Auto`, the block scales the horizontal axis of frequency-domain signals automatically. To do this, the Vector Scope block needs to know the sample period of the original time-domain sequence represented by the frequency-domain data. Specify this period by entering a value for the **Sample time of original time series** parameter.

When **Frequency display limits** is set to `User-defined`, the **Minimum frequency** and **Maximum frequency** parameters set the range of the horizontal axis.

The **Y-axis scaling** parameter allows you to select `Magnitude` or `dB` scaling along the *y*-axis.

Minimum Y-limit and **Maximum Y-limit** parameters set the range of the vertical axis. Setting these parameters is analogous to setting the `ymin` and `ymax` values of the MATLAB axis function.

The **Y-axis label** is the text displayed to the left of the *y*-axis.

User-Defined Inputs

If you select the **Inherit sample increment from input** check box for user-defined input domains, the block scales the horizontal axis by computing the horizontal interval between samples in the input frame from the frame period of the input. For example, when the input frame period is 1, and there are 64 samples per input frame, the interval between samples is computed to be 1/64. Computing the interval this way is usually only valid when the following conditions hold:

- The input is a nonoverlapping time series; the *x*-axis on the scope represents time.
- The input sample period (1/64 in the above example) is equal to the period with which the physical signal was originally sampled.

In cases where not all of these conditions hold, use the **X display offset (samples)** and **Increment per sample in input** parameters.

The **X-axis title** is the text displayed below the *x*-axis.

When **X display limits** is set to `Auto`, the block scales the horizontal axis of user-defined domain signals automatically. To do this, the Vector Scope block needs to know the spacing of the input data. Specify this spacing using the **Increment per sample in input** parameter, I_s. This parameter represents the numerical interval between adjacent *x*-axis points corresponding to the input data. The range of the horizontal axis is $[0, M * I_s * S]$, where M is the number of samples in each consecutive input frame, and S is the **Horizontal display span (number of frames)** parameter that you specify in the **Scope Properties** pane.

When **X display limits** is set to `User-defined`, the **Minimum X-limit (samples)** and **Maximum X-limit (samples)** parameters set the range of the horizontal axis.

Minimum Y-limit and **Maximum Y-limit** parameters set the range of the vertical axis. Setting these parameters is analogous to setting the `ymin` and `ymax` values of the MATLAB `axis` function.

The **Y-axis label** is the text displayed to the left of the *y*-axis.

Line Properties Pane

Use the parameters on the **Line Properties** pane to help you distinguish between two or more independent channels of data on the scope.

The **Line visibilities** parameter specifies which channel's data is displayed on the scope, and which is hidden. The syntax specifies the visibilities in list form, where the term `on` or `off` as a list entry specifies the visibility of the corresponding channel's data. The list entries are separated by the pipe symbol, `|`.

For example, a five-channel signal would ordinarily generate five distinct plots on the scope. To disable plotting of the third and fifth lines, enter the following visibility specification in the **Line visibilities** parameter.

```
 on | on | off | on | off
ch 1 ch 2 ch 3  ch 4 ch 5
```

Note that the first (leftmost) list item corresponds to the first signal channel (leftmost column of the input matrix).

The **Line styles** parameter specifies the line style with which each channel's data is displayed on the scope. The syntax specifies the channel line styles in list form, with each list entry specifying a style for the corresponding channel's data. The list entries are separated by the pipe symbol, `|`.

For example, a five-channel signal would ordinarily generate all five plots with a solid line style. To plot each line with a different style, enter

```
   - | -- | : | -. | -
ch 1 ch 2 ch 3 ch 4 ch 5
```

These settings plot the signal channels with the following styles.

Line Style	Command to Type in Line Style Parameter	Appearance
Solid	-	

Line Style	Command to Type in Line Style Parameter	Appearance
Dashed	`--`	
Dotted	`:`	
Dash-dot	`-.`	
No line	`none`	No line appears

Note that the first (leftmost) list item, `'-'`, corresponds to the first signal channel (leftmost column of the input matrix). See the LineStyle property of the MATLAB `line` function for more information about the style syntax.

The **Line markers** parameter specifies the marker style with which each channel's samples are represented on the scope. The syntax specifies the channels' marker styles in list form, with each list entry specifying a marker for the corresponding channel's data. The list entries are separated by the pipe symbol, `|`.

For example, a five-channel signal would ordinarily generate all five plots with no marker symbol (that is, the individual sample points are not marked on the scope). To instead plot each line with a different marker style, you could enter

```
  * | . | x | s | d
ch 1 ch 2 ch 3 ch 4 ch 5
```

These settings plot the signal channels with the following styles.

Marker Style	Command to Type in Marker Style Parameter	Appearance
Asterisk	`*`	
Point	`.`	
Cross	`x`	
Square	`s`	
Diamond	`d`	

Note that the leftmost list item, `'*'`, corresponds to the first signal channel or leftmost column of the input matrix. See the LineSpec (Line Specification) (MATLAB) property of the MATLAB `line` function for more information about the available markers.

To produce a stem plot for the data in a particular channel, type the word `stem` instead of one of the basic marker shapes.

The **Line colors** parameter specifies the color in which each channel's data is displayed on the scope. The syntax specifies the channel colors in list form, with each list entry specifying a color (in one of the MATLAB `ColorSpec` formats) for the corresponding channel's data. The list entries are separated by the pipe symbol, `|`.

For example, a five-channel signal would ordinarily generate all five plots in the color black. To instead plot the lines with the color order below, enter

```
[0 0 0] | [0 0 1] | [1 0 0 ] | [0 1 0] | [.7529 0 .7529]
 ch 1      ch 2      ch 3       ch 4          ch 5
```

or

```
'k' | 'b' | 'r' | 'g' | [.7529 0 .7529]
ch 1  ch 2  ch 3  ch 4      ch 5
```

These settings plot the signal channels in the following colors (8-bit RGB equivalents shown in the center column).

Color	RGB Equivalent	Appearance
Black	(0,0,0)	
Blue	(0,0,255)	
Red	(255,0,0)	
Green	(0,255,0)	
Dark purple	(192,0,192)	

Note that the leftmost list item, `'k'`, corresponds to the first signal channel or leftmost column of the input matrix. See the MATLAB function `ColorSpec` for more information about the color syntax.

Vector Scope Window

The title that appears in the title bar of the scope window is the same as the block title. In addition to the standard MATLAB figure window menus such as **File**, **Window**, and **Help**, the Vector Scope window contains **View**, **Axes**, and **Channels** menus.

The options in the **View** menu allow you to zoom in and out of the scope window:

- To zoom in on the scope window, you must first select **View** > **Zoom In** or click the corresponding `Zoom In` toolbar button ( ). You can then zoom in by clicking in the center of your area of interest, or by clicking and dragging your cursor to draw a rectangular area of interest inside of the scope window.
- To zoom in on the *x*-axis of the scope window, you must first select **View** > **Zoom X**, or click the corresponding `Zoom X-Axis` toolbar button ( ) on the scope window. You can then zoom in on the *x*-axis with a single click inside the scope window or by clicking and dragging the cursor along the *x*-axis over your area of interest.
- To zoom in on the *y*-axis of the scope window, you must first select **View** > **Zoom Y** or click the corresponding `Zoom Y-Axis` toolbar button ( ). You can then zoom in on the *y*-axis with a single click inside the scope window or by clicking and dragging the cursor along the *y*-axis over your area of interest.
- To return to the original view of the scope window, you have the following options:
  - Select **Full View** from the **View** menu .
  - Click the `Restore default view` toolbar button ( ) on the Vector Scope window.
  - Right-click inside the scope window, and select `Reset to Original View`.

Note: To zoom out in smaller increments, you can right-click inside of the scope window and select `Zoom Out`. You can also zoom out by holding down the **Shift** key and clicking the left mouse button inside the scope window.

The parameters that you set using the **Axes** menu apply to all channels. Many of the parameters in this menu are also accessible through the block parameters dialog box. For descriptions of these parameters, see “Display Properties Pane” on page 2-1902. Below are descriptions of other parameters in the **Axes** menu:

- **Refresh** erases all data on the scope display, except for the most recent trace.
- **Autoscale** resizes the *y*-axis to best fit the vertical range of the data.

Note: The **Minimum Y-limit** and **Maximum Y-limit** parameters on the **Axis properties** pane of the block dialog are not updated to display the numerical limits selected by the autoscale feature.

- **Save Axes Settings** allows you to save the current axes settings. When you select this option, the **Minimum Y-limit** and **Maximum Y-limit** parameters of the **Axes Properties** pane update with the current *y*-axes limits. The **Time display limits** (or **Frequency display limits**) parameter is set to `User-defined`, and the current *x*-axes limits are saved in the **Minimum X-limit** and **Maximum X-limit** (or **Minimum Frequency** and **Maximum Frequency**) parameters. To save these axes settings for your next MATLAB session, you need to resave your model.
- **Save Scope Position** updates the **Scope position** parameter on the **Display Properties** pane of the block dialog to reflect the scope window's current position and size. To make the scope window open at a particular location on the screen when the simulation runs, drag the window to the desired location, resize it, and select **Save Scope Position** from the **Axes** menu.

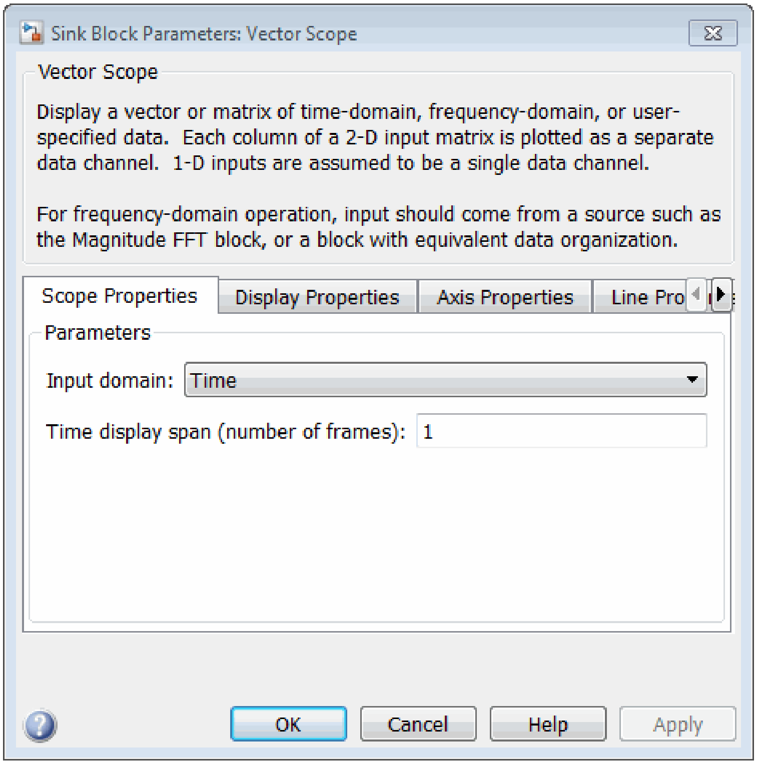
The properties listed in the **Channels** menu apply to a particular channel. All of the parameters in this menu are also accessible through the block parameters dialog box. For descriptions of these parameters, see “Line Properties Pane” on page 2-1907.

Many of these options can also be accessed by right-clicking with the mouse anywhere on the scope display. The menu that is displayed contains a combination of the options available in the **View**, **Axes** and **Channels** menus.

Note When you select **Compact Display** from the **Axes** menu, the scope window menus are no longer visible. Right-click in the Vector Scope window and click **Compact Display** in order to make the menus reappear.

Dialog Box

Scope Properties Pane



Input domain

Select the domain of the input. Your choices are `Time`, `Frequency`, or `User-defined`. Tunable (Simulink).

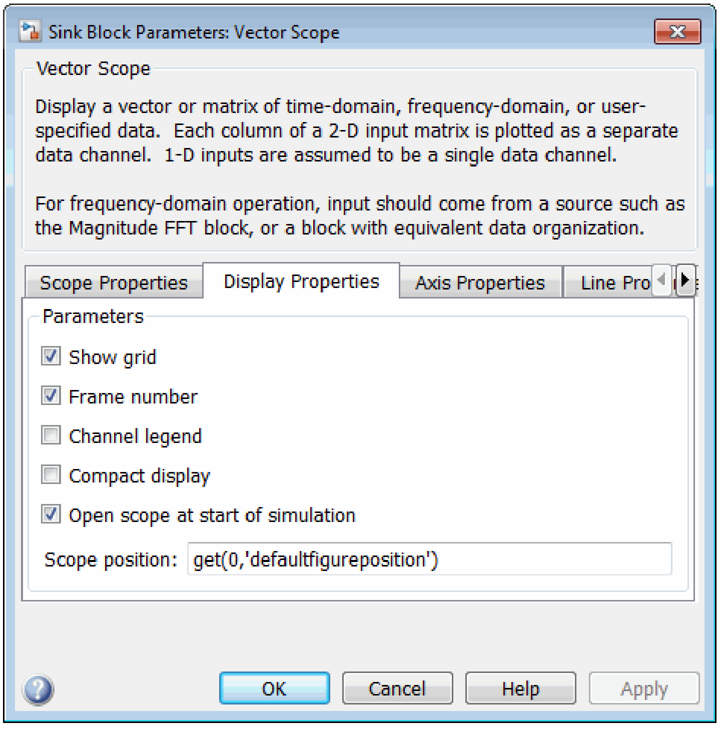
Time display span (number of frames)

The number of consecutive frames to display (horizontally) on the scope at any one time. This parameter is visible when the **Input domain** parameter is set to `Time`.

Horizontal display span (number of frames)

The number of consecutive frames to display (horizontally) on the scope at any one time. This parameter is visible when the **Input domain** parameter is set to `User-defined`.

Display Properties Pane



Show grid

Toggle the scope grid on and off.Tunable (Simulink).

Frame number

If you select this check box, the number of the current frame in the input sequence appears in the Vector Scope window. Tunable (Simulink).

Channel legend

Toggles the legend on and off. Tunable (Simulink).

Compact display

Resizes the scope to fill the window. Tunable (Simulink).

Open scope at start of simulation

Select this check box to open the scope at the start of the simulation. When this parameter is cleared, the scope does not open automatically during the simulation. Tunable (Simulink).

Note: Before running a model that contains a Vector Scope block in Accelerator, Rapid Accelerator, or External mode, you must select the **Open scope at start of simulation** check box. If you do not select this check box before running your model for the first time, the scope will not display your simulation data.

Open scope immediately

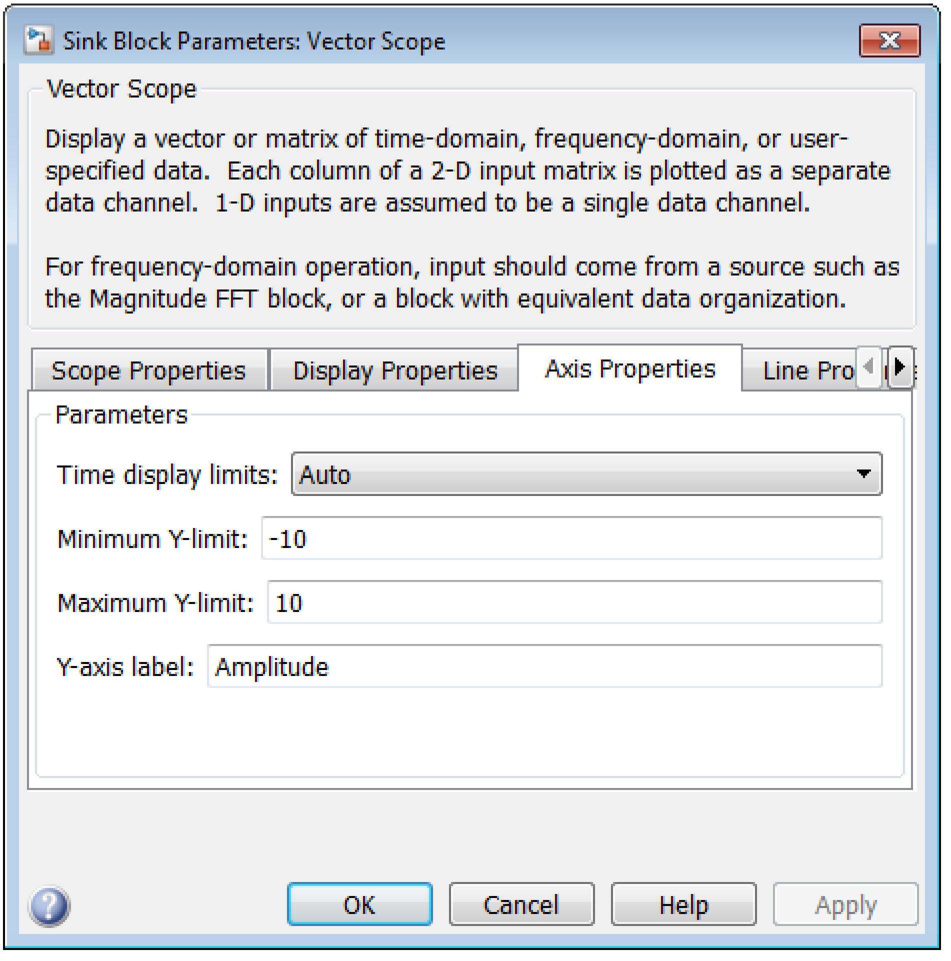
If the scope is not open during simulation, select this check box to open it. This parameter is visible only while the simulation is running.

Scope position

A four-element vector of the form `[left bottom width height]` specifying the position of the scope window. `(0,0)` is the lower-left corner of the display. Tunable (Simulink).

Axis Properties Pane

The parameters that are available on the **Axis Properties** pane depend on the setting of the **Input domain** parameter on the **Scope Properties** pane. When `Time` is selected for the **Input domain** parameter, the following parameters are available on the **Axis Properties** pane:



Time display limits

Select `Auto` to have the limits of the *x*-axis set for you automatically, or `User-defined` to set the limits yourself in the **Minimum X-limit (s)** and **Maximum X-limit (s)** parameters.

Minimum X-limit (s)

Specify the minimum value of the *x*-axis in seconds. This parameter is only visible if the **Time display limits** parameter is set to `User-defined`. Tunable (Simulink).

Maximum X-limit (s)

Specify the maximum value of the *x*-axis in seconds. This parameter is only visible if the **Time display limits** parameter is set to `User-defined`. Tunable (Simulink).

Minimum Y-limit

Specify the minimum value of the *y*-axis. Setting this parameter is analogous to setting the `ymin` value of the MATLAB `axis` function. Tunable (Simulink).

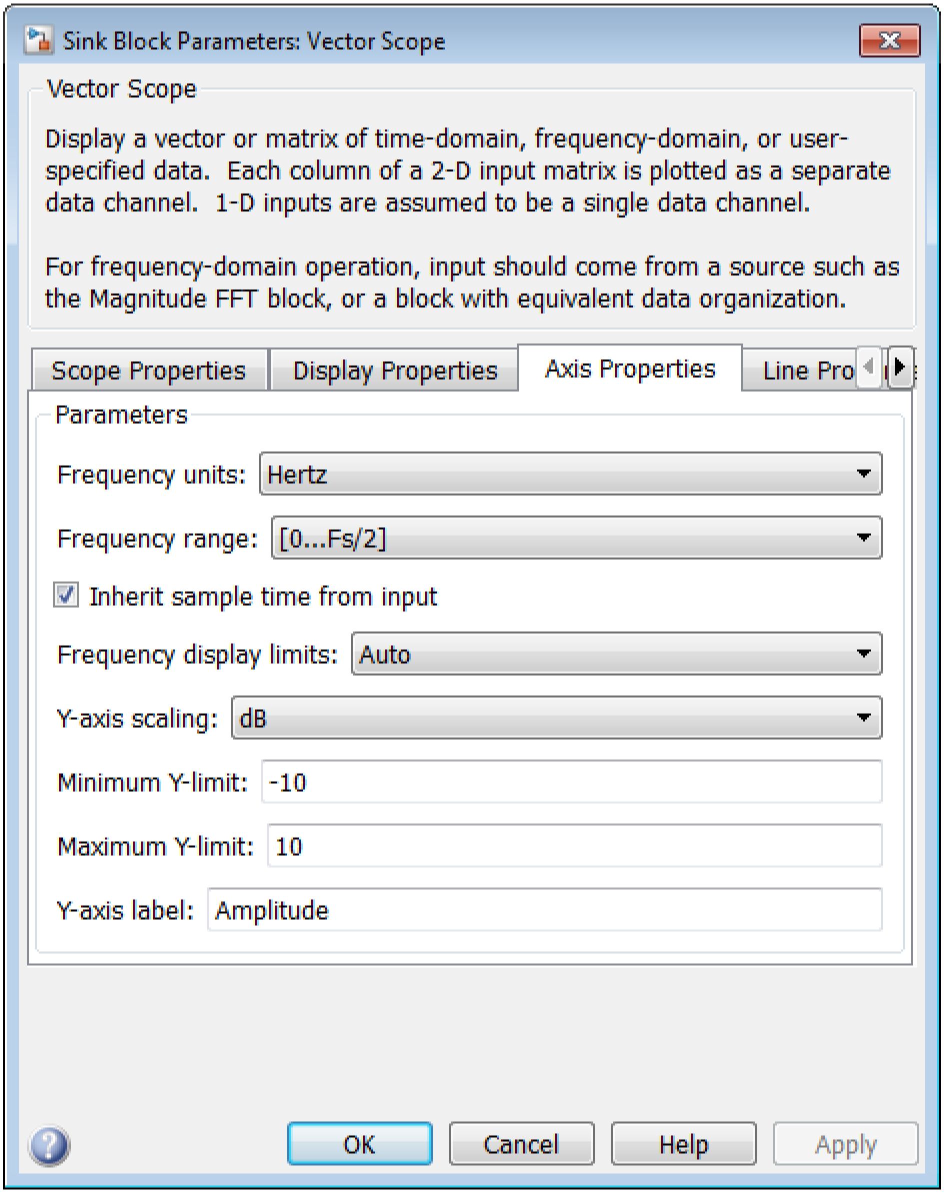
Maximum Y-limit

Specify the maximum value of the *y*-axis. Setting this parameter is analogous to setting the `ymax` value of the MATLAB `axis` function. Tunable (Simulink).

Y-axis label

Specify text to be displayed to the left of the *y*-axis. Tunable (Simulink).

When `Frequency` is selected for the **Input domain** parameter, the following parameters are available on the **Axis Properties** pane:



Frequency units

Choose the frequency units for the x-axis, `Hertz` or `rad/sec`. Tunable (Simulink).

Frequency range

Specify the frequency range over which to plot the data. Tunable (Simulink).

Inherit sample time from input

If you select this check box, the block computes the time-domain sample period from the frame period and frame size of the frequency-domain input. Use this parameter only when the length of the each frame of frequency-domain data is the same as the length of the frame of time-domain data from which it was generated. Tunable (Simulink).

Sample time of original time series

Enter the sample period, T_s, of the original time-domain signal. This parameter is only visible when the **Inherit sample time from input** check box is not selected. Tunable (Simulink).

Frequency display limits

Select `Auto` to have the limits of the *x*-axis set for you automatically, or `User-defined` to set the limits yourself in the **Minimum frequency** and **Maximum frequency** parameters.

Minimum frequency

Specify the minimum frequency value of the *x*-axis in Hertz or rad/sec. This parameter is only visible if the **Frequency display limits** parameter is set to `User-defined`. Tunable (Simulink).

Maximum frequency

Specify the maximum frequency value of the *x*-axis in Hertz or rad/sec. This parameter is only visible if the **Frequency display limits** parameter is set to `User-defined`. Tunable (Simulink).

Y-axis scaling

Choose either `dB` (decibel) or `Magnitude` scaling for the *y*-axis. Tunable (Simulink).

Minimum Y-limit

Specify the minimum value of the *y*-axis. Setting this parameter is analogous to setting the `ymin` value of the MATLAB `axis` function. Tunable (Simulink).

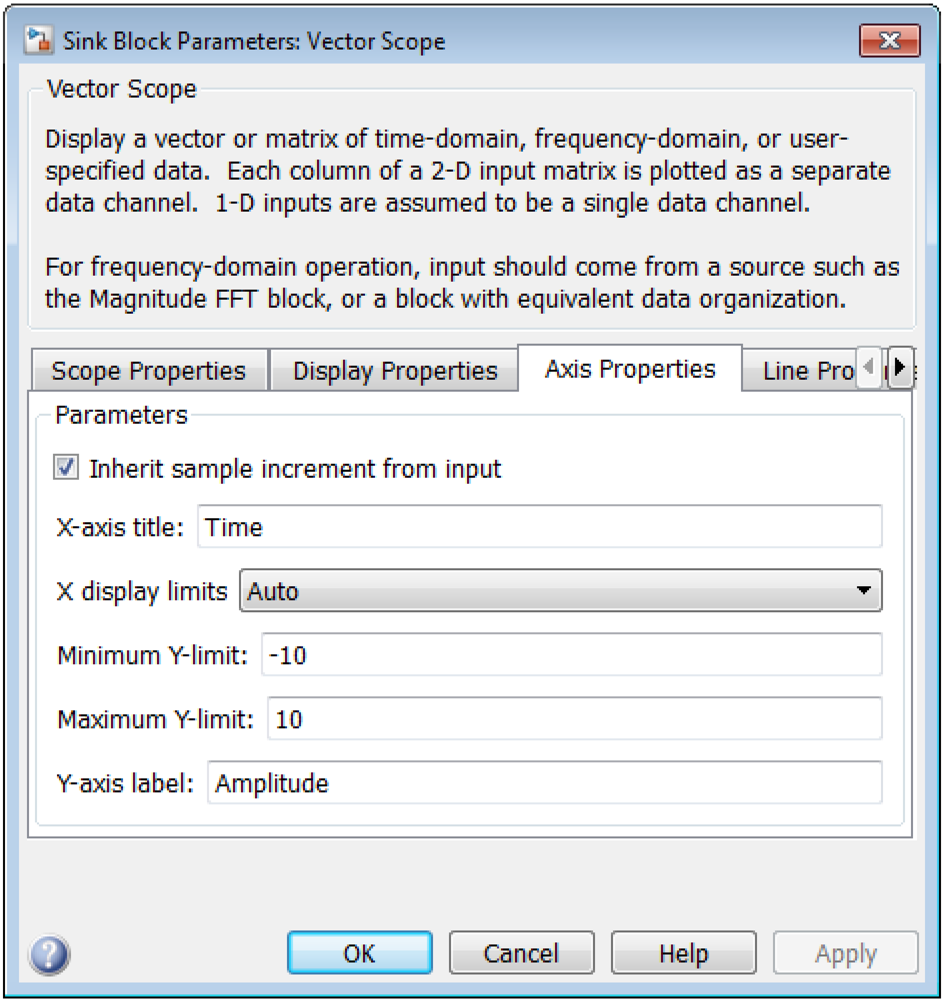
Maximum Y-limit

Specify the maximum value of the *y*-axis. Setting this parameter is analogous to setting the `ymax` value of the MATLAB `axis` function. Tunable (Simulink).

Y-axis label

Specify text to be displayed to the left of the *y*-axis. Tunable (Simulink).

When `User-defined` is selected for the **Input domain** parameter, the following parameters are available on the **Axis Properties** pane:



Inherit sample increment from input

When you select this check box, the block scales the horizontal axis by computing the horizontal interval between samples in the input frame from the frame period of the input. Use this parameter only when the input's sample period is equal to the period with which the physical signal was originally sampled. Tunable (Simulink).

Increment per sample in input

Enter the numerical interval between adjacent *x*-axis points corresponding to the user-defined input data. This parameter is only visible when the **Inherit sample increment from input** check box is not selected. Tunable (Simulink).

X display offset (samples)

Specify an offset for the *x*-axis display in samples. This parameter is only visible when the **Inherit sample increment from input** check box is not selected. Tunable (Simulink).

X-axis title

Enter the text to be displayed below the *x*-axis. Tunable (Simulink).

X display limits

Select `Auto` to have the limits of the *x*-axis set for you automatically, or `User-defined` to set the limits yourself in the **Minimum X-limit (samples)** and **Maximum X-limit (samples)** parameters.

Minimum X-limit (samples)

Specify the minimum value of the *x*-axis in samples. This parameter is only visible if the **X display limits** parameter is set to `User-defined`. Tunable (Simulink).

Maximum X-limit (samples)

Specify the maximum value of the *x*-axis in samples. This parameter is only visible if the **X display limits** parameter is set to `User-defined`. Tunable (Simulink).

Minimum Y-limit

Specify the minimum value of the *y*-axis. Setting this parameter is analogous to setting the `ymin` value of the MATLAB `axis` function. Tunable (Simulink).

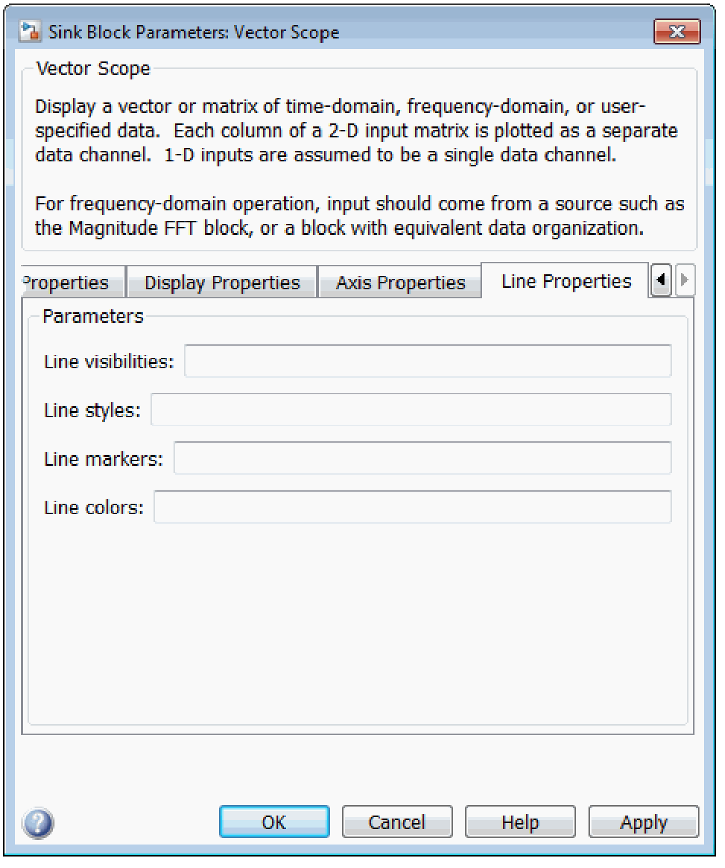
Maximum Y-limit

Specify the maximum value of the *y*-axis. Setting this parameter is analogous to setting the `ymax` value of the MATLAB `axis` function. Tunable (Simulink).

Y-axis label

Specify text to be displayed to the left of the *y*-axis. Tunable (Simulink).

Line Properties Pane



Line visibilities

Enter `on` or `off` to specify the visibility of the various channels' scope traces. Separate your choices for each channel with by a pipe (|) symbol. Tunable (Simulink).

Line styles

Enter the line styles of the various channels' scope traces. Separate your choices for each channel with by a pipe (|) symbol. Tunable (Simulink).

Line markers

Enter the line markers of the various channels' scope traces. Separate your choices for each channel with by a pipe (|) symbol. Tunable (Simulink).

Line colors

Enter the colors of the various channels' scope traces using the `ColorSpec` formats. Separate your choices for each channel with by a pipe (|) symbol. Tunable (Simulink).

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

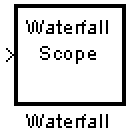
See Also

`Matrix Viewer`	DSP System Toolbox
`Spectrum Analyzer`	DSP System Toolbox

Introduced before R2006a

Waterfall

View vectors of data over time



Library

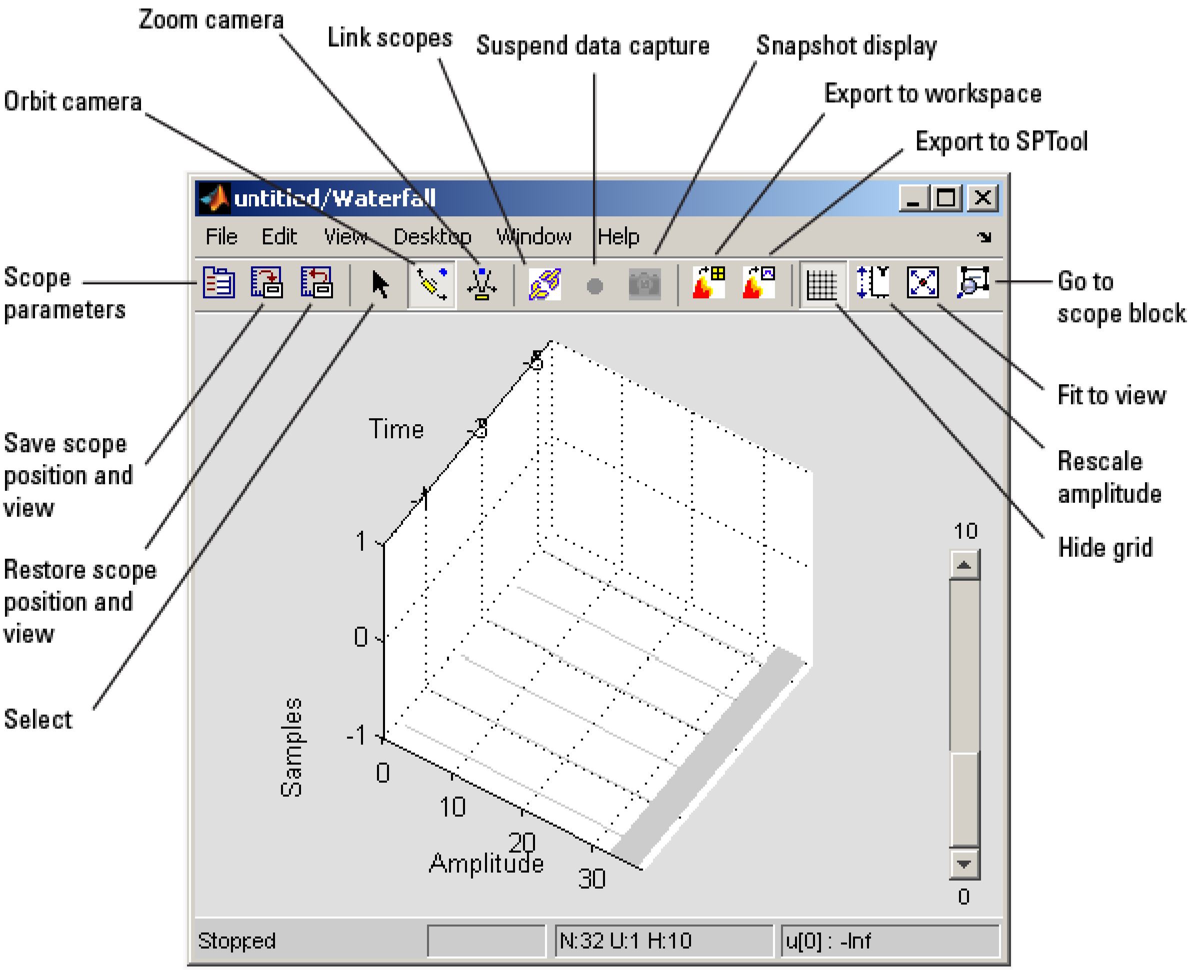
Sinks

`dspsnks4`

Description

The Waterfall block displays multiple vectors of data at one time. These vectors represent the input data at consecutive sample times. The input to the block can be real or complex-valued data vectors of any data type including fixed-point data types. However, the input is converted to double-precision before the block processes the data. The Waterfall block displays only real-valued, double-precision vectors of data.

The data is displayed in a three-dimensional axis in the Waterfall window. By default, the x-axis represents amplitude, the y-axis represents samples, and the z-axis represents time. You can adjust the number of sample vectors that the block displays, move and resize the Waterfall window, and modify block parameter values during the simulation. The Waterfall window has toolbar buttons that enable you to zoom in on displayed data, suspend data capture, freeze the scope's display, save the scope position, and export data to the workspace. The toolbar buttons are labeled in the following figure, which shows the Waterfall window as it appears when you double-click a Waterfall block.



Sections of This Reference Page

- "Waterfall Parameters" on page 2-1925
- "Display Parameters" on page 2-1927
- "Axes Parameters" on page 2-1928
- "Data History Parameters" on page 2-1929
- "Triggering Parameters" on page 2-1931

- "Scope Trigger Function" on page 2-1933
- "Transform Parameters" on page 2-1935
- "Scope Transform Function" on page 2-1938
- "Examples" on page 2-1938

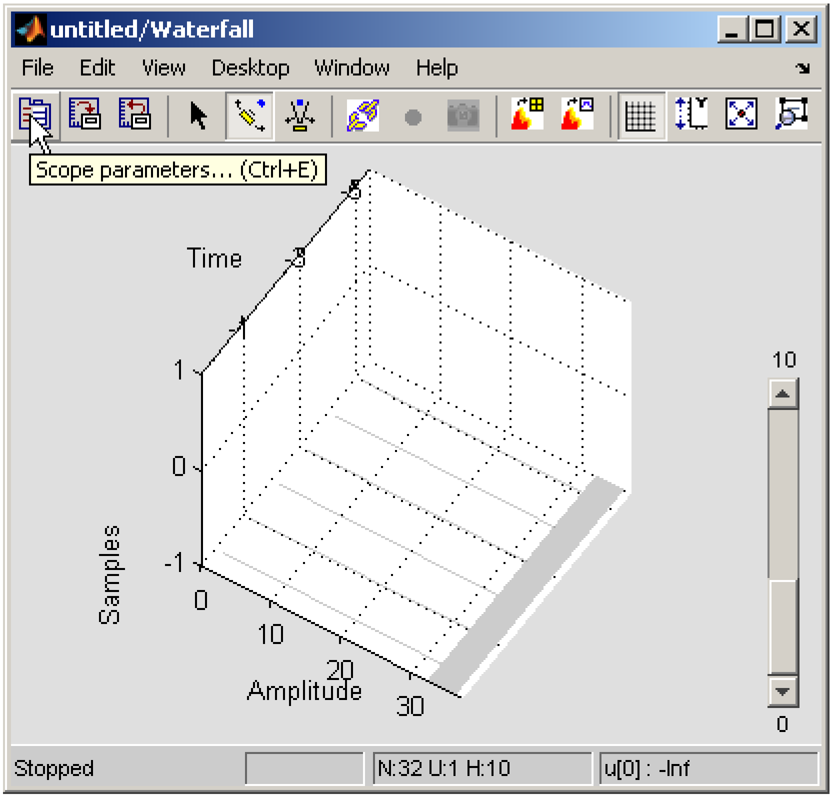
Waterfall Parameters

You can control the display and behavior of the Waterfall window using the Parameters dialog box.

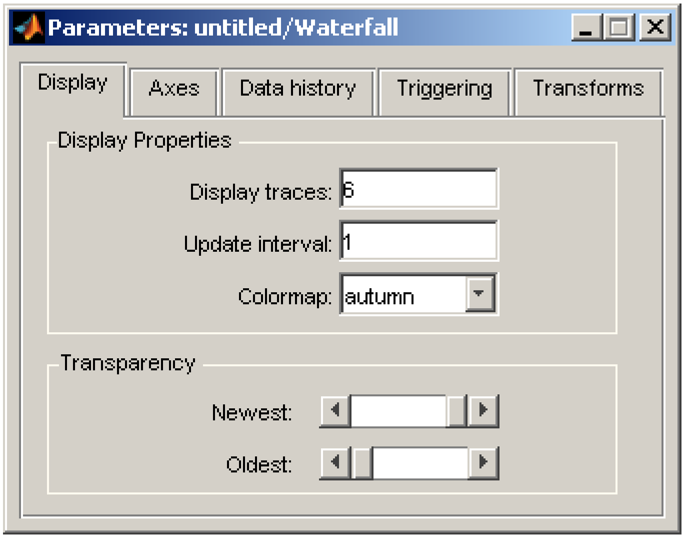
Note You can alter the Waterfall parameters while the simulation is running. However, when you make changes to values in text boxes, you must click **Enter** or click outside the text box before the block accepts your changes.

---

1 To open the Parameters dialog box, click the **Scope parameters** button.



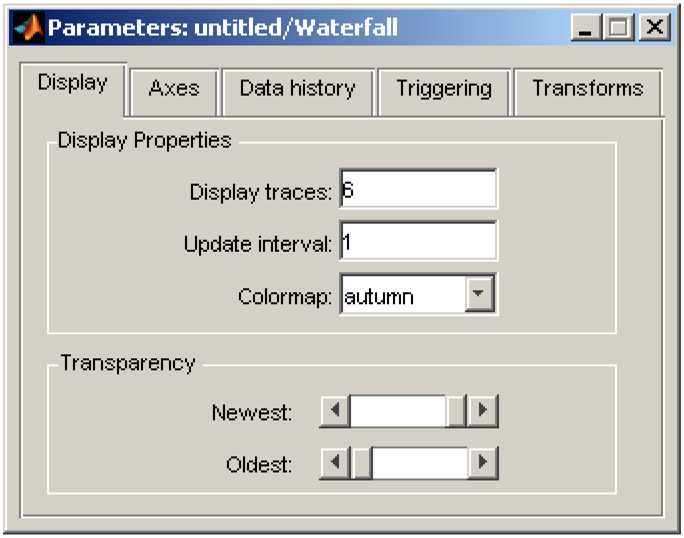
The Parameters dialog box appears.



2 Click on the different panes to enter parameter settings.

Display Parameters

The following parameters control the Waterfall window's display.



Display traces

Enter the number of vectors of data to be displayed in the Waterfall window.

Update interval

Enter the number of vectors the block should store before it displays them to the window.

Colormap

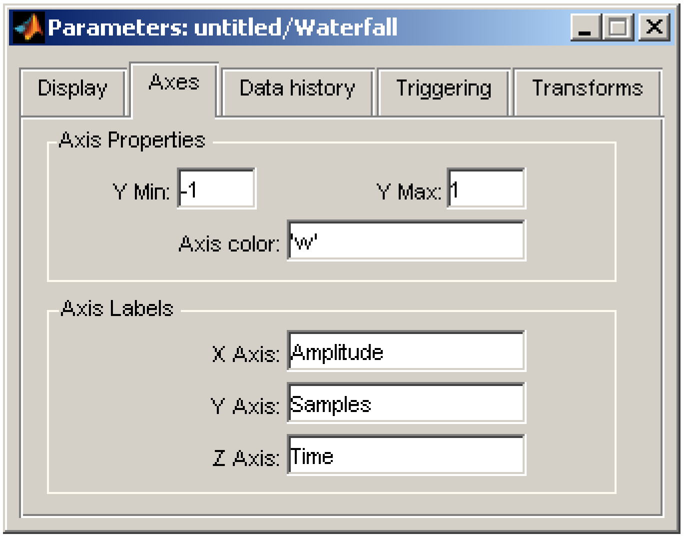
Choose a colormap for the displayed data.

Transparency

Specify the transparency of the newest and oldest data vectors. Placing the slider in the left-most position tells the block to make the data vector transparent. Placing the slider in the right-most position tells the block to make the data vector opaque. The intermediate data vectors transition between the two chosen transparency values.

Axes Parameters

The following parameters control the axes in the Waterfall window.



Y Min

Enter the minimum value of the *y*-axis.

Y Max

Enter the maximum value of the *y*-axis.

Axis color

Enter a background color for the axes. Specify the color using a character vector. For example, to specify black, enter `'k'`.

X Axis

Enter the *x*-axis label.

Y Axis

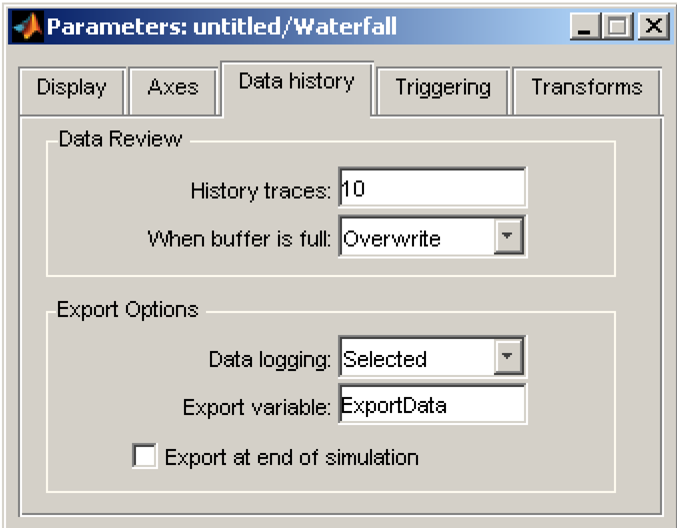
Enter the *y*-axis label.

Z Axis

Enter the *z*-axis label.

Data History Parameters

The following parameters control how many input data vectors the Waterfall block stores. They also control how the data is exported to the MATLAB workspace or SPTool.



History traces

Enter the number of vectors (traces) that you want the block to store.

When the buffer is full

Use this parameter to control the behavior of the block when the buffer is filled:

- `Overwrite` — The old data is replaced with the new data.
- `Suspend` — The block stops storing data in the buffer; however, the simulation continues to run.
- `Extend` — The block extends the buffer so that it can continue to store all the input data.

Data logging

Use this parameter to control which data is exported from the block:

- `Selected` — The selected data vector is exported.
- `All visible` — All of the data vectors displayed in the Waterfall window are exported.
- `All history` — All of the data vectors stored in the block's history buffer are exported.

Export variable

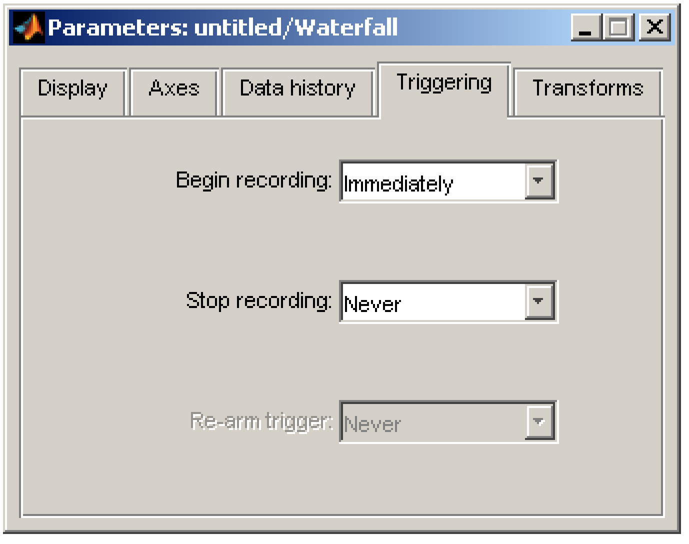
Enter the name of the variable that represents your data in the MATLAB workspace or SPTool. The default variable name is `ExportData`.

Export at end of simulation

Select this check box to automatically export the data to the MATLAB workspace when the simulation stops.

Triggering Parameters

The following parameters control when the Waterfall block starts and stops capturing data.



Begin recording

This parameter controls when the Waterfall block starts capturing data:

- `Immediately` — The Waterfall window captures the input data as soon as the simulation starts.
- `After T seconds` — The **Time, T** parameter appears in the dialog box. Enter the number of seconds the block should wait before it begins capturing data.
- `After N inputs` — The **Count, N** parameter appears in the dialog box. Enter the number of inputs the block should receive before it begins capturing data.

- `User-defined` — The **Function name** parameter appears in the dialog box. Enter the name of a MATLAB function that defines when the block should begin capturing data. For more information about how you define this function, see "Scope Trigger Function" on page 2-1933.

Stop recording

This parameter controls when the Waterfall block stops capturing data:

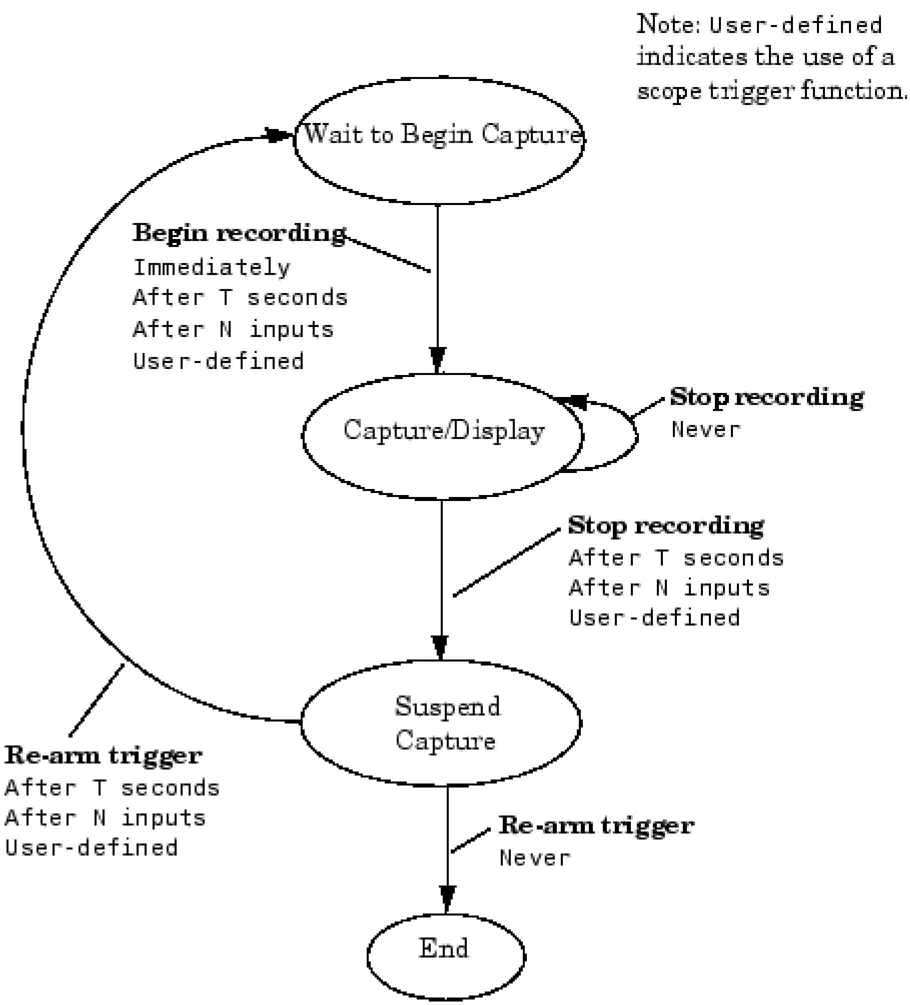
- `Never` — The block captures the input data as long as the simulation is running.
- `After T seconds` — The **Time, T** parameter appears in the dialog box. Enter the number of seconds the block should wait before it stops capturing data.
- `After N inputs` —The **Count, N** parameter appears in the dialog box. Enter the number of inputs the block should receive before it stops capturing data.
- `User-defined` — The **Function name** parameter appears in the dialog box. Enter the name of a MATLAB function that defines when the block should stop capturing data. For more information about how you define this function, see "Scope Trigger Function" on page 2-1933.

Re-arm trigger

This parameter controls when the Waterfall block begins waiting to capture data. It is available only when you select `After T seconds`, `After N inputs`, or `User-defined` for the **Stop recording** parameter:

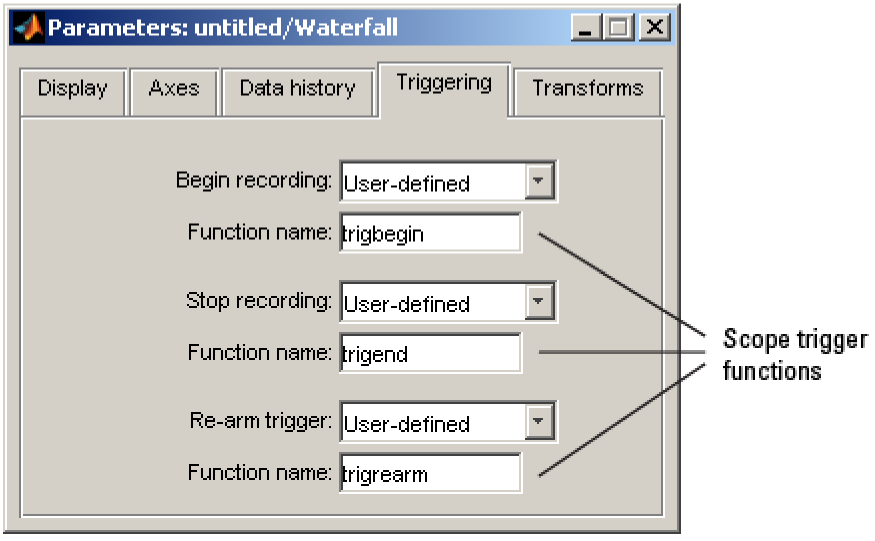
- `Never` — The Waterfall Scope block starts and stops capturing data as defined by the **Begin recording** and **Stop recording** parameters.
- `After T seconds` — The **Time, T** parameter appears in the dialog box. Enter the number of seconds the block should wait before it begins waiting to capture data.
- `After N inputs` — The **Count, N** parameter appears in the dialog box. Enter the number of inputs the block should receive before it begins waiting to capture data.
- `User-defined` — The **Function name** parameter appears in the dialog box. Enter the name of a MATLAB function that defines when the block should begin waiting to capture data. For more information about how you define this function, see "Scope Trigger Function" on page 2-1933.

The triggering process is illustrated in the state diagram below.



Scope Trigger Function

You can create custom scope trigger functions to control when the scope starts, stops, or begins waiting to capture data.



These functions must be valid MATLAB functions and be located either in the current folder or on the MATLAB path.

Each scope trigger function must have the following form

`y = functionname(blk,t,u),`

where `functionname` refers to the name you give your scope trigger function. The variable `blk` is the Simulink block handle. When the scope trigger function is called by the block, Simulink automatically populates this variable with the handle of the Waterfall block. The variable `t` is the current simulation time, represented by a real, double-precision, scalar value. The variable `u` is the vector input to the block. The output of the scope trigger function, `y`, is interpreted as a logical signal. It is either true or false:

- Begin recording scope trigger function
  - When the output of this scope trigger function is true, the Waterfall block starts capturing data.
  - When the output is false, the block remains in its current state.
- Stop recording scope trigger function
  - When the output of this scope trigger function is true, the block stops capturing data.

  - When the output is false, the block remains in its current state.
- Re-arm trigger scope trigger function
  - When the output of this scope trigger function is true, the block waits for a begin recording event.
  - When the output is false, the block remains in its current state.

Note The Waterfall block passes its input data directly to the scope trigger functions. These functions do not use the transformed data defined by the Transform parameters.

The following is an example of a scope trigger function. This function, called `trigPower` detects when the energy in u exceeds a certain threshold.

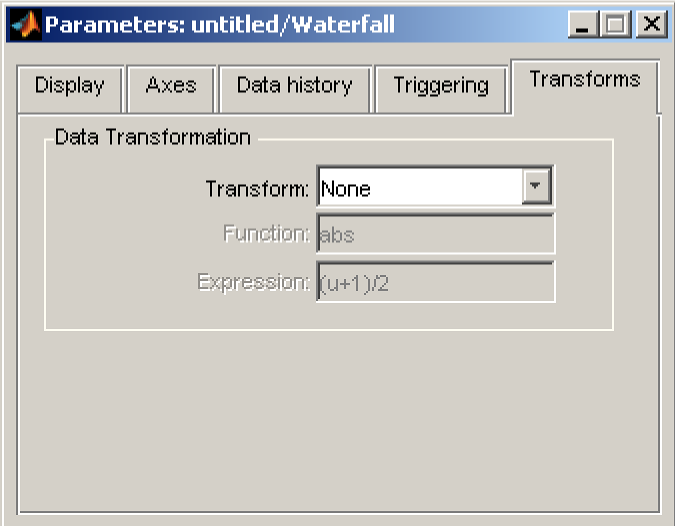
```
function y = trigPower(blk, t, u)

y = (u'*u > 2300);
```

The following is another example of a scope trigger function. This function, called `count3`, triggers the scope once three vectors with positive means are input to the block. Then, the function resets itself and begins searching for the next three input vectors with positive means. This scope trigger function is valid only when one Waterfall block is present in your model.

```
function y = count3(blk, t, u)

persistent state;
if isempty(state); state = 0; end
if mean(u)>0; state = state+1; end
y = (state>=3);
if y; state = 0; end
```

Transform Parameters

The following parameters transform the input data to the Waterfall block. The result of the transform is displayed in the Waterfall window.



Note The block assumes that the input to the block corresponds to the **Transform** parameter you select. For example, when you choose `Complex-> Angle`, the block assumes that the input is complex. The block does not produce an error when the input is not complex. Therefore, you must verify the format of your input data to guarantee that a meaningful result is displayed in the Waterfall window.

Transform

Choose a transform that you would like to apply to the input of the Waterfall block:

- `None` — The input is displayed as it is received by the block.
- `Amplitude-> dB` — The block converts the input amplitude into decibels.
- `Complex-> Mag Lin` — The block converts the complex input into linear magnitude.
- `Complex-> Mag dB` — The block converts the complex input into magnitude in decibels.
- `Complex-> Angle` — The block converts the complex input into phase.
- `FFT-> Mag Lin Fs/2` — The block takes the linear magnitude of the FFT input and plots it from 0 to the Nyquist frequency.

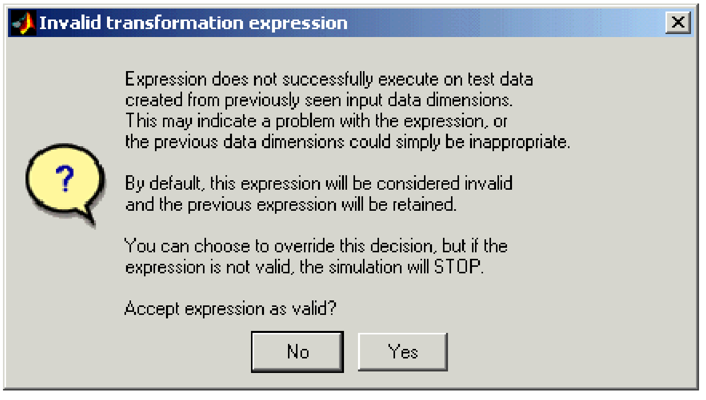
- `FFT-> Mag dB Fs/2` — The block takes the magnitude of the FFT input, converts it to decibels, and plots it from 0 to the Nyquist frequency.
- `FFT-> Angle Fs/2` — The block converts the FFT input into phase and plots it from 0 to the Nyquist frequency.
- `Power-> dB` — The block converts the input power into decibels.

Function

This parameter is only available when you select `User-defined fcn` for the **Transform** parameter. Enter a function that you would like to apply to the input of the Waterfall block. For more information about how you define this function, see "Scope Transform Function" on page 2-1938.

Expression

This parameter is only available when you select `User-defined expr` for the **Transform** parameter. Enter an expression that you would like to apply to the input of the Waterfall block. The result of this expression must be real-valued. When you write the expression, be sure to include only one unknown variable. The block assumes this unknown variable represents the input to the block. When the block believes your expression is invalid, the following window appears.



When you click **No**, your expression is not applied to the input. When you click **Yes** and your expression is invalid, your simulation stops and Simulink displays an error.

Scope Transform Function

You can create a scope transform function to control how the Waterfall block transforms your input data. This function must have a valid MATLAB function name and be located either in the current folder or on the MATLAB path.

Your scope transform function must have the following form

```
y = functionname(u),
```

where `functionname` refers to the name you give your function. The variable `u` is the real or complex vector input to the block. The output of the scope transform function, `y`, must be a double-precision, real-valued vector. When it is not, the simulation stops and Simulink displays an error. Note that the output vector does not need to be the same size as the input vector.

Examples

The following examples illustrate some capabilities of the Waterfall block.

- "Exporting Data" on page 2-1938
- "Capturing Data" on page 2-1939
- "Linking Scopes" on page 2-1939
- "Selecting Data" on page 2-1941
- "Zooming" on page 2-1943
- "Rotating the Display" on page 2-1943
- "Scaling the Axes" on page 2-1943
- "Saving Scope Settings" on page 2-1943

Exporting Data

You can use the Waterfall block to export data to the MATLAB workspace or to SPTool:

1. Open and run the `dspanc` example.
2. While the simulation is running, click the **Export to Workspace** button.
3. Type `whos` at the MATLAB command line.

The variable `ExportData` appears in your MATLAB workspace. `ExportData` is a 40-by-6 matrix. This matrix represents the six data vectors that were present in the Waterfall window at the time you clicked the **Export to Workspace** button. Each column of this matrix contains 40 filter coefficients. The columns of data were captured at six consecutive instants in time.

You can control what data is exported using the **Data logging** parameter in **Data history** pane of the Parameters dialog box. For more information, see "Data History Parameters" on page 2-1929.

4. While the simulation is running, click the **Export to SPTool** button.

   The SPTool GUI opens and the variable `ExportData` is displayed in the **Signals** list.

For more information about SPTool, see the Signal Processing Toolbox documentation.

Capturing Data

You can use the Waterfall block to interact with your data while it is being captured:

1. Open and run the `dspanc` example.
2. While the simulation is running, click the **Suspend data capture** button.

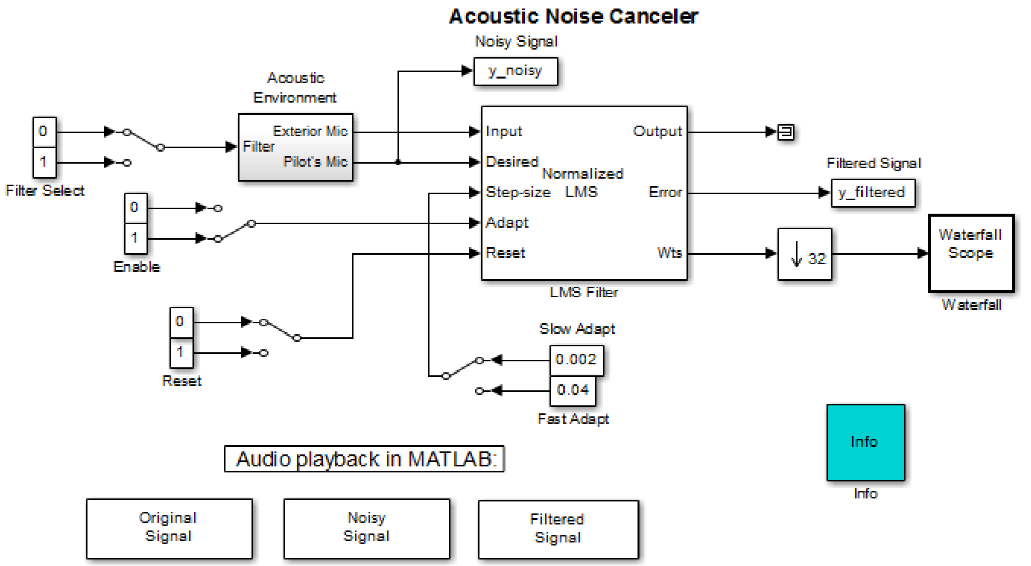
   The Waterfall block no longer captures or displays the data coming from the Downsample block.
3. To continue capturing data, click the **Resume data capture** button.
4. To freeze the data display while continuing to capture data, click the **Snapshot display** button.
5. To view the Waterfall block that the data is coming from, click the **Go to scope block** button.

   In the Simulink model window, the Waterfall block that corresponds to the active Waterfall window flashes. This feature is helpful when you have more than one Waterfall block in a model and you want to clarify which data is being displayed.

Linking Scopes

You can link several Waterfall blocks together in order to capture the effect of a model event in all of the Waterfall windows in the model:

1 Open the `dspanc` example.
2 Drag a second Waterfall block into the example model.
3 Connect this block to the Output port of the LMS Filter block as shown in the figure below.



4 Run the model and view the model behavior in both Waterfall windows.
5 In the dspanc/Waterfall window, click the **Link scopes** button.
6 In the same window, click the **Suspend data capture** button.

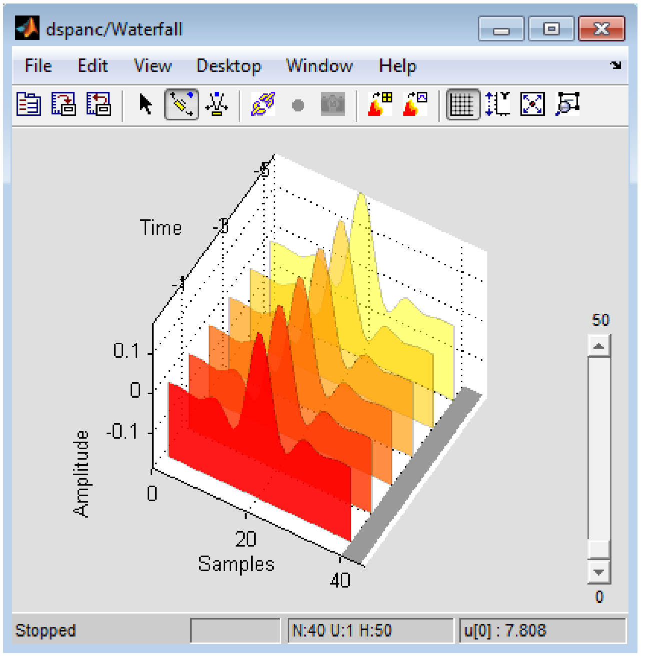
   The data capture is suspended in both scope windows.
7 Click the **Resume data capture** button.

   The data capture resumes in both scope windows.
8 In the dspanc/Waterfall window, click the **Snapshot display** button.

   In both scope windows, the data display freezes while the block continues to capture data.
9 To continue displaying the captured data, click the **Resume display** button.

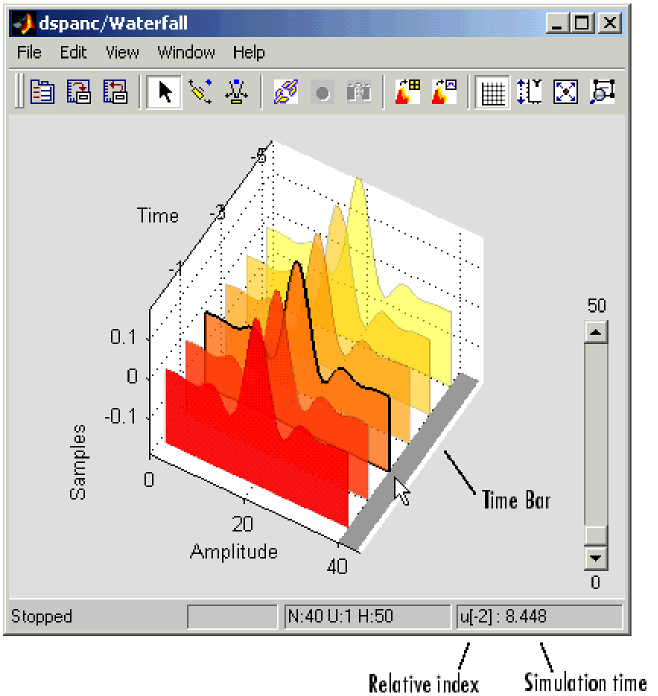
Selecting Data

The following figure shows the Waterfall window displaying the output of the `dspanc` example:



1. To select a particular set of data, click the **Select** button.
2. Click on the Time Bar at the bottom right of the axes to select a vector of data.

   The Waterfall block highlights the selected trace.



While the simulation is running, in the bottom right corner, the Waterfall window displays the relative index of the selected trace. For example, in the previous figure, the selected vector is two sample times away from the most current data vector. When the simulation is stopped, the Waterfall window displays both the relative index and the simulation time associated with the selected trace.

3 To deselect the data vector, click it again.

4 Click-and-drag along the Time Bar.

Your selection follows the movement of the pointer.

You can use this feature to choose a particular vector to export to the MATLAB workspace or SPTool. For more information, see "Data History Parameters" on page 2-1929.

Zooming

You can use the Waterfall window to zoom in on data:

1. Click the **Zoom camera** button.
2. In the Waterfall window, click and hold down the left mouse button.
3. Move the mouse up and down and side-to-side to move closer and farther away from the axes.
4. To resize the axes to fit the Waterfall window, click the **Fit to view** button.

Rotating the Display

You can rotate the data displayed in the Waterfall window:

1. Click on the **Orbit camera** button.
2. In the Waterfall window, click and hold down the left mouse button.
3. Move the mouse in a circular motion to rotate the axes.
4. To return to the position of the original axes, click the **Restore scope position and view** button.

Scaling the Axes

You can use the Waterfall window to rescale the *y*-axis values:

1. Open and run the `dspanc` example.
2. Click the **Rescale amplitude** button.

   The *y*-axis changes so that its minimum value is zero. The maximum value is scaled to fit the data displayed.

Alternatively, you can scale the *y*-axis using the **Y Min** and **Y Max** parameters in the **Axes** pane of the Parameters dialog box. This is helpful when you want to undo the effects of rescaling the amplitude. For more information, see “Axes Parameters” on page 2-1928.

Saving Scope Settings

The Waterfall block can save the screen position and viewpoint of the Waterfall window:

1 Click the **Save scope position and view** button.
2 Close the Waterfall window.
3 Reopen the Waterfall window.

It reopens at the same place on your screen. The viewpoint of the axes also remains the same.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

The Waterfall block accepts any of these data types as input. However, the input is converted to double-precision before the block processes the data. The Waterfall block displays only real-valued, double-precision vectors of data.

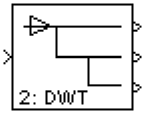
See Also

Scope	Simulink
Time Scope	DSP System Toolbox
Vector Scope	DSP System Toolbox
`Spectrum Analyzer`	DSP System Toolbox
Matrix Viewer	DSP System Toolbox
Triggered To Workspace	DSP System Toolbox

Introduced before R2006a

Wavelet Analysis (Obsolete)

Decompose signal into components of logarithmically decreasing frequency intervals and sample rates (requires the Wavelet Toolbox product)



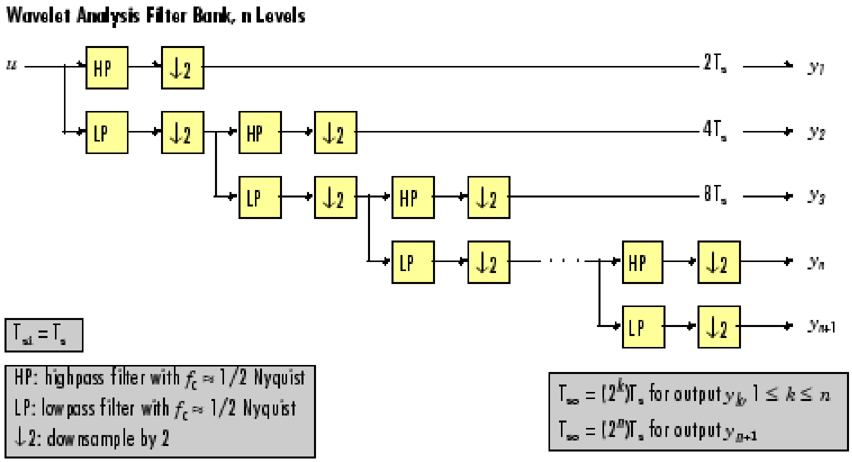
Library

dspobslib

Description

Note: The Wavelet Analysis block will be removed from the product in a future release. We strongly recommend replacing this block with the DWT block.

The Wavelet Analysis block uses the Wavelet Toolbox `wfilters` function to construct a dyadic analysis filter bank that decomposes a broadband signal into a collection of successively more bandlimited components. An *n*-level filter bank structure is shown below, where *n* is specified by the **Number of levels** parameter.



At each level, the low-frequency output of the previous level is decomposed into adjacent high- and low-frequency subbands by a highpass (HP) and lowpass (LP) filter pair. Each of the two output subbands is half the bandwidth of the input to that level. The bandlimited output of each filter is maximally decimated by a factor of 2 to preserve the bit rate of the original signal.

Filter Coefficients

The filter coefficients for the highpass and lowpass filters are computed by the Wavelet Toolbox function `wfilters`, based on the wavelet specified in the **Wavelet name** parameter. The table below lists the available options.

Wavelet Name	Sample Wavelet Function Syntax
Haar	`wfilters('haar')`
Daubechies	`wfilters('db4')`
Symlets	`wfilters('sym3')`
Coiflets	`wfilters('coif1')`
Biorthogonal	`wfilters('bior3.1')`
Reverse Biorthogonal	`wfilters('rbio3.1')`
Discrete Meyer	`wfilters('dmey')`

The **Daubechies**, **Symlets**, and **Coiflets** options enable a secondary **Wavelet order** parameter that allows you to specify the wavelet order. For example, if you specify a **Daubechies** wavelet with **Wavelet order** equal to `6`, the Wavelet Analysis block calls the `wfilters` function with input argument `'db6'`.

The **Biorthogonal** and **Reverse Biorthogonal** options enable a secondary **Filter order [synthesis / analysis]** parameter that allows you to independently specify the wavelet order for the analysis and synthesis filter stages. For example, if you specify a **Biorthogonal** wavelet with **Filter order [synthesis / analysis]** equal to `[2 / 6]`, the Wavelet Analysis block calls the `wfilters` function with input argument `'bior2.6'`.

See the Wavelet Toolbox documentation for more information about the `wfilters` function. If you want to explicitly specify the FIR coefficients for the analysis filter bank, use the `Dyadic Analysis Filter bank` block.

Tree Structure

The wavelet tree structure has n+1 outputs, where n is the number of levels. The sample rate and bandwidth of the top output are half the input sample rate and bandwidth. The sample rate and bandwidth of each additional output (except the last) are half that of the output from the previous level. In general, for an input with sample period $T_{si} = T_s$, and bandwidth BW, output y_k has sample period $T_{so,k}$ and bandwidth BW_k.

$$T_{so,k} = \begin{cases} \left(2^k\right)T_s & (1 \le k \le n) \\ \left(2^n\right)T_s & (k = n+1) \end{cases}$$

$$BW_k = \begin{cases} \dfrac{BW}{2^k} & (1 \le k \le n) \\ \dfrac{BW}{2^n} & (k = n+1) \end{cases}$$

Note that in frame-based mode, the change in the sample period of output y_k is reflected by its frame size $M_{o,k}$, rather than by its frame rate.

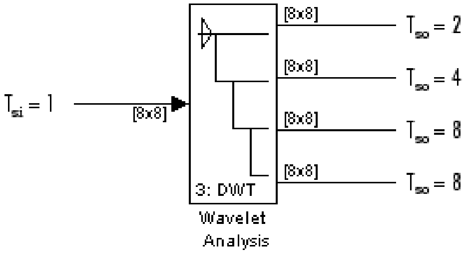
$$M_{o,k} = \begin{cases} \dfrac{M_i}{2^k} & (1 \le k \le n) \\ \dfrac{M_i}{2^n} & (k = n+1) \end{cases}$$

The bottom two outputs (y_n and y_{n+1}) share the same sample period, bandwidth, and frame size because they originate at the same tree level.

Sample-Based Operation

An M-by-N sample-based matrix input is treated as $M*N$ independent channels, and the block filters each channel independently over time. The output at each port is the same size as the input, one output channel for each input channel. As described earlier, each output port has a different sample period.

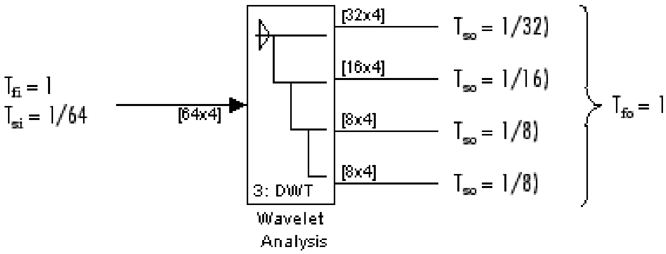
The figure below shows the input and output sample periods for a 64-channel sample-based input to a three-level filter bank. The input has a period of 1, so the fastest output has a period of 2.



Frame-Based Operation

An M_i-by-N frame-based matrix input is treated as N independent channels, and the block filters each channel independently over time. The input frame size M_i must be a multiple of 2^n, and n is the number of filter bank levels. For example, a frame size of 8 would be appropriate for a three-level tree (2^3=8). The number of columns in each output is the same as the number of columns in the input.

Each output port has the same frame period as the input. The reduction in the output sample rates results from the smaller output frame sizes, as shown in the example below for a four-channel input to a three-level filter bank.



Zero Latency

The Wavelet Analysis block has no tasking latency for frame-based operation, which is always single-rate. The block therefore analyzes the first input sample (received at *t*=0) to produce the first output sample at each port.

Nonzero Latency

For sample-based operation, the Wavelet Analysis block is multirate and has 2^{n-1} samples of latency in both Simulink tasking modes. As a result, the block repeats a zero initial condition in each channel for the first 2^{n-1} output samples, before propagating the first analyzed input sample (computed from the input received at *t*=0).

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Time-Based Scheduling and Code Generation" (Simulink Coder).

Parameters

Wavelet name

The wavelet used in the analysis.

Wavelet order

The order for the **Daubechies**, **Symlets**, and **Coiflets** wavelets. This parameter is available only when one of these wavelets is selected in the **Wavelet name** menu.

Filter order [synthesis / analysis]

The filter orders for the synthesis and analysis stages of the **Biorthogonal** and **Reverse Biorthogonal** wavelets. For example, `[2 / 6]` selects a second-order synthesis stage and a sixth-order analysis stage. The **Filter order** parameter is available only when one of the above wavelets is selected in the **Wavelet name** menu.

Number of levels

The number of filter bank levels. An *n*-level structure has *n*+1 outputs.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types

- Double-precision floating point

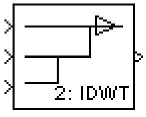
See Also

`Dyadic Analysis Filter bank`	DSP System Toolbox
`wfilters`	Wavelet Toolbox

Introduced in R2008b

Wavelet Synthesis (Obsolete)

Reconstruct signal from its multirate bandlimited components (requires the Wavelet Toolbox product)



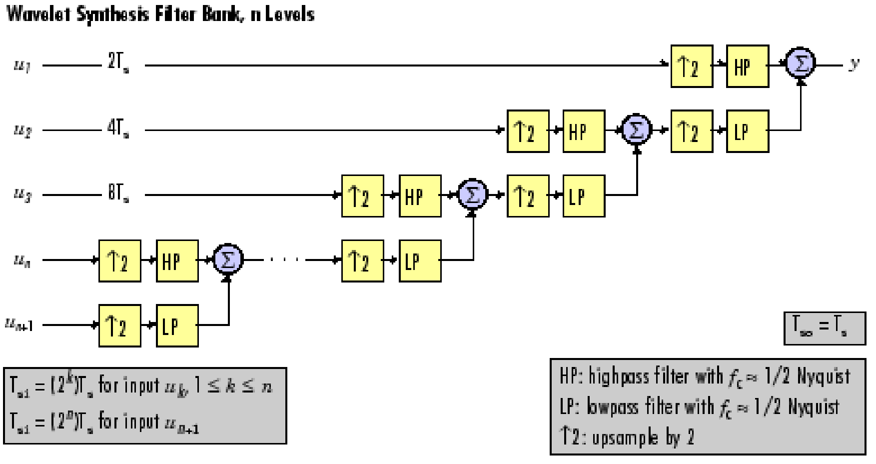
Library

`dspobslib`

Description

Note: The Wavelet Synthesis block will be removed from the product in a future release. We strongly recommend replacing this block with the `IDWT` block.

The Wavelet Synthesis block uses the Wavelet Toolbox `wfilters` function to reconstruct a signal that was decomposed by the Wavelet Analysis (Obsolete) block. The reconstruction or synthesis process is the inverse of the analysis process, and restores the original signal by upsampling, filtering, and summing the bandlimited inputs in stages corresponding to the analysis process. An n-level synthesis filter bank structure is shown below, where n is specified by the **Number of levels** parameter.



At each level, the two bandlimited inputs (one low-frequency, one high-frequency, both with the same sample rate) are upsampled by a factor of 2 to match the sample rate of the input to the next stage. They are then filtered by a highpass (HP) and lowpass (LP) filter pair with coefficients calculated to cancel (in the subsequent summation) the aliasing introduced in the corresponding analysis filter stage. The output from each (upsample-filter-sum) level has twice the bandwidth and twice the sample rate of the input to that level.

For perfect reconstruction, the Wavelet Synthesis and Wavelet Analysis blocks must have the same parameter settings.

Filter Coefficients

The filter coefficients for the highpass and lowpass filters are computed by the Wavelet Toolbox function `wfilters`, based on the wavelet specified in the **Wavelet name** parameter. The table below lists the available options.

Wavelet Name	Sample Wavelet Function Syntax
Haar	`wfilters('haar')`
Daubechies	`wfilters('db4')`
Symlets	`wfilters('sym3')`
Coiflets	`wfilters('coif1')`
Biorthogonal	`wfilters('bior3.1')`

Wavelet Name	Sample Wavelet Function Syntax
Reverse Biorthogonal	`wfilters('rbio3.1')`
Discrete Meyer	`wfilters('dmey')`

The **Daubechies**, **Symlets**, and **Coiflets** options enable a secondary **Wavelet order** parameter that allows you to specify the wavelet order. For example, if you specify a **Daubechies** wavelet with **Wavelet order** equal to `6`, the Wavelet Synthesis block calls the `wfilters` function with input argument `'db6'`.

The **Biorthogonal** and **Reverse Biorthogonal** options enable a secondary **Filter order [synthesis / analysis]** parameter that allows you to independently specify the wavelet order for the analysis and synthesis filter stages. For example, if you specify a **Biorthogonal** wavelet with **Filter order [synthesis / analysis]** equal to `[2 / 6]`, the Wavelet Synthesis block calls the `wfilters` function with input argument `'bior2.6'`.

See the Wavelet Toolbox documentation for more information about the `wfilters` function. If you want to explicitly specify the FIR coefficients for the synthesis filter bank, use the `Dyadic Synthesis Filter bank` block.

Tree Structure

The wavelet tree structure has n+1 inputs, where n is the number of levels. The sample rate and bandwidth of the output are twice the sample rate and bandwidth of the top input. The sample rate and bandwidth of each additional input (except the last) are half that of the input to the previous level.

$$T_{si,k+1} = 2T_{si,k} \quad 1 \le k < n$$

$$BW_{k+1} = \frac{BW_k}{2} \quad 1 \le k < n$$

The bottom two inputs (u_n and u_{n+1}) should have the same sample rate and bandwidth since they are processed by the same level.

$$T_{si,n+1} = T_{si,n}$$

$$BW_{n+1} = BW_n$$

Note that in frame-based mode, the sample period of input u_k is reflected by its frame size$M_{i,k}$, rather than by its frame rate.

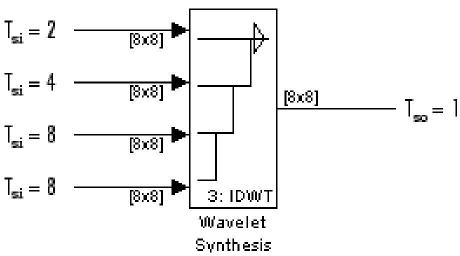
$$M_{i,k+1} = \frac{M_{i,k}}{2} \qquad 1 \le k < n$$

$$M_{i,n+1} = M_{i,n}$$

Sample-Based Operation

An M-by-N sample-based matrix input is treated as $M*N$ independent channels, and the block filters each channel independently over time. The output is the same size as the input at each port, one output channel for each input channel. As described earlier, each input port has a different sample period.

The figure below shows the input and output sample periods for the four 64-channel sample-based inputs to a three-level filter bank. The fastest input has a period of 2, so the output period is 1.

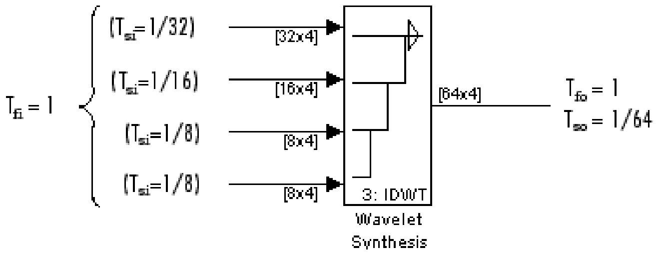


Frame-Based Operation

An M_i-by-N frame-based matrix input is treated as N independent channels, and the block filters each channel independently over time. The number of columns in the output is the same as the number of columns in the input.

All inputs must have the same frame period, which is also the output frame period. The different input sample rates should be represented by the input frame sizes: If the input to the top port has frame size M_i, the input to the second-from-top port should have frame size $M_i/2$, the input to the third-from-top port should have frame size $M_i/4$, and so on. The input to the bottom port should have the same frame size as the second-from-bottom port. The increase in the sample rate of the output is also represented by its frame size, which is twice the largest input frame size.

The relationship between sample periods, frame periods, and frame sizes is shown below for a four-channel frame-based input to a 3-level filter bank.



Zero Latency

The Wavelet Synthesis block has no tasking latency for frame-based operation, which is always single-rate. The block therefore uses the first input samples (received at t=0) to synthesize the first output sample.

Nonzero Latency

For sample-based operation, the Wavelet Synthesis block is multirate and has the following tasking latencies:

- 2^n-2 samples in Simulink's single-tasking mode
- 2^n samples in Simulink's multitasking mode

In the above cases, the block repeats a zero initial condition in each channel for the first D output samples, where D is the latency shown above. For example, in single-tasking mode the block generates 2^n-2 zero-valued output samples in each channel before propagating the first synthesized output sample (computed from the inputs received at t=0).

Note For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” and “Time-Based Scheduling and Code Generation” (Simulink Coder).

Parameters

Wavelet name

The wavelet used in the synthesis.

Wavelet order

The order for the **Daubechies**, **Symlets**, and **Coiflets** wavelets. This parameter is available only when one of these wavelets is selected in the **Wavelet name** menu.

Filter order [synthesis / analysis]

The filter orders for the synthesis and analysis stages of the **Biorthogonal** and **Reverse Biorthogonal** wavelets. For example, `[2 / 6]` selects a second-order synthesis stage and a sixth-order analysis stage. The **Filter order** parameter is available only when one of the above wavelets is selected in the **Wavelet name** menu.

Number of levels

The number of filter bank levels. An n-level structure has n+1 outputs.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types

- Double-precision floating point

See Also

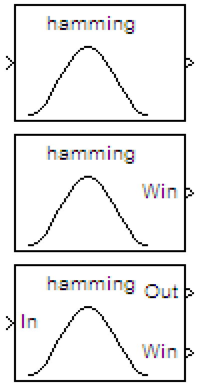
`Dyadic Synthesis Filter bank`	DSP System Toolbox

`wfilters`	Wavelet Toolbox

Introduced in R2008b

Window Function

Compute and/or apply window to input signal



Library

Signal Operations

`dspsigops`

Description

The Window Function block computes a window and/or applies a window to an input signal. The input signal can be a matrix or an N-D array.

Operation Modes

The Window Function block has three modes of operation that you can select via the **Operation** parameter. In each mode, the block first creates a window vector w by sampling the window specified in the **Window type** parameter at M discrete points. The operation modes are:

- `Apply window to input`

In this mode, the block computes an M-by-1 window vector w and applies it to the input. The output y always has the same dimension as the input.

When the input is an M-by-N matrix u, the window is multiplied element-wise with each of the N channels in the input matrix u. This is equivalent to the following MATLAB code:

```
y = repmat(w,1,N) .* u        % Equivalent MATLAB code
```

The window is always applied to the first dimension:

$$y(i,j,\ldots,k) = w(i) * u(i,j,\ldots,k) \qquad i = 1,\ldots,M, \;\; j = 1,\ldots,N, \;\; \ldots, \;\; k = 1,\ldots,P$$

A length-M unoriented vector input is treated as an M-by-1 matrix.

- `Generate window`

  In this mode, the block generates an unoriented window vector w with length M specified by the **Window length** parameter. The In port is disabled for this mode.

- `Generate and apply window`

  In this mode, the block generates an M-by-1 window vector w and applies it to the input. The block produces two outputs:

  - At the Out port, the block produces the result of the multiplication y, which has the same dimension as the input.
  - At the Win port, the block produces the M-by-1 window vector w.

  When the input is an M-by-N matrix u, the window is multiplied element-wise with each of the N channels in the input matrix u. This is equivalent to the following MATLAB code:

```
y = repmat(w,1,N) .* u        % Equivalent MATLAB code
```

The window is always applied to the first dimension:

$$y(i,j,\ldots,k) = w(i) * u(i,j,\ldots,k) \qquad i = 1,\ldots,M, \;\; j = 1,\ldots,N, \;\; \ldots, \;\; k = 1,\ldots,P$$

A length-M 1-D vector input is treated as an M-by-1 matrix.

Window Type

The following table lists the available window types. For complete information about the window functions, consult the Signal Processing Toolbox documentation.

Window Type	Description
Bartlett	Computes a Bartlett window. `w = bartlett(M)`
Blackman	Computes a Blackman window. `w = blackman(M)`
Boxcar	Computes a rectangular window. `w = rectwin(M)`
Chebyshev	Computes a Chebyshev window with stopband ripple `R`. `w = chebwin(M,R)`
Hamming	Computes a Hamming window. `w = hamming(M)`
Hann	Computes a Hann window (also known as a Hanning window). `w = hann(M)`
Hanning	Obsolete. This window type is included only for compatibility with older models. Use the `Hann` **Window type** instead of `Hanning` whenever possible.
Kaiser	Computes a Kaiser window with the Kaiser parameter `beta`. `w = kaiser(M,beta)`
Taylor	Computes a Taylor window. `w = taylorwin(M)`
Triang	Computes a triangular window. `w = triang(M)`
User Defined	Computes the user-defined window function specified by the entry in the **Window function name** parameter, `usrwin`.

Window Type	Description
	`w = usrwin(M) % Window takes no extra parameters` `w = usrwin(M,x1,...,xn) % Window takes extra parameters {x1 ... xn}`

Window Sampling

For the generalized-cosine windows (`Blackman`, `Hamming`, `Hann`, and `Hanning`), the **Sampling** parameter determines whether the window samples are computed in a periodic or a symmetric manner. For example, when **Sampling** is set to `Symmetric`, a Hamming window of length M is computed as

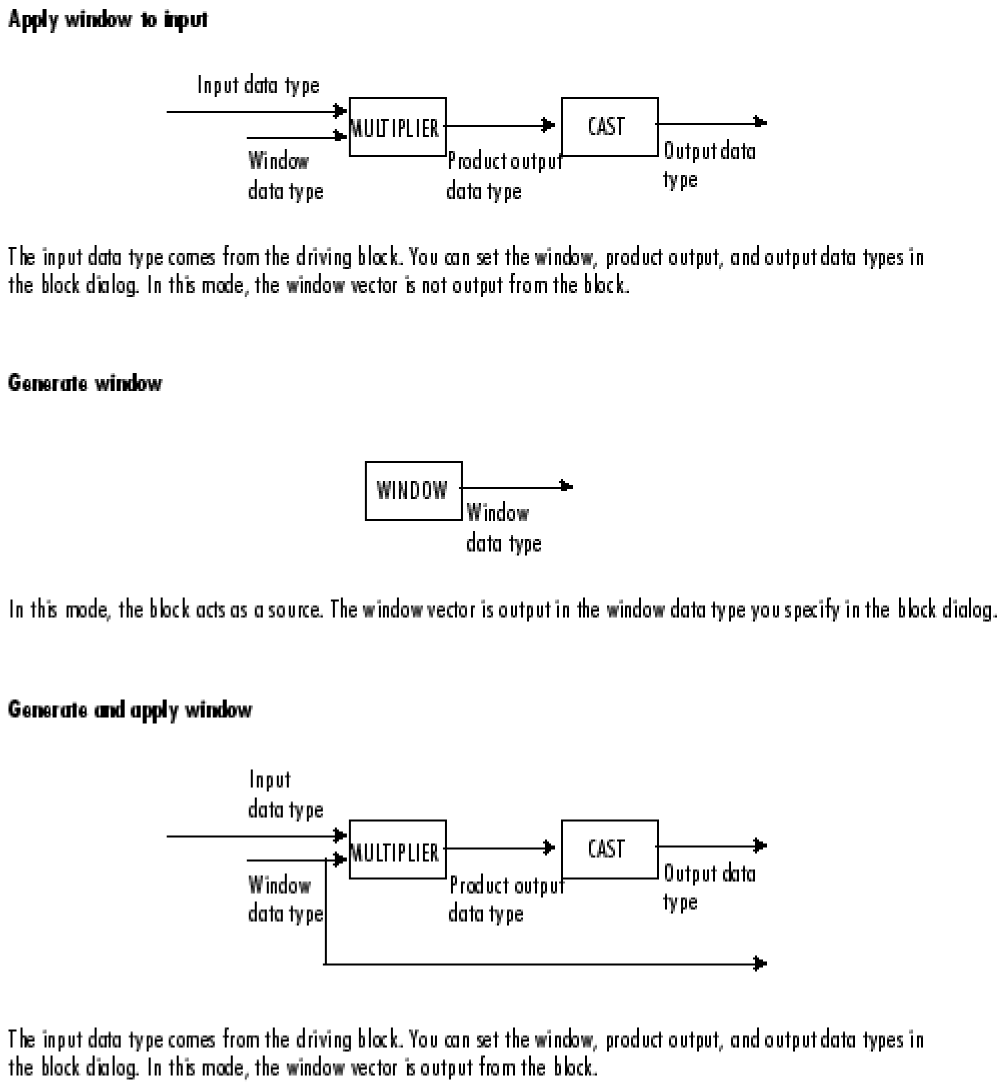
```
w = hamming(M)     % Symmetric (aperiodic) window
```

When **Sampling** is set to `Periodic`, the same window is computed as

```
w = hamming(M+1)     % Periodic (asymmetric) window
w = w(1:M)
```

Fixed-Point Data Types

The following diagram shows the data types used within the Window Function block for fixed-point signals for each of the three operating modes.



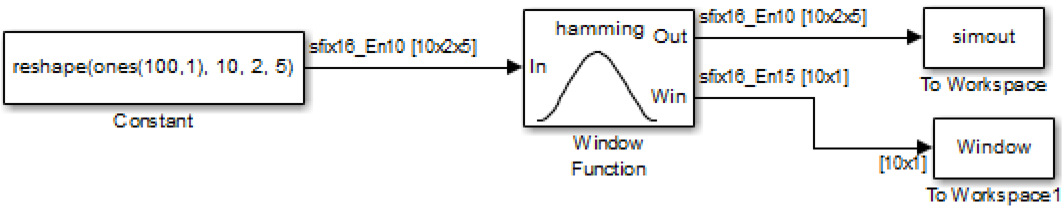
You can set the window, product output, and output data types in the block dialog box. For more information see the “Dialog Box” on page 2-1963 section.

Examples

The following model uses the Window Function block to generate and apply a Hamming window to a 3-dimensional input array.

In this example, set the **Operation** mode of the Window Function block to `Generate and apply window`, so the block provides two outputs: the window vector *Window* at the Win port, and the result of the multiplication *simout* at the Out port.

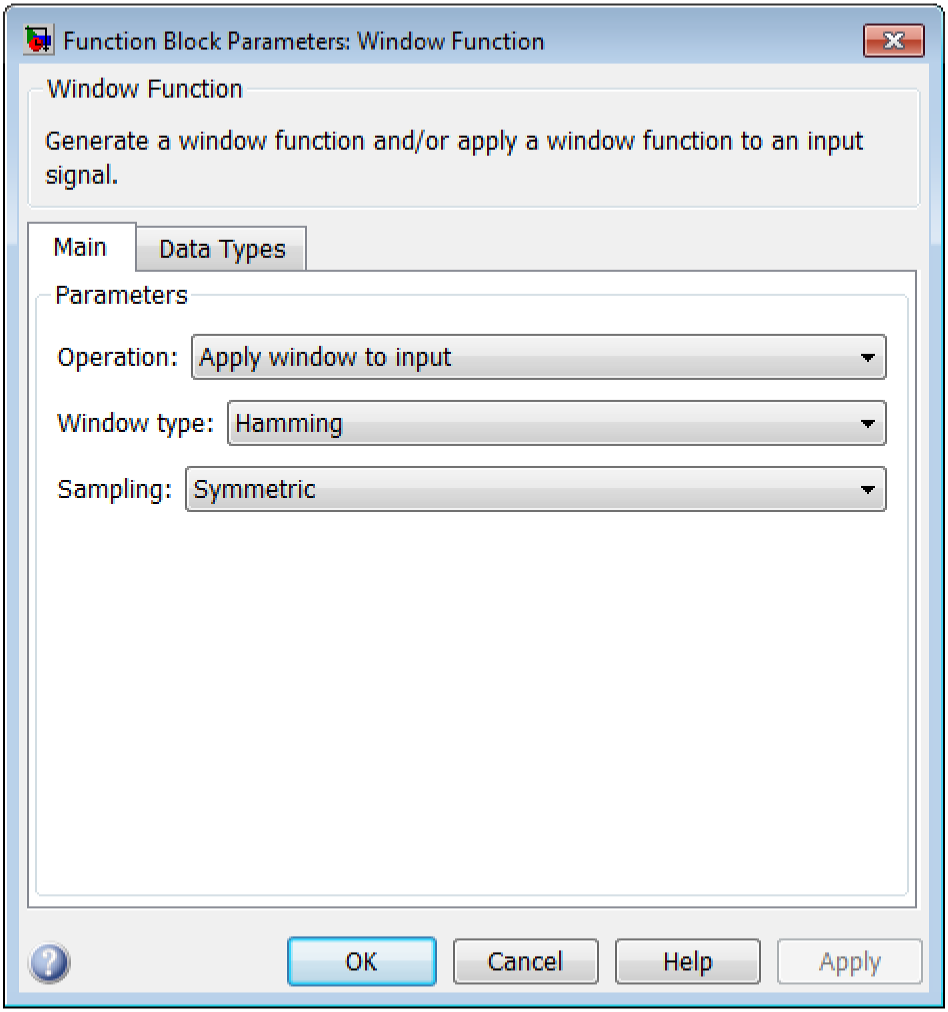
Open the model by typing `ex_windowfunction_ref` at the MATLAB command line, and run it.



- The length of the first dimension of the input array is 10, so the Window Function block generates and outputs a Hamming window vector of length 10. To see the window vector generated by the Window Function block, type `Window` at the MATLAB command line.
- To see the result of the multiplication, type `simout` at the MATLAB command line.

Dialog Box

The **Main** pane of the Window Function block dialog appears as follows.



Operation

Specify the block's operation, as discussed in "Operation Modes" on page 2-1958. The port configuration of the block is updated to match the setting of this parameter.

Window type

Specify the window type to apply, as listed in "Window Type" on page 2-1960. Tunable (Simulink) in simulation only.

Sampling

Specify the window sampling for generalized-cosine windows. This parameter is only visible when you select `Blackman`, `Hamming`, `Hann`, or `Hanning` for the **Window type** parameter. Tunable (Simulink) in simulation only.

Sample Mode

Specify the sample mode for the block, `Continuous` or `Discrete`, when it is in `Generate Window` mode. In the `Apply window to output` and `Generate and apply window` modes, the block inherits the sample time from its driving block. Therefore, this parameter is only visible when you select `Generate window` for the **Operation** parameter.

Sample time

Specify the sample time for the block when it is in `Generate window` and `Discrete` modes. In `Apply window to output` and `Generate and apply window` modes, the block inherits the sample time from its driving block. This parameter is only visible when you select `Discrete` for the **Sample Mode** parameter.

Window length

Specify the length of the window to apply. This parameter is only visible when you select `Generate window` for the **Operation** parameter. Otherwise, the window vector length is computed to match the length of the first dimension of the input.

Stopband attenuation in dB

Specify the level of stopband attenuation, R_s, in decibels. This parameter is only visible when you select `Chebyshev` for the **Window type** parameter. Tunable (Simulink) in simulation only.

Beta

Specify the `Kaiser` window β parameter. Increasing β widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. This parameter is only visible when you select `Kaiser` for the **Window type** parameter. Tunable (Simulink) in simulation only.

Number of sidelobes

Specify the number of sidelobes as a scalar integer value greater than zero. This parameter is only visible when you select `Taylor` for the **Window type** parameter.

Maximum sidelobe level relative to mainlobe (dB)

Specify, in decibels, the maximum sidelobe level relative to the mainlobe. This parameter must be a scalar less than or equal to zero. The default value of –30 produces sidelobes with peaks 30 dB down from the mainlobe peak. This parameter is only visible when you select `Taylor` for the **Window type** parameter.

Window function name

Specify the name of the user-defined window function to be calculated by the block. This parameter is only visible when you select `User defined` for the **Window type** parameter.

Specify additional arguments to the hamming function

Select to enable the **Cell array of additional arguments** parameter, when the user-defined window requires parameters other than the window length. This parameter is only visible when you select `User defined` for the **Window type** parameter.

Cell array of additional arguments

Specify the extra parameters required by the user-defined window function, besides the window length. This parameter is only available when you select the **Specify additional arguments to the hamming function** parameter. The entry must be a cell array.

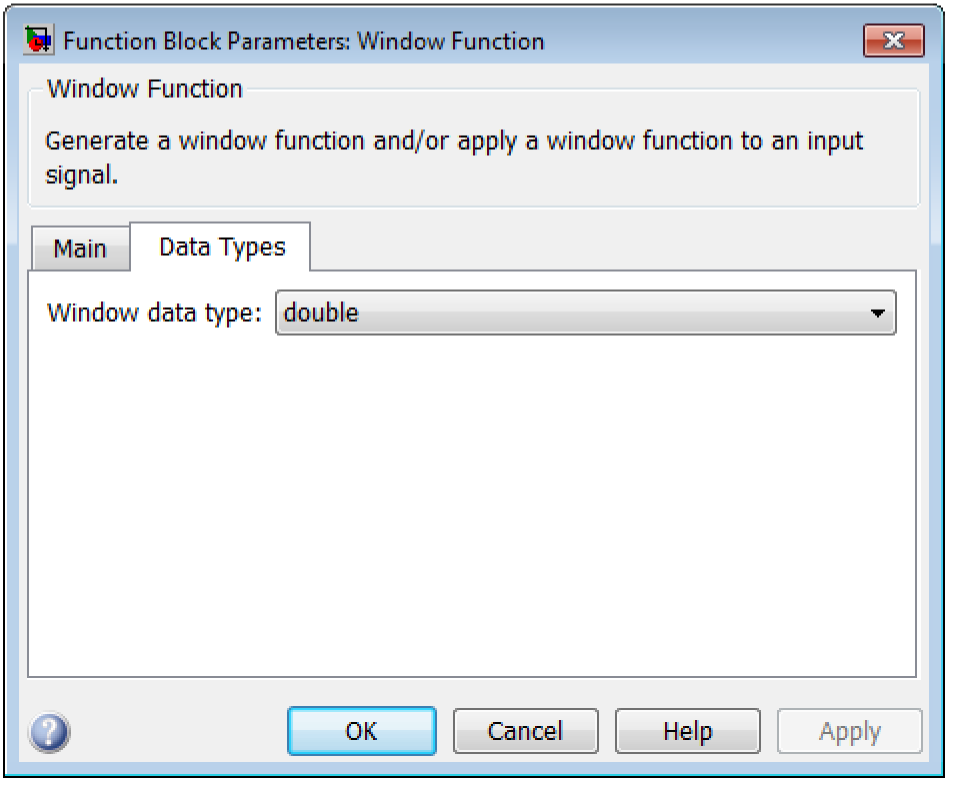
The **Data Types** pane of the Window Function block dialog is discussed in the following sections:

“Parameters for Generate Window Only Mode” on page 2-1966

“Parameters for Apply Window Modes” on page 2-1968

Parameters for Generate Window Only Mode

The **Data Types** pane of the Window Function block dialog appears as follows when the **Operation** parameter is set to `Generate window`.



Window data type

Specify the window data type in one of the following ways:

- Choose `double` or `single` from the list.
- Choose `Fixed-point` to specify the window data type and scaling in the **Signed**, **Word length**, **Set fraction length in output to**, and **Fraction length** parameters.
- Choose `User-defined` to specify the window data type and scaling in the **User-defined data type**, **Set fraction length in output to**, and **Fraction length** parameters.
- Choose `Inherit via back propagation` to set the window data type and scaling to match the following block.

Signed

Select to output a signed fixed-point signal. Otherwise, the signal is unsigned.

Word length

Specify the word length, in bits, of the fixed-point window data type. This parameter is only visible when you select `Fixed-point` for the **Window data type** parameter.

User-defined data type

Specify any built-in or fixed-point data type. You can specify fixed-point data types using the Fixed-Point Designer functions `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac`. This parameter is only visible when you select `User-defined` for the **Window data type** parameter.

Set fraction length in output to

Specify the scaling of the fixed-point window data type by either of the following two methods:

- Choose `Best precision` to have the window data type scaling automatically set such that the output signal has the best possible precision.
- Choose `User-defined` to specify the window data type scaling in the **Fraction length** parameter.

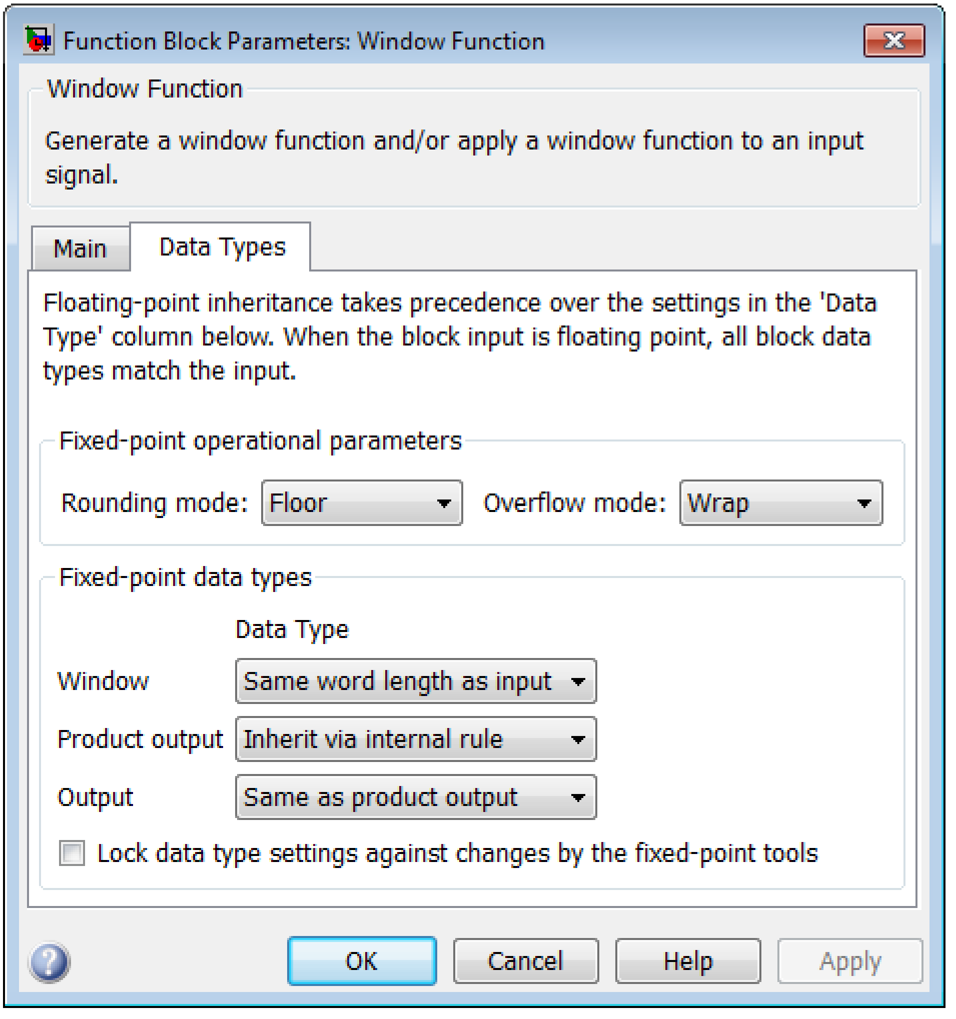
This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Window data type** parameter, and when the specified window data type is a fixed-point data type.

Fraction length

Specify the fraction length, in bits, of the fixed-point window data type. This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Window data type** parameter and `User-defined` for the **Set fraction length in output to** parameter.

Parameters for Apply Window Modes

The **Data Types** pane of the Window Function block dialog appears as follows when the **Operation** parameter is set to either `Apply window to input` or `Generate and apply window`.



Rounding mode

Select the rounding mode for fixed-point operations.

The window vector w does not obey this parameter; it always rounds to `Nearest`.

Note: The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when both of the following conditions exist:

- **Product output data type** is `Inherit via internal rule`
- **Output data type** is `Same as product output`

With these data type settings, the block is effectively operating in full precision mode.

Overflow mode

Select the overflow mode for fixed-point operations.

The window vector *w* does not obey this parameter; it is always saturated.

Window

Choose how you specify the word length and fraction length of the window vector *w*.

When you select `Same word length as input`, the word length of the window vector elements is the same as the word length of the input. The fraction length is automatically set to the best precision possible.

When you select `Specify word length`, you can enter the word length of the window vector elements in bits. The fraction length is automatically set to the best precision possible.

When you select `Binary point scaling`, you can enter the word length and the fraction length of the window vector elements in bits.

When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the window vector elements. This block requires power-of-two slope and a bias of zero.

The window vector does not obey the **Rounding mode** and **Overflow mode** parameters; it is always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you want to designate the product output word and fraction lengths.

- When you select `Inherit via internal rule`, the product output word length and fraction length are calculated automatically. For information about how the product output word and fraction lengths are calculated when an internal rule is used, see “Inherit via Internal Rule”.
- When you select `Same as input`, these characteristics match those of the input to the block.

- When you select `Binary point scaling`, you can enter the word length and the fraction length of the product output, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Output

Choose how you specify the word length and fraction length of the output of the block:

- When you select `Same as product output`, these characteristics match those of the product output.
- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of the output, in bits.
- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
Win	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit integers

See Also

See Also

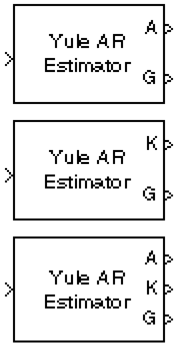
Functions
`bartlett` | `blackman` | `chebwin` | `hamming` | `hann` | `kaiser` | `rectwin` | `taylorwin` | `triang`

Blocks
`FFT`

Introduced before R2006a

Yule-Walker AR Estimator

Compute estimate of autoregressive (AR) model parameters using Yule-Walker method



Library

Estimation / Parametric Estimation

`dspparest3`

Description

The Yule-Walker AR Estimator block uses the Yule-Walker AR method, also called the autocorrelation method, to fit an autoregressive (AR) model to the windowed input data by minimizing the forward prediction error in the least squares sense. This formulation leads to the Yule-Walker equations, which are solved by the Levinson-Durbin recursion. Block outputs are always nonsingular.

The Yule-Walker AR Estimator block can output the AR model coefficients as polynomial coefficients, reflection coefficients, or both. The input can be a row vector, a column vector, or an unoriented vector which is assumed to be the output of an AR system driven by white noise. The block accepts matrices, and treats each column of the matrix as a channel. If the input is a row vector of length N, the input is treated as N different channels. If the input is an unoriented vector, the input is treated as a single channel. The block computes the normalized estimate of the AR system parameters, $A(z)$, independently for each successive input frame.

$$H(z) = \frac{\sqrt{G}}{A(z)} = \frac{\sqrt{G}}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

When you select **Inherit estimation order from input dimensions**, the order *p* of the all-pole model is one less than the length of each input channel. Otherwise, the order is the value specified by the **Estimation order** parameter. To guarantee a valid output, you must set the **Estimation order** parameter to be a scalar less than or equal to half the input channel length. The Yule-Walker AR Estimator and Burg AR Estimator blocks return similar results for large frame sizes.

When **Output(s)** is set to `A`, port A is enabled. For each channel, port A outputs a column of length *p*+1 that contains the normalized estimate of the AR model coefficients in descending powers of *z*

```
[1 a(2) ... a(p+1)]
```

When **Output(s)** is set to `K`, port K is enabled. For each channel, port K outputs a length-*p* column whose elements are the AR model reflection coefficients. When **Output(s)** is set to `A and K`, both port A and K are enabled, and each port outputs the respective AR model coefficients for each channel.

The square of the model gain, *G*, is provided at port G. *G* is a scalar for each channel.

See the Burg AR Estimator block reference page for a comparison of the Burg AR Estimator, Covariance AR Estimator, Modified Covariance AR Estimator, and Yule-Walker AR Estimator blocks.

Parameters

Output(s)

The type of AR model coefficients output by the block. The block can output polynomial coefficients (`A`), reflection coefficients (`K`), or both (`A and K`).

Inherit estimation order from input dimensions

When selected, sets the estimation order *p* to one less than the length of each input channel.

Estimation order

The order of the AR model, *p*. This parameter is enabled when you do not select **Inherit estimation order from input dimensions**.

References

Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
A	• Double-precision floating point • Single-precision floating point
K	• Double-precision floating point • Single-precision floating point
G	• Double-precision floating point • Single-precision floating point

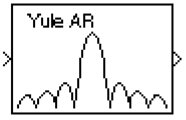
See Also

`Burg AR Estimator`	DSP System Toolbox
`Covariance AR Estimator`	DSP System Toolbox
`Modified Covariance AR Estimator`	DSP System Toolbox
`Yule-Walker Method`	DSP System Toolbox
`aryule`	Signal Processing Toolbox

Introduced before R2006a

Yule-Walker Method

Power spectral density estimate using Yule-Walker method



Library

Estimation / Power Spectrum Estimation

`dspspect3`

Description

The Yule-Walker Method block estimates the power spectral density (PSD) of the input using the Yule-Walker AR method. This method, also called the *autocorrelation method*, fits an autoregressive (AR) model to the windowed input data. It does so by minimizing the forward prediction error in the least squares sense. This formulation leads to the Yule-Walker equations, which the Levinson-Durbin recursion solves. Block outputs are always nonsingular.

The input must be a column vector. This input represents a frame of consecutive time samples from a single-channel signal. The block outputs a column vector containing the estimate of the power spectral density of the signal at N_{fft} equally spaced frequency points. The frequency points are in the range $[0,F_s)$, where F_s is the sampling frequency of the signal.

When you select **Inherit estimation order from input dimensions**, the order of the all-pole model is one less that the input frame size. Otherwise, the **Estimation order** parameter value specifies the order. To guarantee a valid output, the **Estimation order** parameter must be less than or equal to half the input vector length. The block computes the spectrum from the FFT of the estimated AR model parameters.

Selecting the **Inherit FFT length from estimation order** parameter specifies that N_{fft} is one greater than the estimation order. Clearing the **Inherit FFT length from**

estimation order check box allows you to use the **FFT length** parameter to specify N_{fft} as a power of 2. The block zero-pads or wraps the input to N_{fft} before computing the FFT.

When you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must hold:

- The input to the block is the original signal, with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

See the Burg Method block reference for a comparison of the Burg Method, Covariance Method, Modified Covariance Method, and Yule-Walker AR Estimator blocks. The Yule-Walker AR Estimator and Burg Method blocks return similar results for large buffer lengths.

Parameters

Inherit estimation order from input dimensions

When you select this option, it sets the estimation order to one less than the length of the input vector.

Estimation order

Specify the order of the AR model. This parameter is only visible when you clear the **Inherit estimation order from input dimensions** check box.

Inherit FFT length from estimation order

When you select the **Inherit FFT length from estimation order** check box, the FFT length is one greater than the estimation order. To specify the number of points on which to perform the FFT, clear the **Inherit FFT length from estimation order** check box. You can then specify a power-of-two FFT length using the **FFT length** parameter.

FFT length

Enter the number of data points on which to perform the FFT, N_{fft}. When N_{fft} is larger than the input frame size, the block zero-pads each frame as needed. When

N_{fft} is smaller than the input frame size, the block wraps each frame as needed. This parameter becomes visible only when you clear the **Inherit FFT length from input dimensions** check box.

Inherit sample time from input

When you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must hold:

- The input to the block is the original signal, with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

Sample time of original time series

Specify the sample time of the original time-domain signal. This parameter becomes visible only when you clear the **Inherit sample time from input** check box.

References

Kay, S. M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L. Jr., *Digital Spectral Analysis with Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

Orfanidis, S. J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
Output	• Double-precision floating point • Single-precision floating point

The output data type is the same as the input data type.

See Also

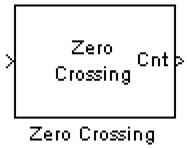
`Burg Method`	DSP System Toolbox
`Covariance Method`	DSP System Toolbox
`Levinson-Durbin`	DSP System Toolbox
`Autocorrelation LPC`	DSP System Toolbox
`Short-Time FFT`	DSP System Toolbox
`Yule-Walker AR Estimator`	DSP System Toolbox

See “Spectral Analysis”for related information.

Introduced before R2006a

Zero Crossing

Count number of times signal crosses zero in single time step



Library

Signal Operations

`dspsigops`

Description

The Zero Crossing block concludes that a signal in a given channel has passed through zero if it meets any of the following criteria, where x_i is the current signal value, x_{i-1} is the previous signal value, and so on:

- $x_i < 0$ and $x_{i-1} > 0$
- $x_i > 0$ and $x_{i-1} < 0$
- For some positive integer L, $x_i < 0$, $x_{i-l} = 0$, and $x_{i-L-1} > 0$, where $0 \le l \le L$.
- For some positive integer L, $x_i > 0$, $x_{i-l} = 0$, and $x_{i-L-1} < 0$, where $0 \le l \le L$.

For the first input value, x_{i-1} and x_{i-2} are zero. The block outputs the number of times the signal crosses zero in a single time step at the Cnt port.

The input to this block must be a real-valued fixed-point or floating-point signal. If you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats each element of the input as a time-varying channel. If you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each column of the input as an independent channel.

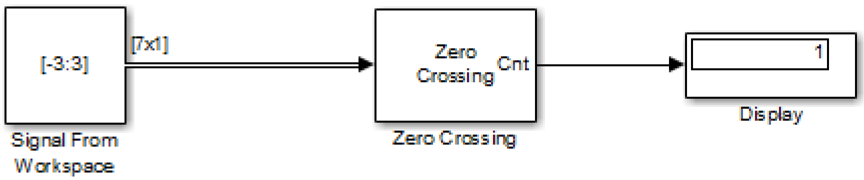
Examples

The following example, ex_zero_crossing illustrates the behavior of the Zero Crossing block.

To run the model for one time step, the **Stop time** is set to 0.

1 Run the model.

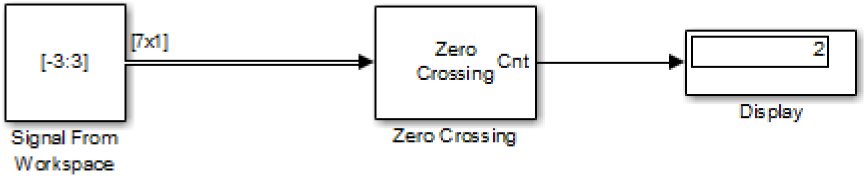
  Because the signal passes through zero once during the first time step, the Zero Crossing block finds one zero crossing. The number of detected zero crossings in the first time step is shown in the Display block in the following figure.



2 To run the model for two time steps, change the simulation **Stop time** to 1. To do so, open the Configuration Parameters dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. In the **Solver** pane, set **Stop time** to 1.

3 Run the model.

  The Zero Crossing block remembers that the last value of the last frame was 3. Therefore, the signal passes through zero twice during the second time step. It passes through zero while going from 3 to -3, and it passes through zero again while going from -3 to 3. The Zero Crossing block finds two zero crossings in the second time step as shown in the Display block in the following figure.



Parameters

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` (default) — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating-point • Single-precision floating-point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Cnt	• 32-bit unsigned integers

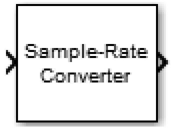
See Also

Hit Crossing   Simulink

Introduced before R2006a

Sample-Rate Converter

Multistage sample-rate conversion



Library

Signal Operations

`dspsigops`

Description

The Sample-Rate Converter block implements a multistage FIR sample-rate converter. This multistage FIR converter converts the rate of each channel of the input signal from the input sample rate to the output sample rate. Multistage implementations minimize the amount of computation required by the sample-rate conversions by first reducing the sample rate of the input signal. Next, the block determines the optimal number of decimators and interpolators required based on the parameters specified in the block dialog box. Then the block designs filters in the individual stages accordingly.

The input frame size must be a multiple of the decimation factor of the rate converter. The decimation factor depends on the parameter setting of the converter. To determine the decimation factor, in the block dialog box, click **View Info** .

Each column of a two-dimensional input signal is treated as a separate channel. If the input is a two-dimensional signal, the first dimension represents the channel length (or frame size), and the second dimension represents the number of channels. If the input is a one-dimensional signal, then it is interpreted as a single channel. The inputs to the block can be single or double, and real or complex.

Parameters

Sample rate of input signal (Hz)

Sample rate of the input signal, specified as a positive scalar in Hz. The input sample rate must be greater than the bandwidth of interest. The default is 48e3.

Sample rate of output signal (Hz)

Sample rate of the output signal, specified as a positive scalar in Hz. The output sample rate must be greater than the bandwidth of interest. The default is 96e3.

Tolerance for output sample rate

Maximum allowed tolerance for output sample rate, specified as a positive scalar in the range [0,1]. The default is 0.

The actual output sample rate varies but is within the specified range. For example, suppose that you set the **Tolerance for output sample rate**, to 0.01. Then the actual output sample rate is in the range given by sample rate of output signal ± 1%. This flexibility allows for a simpler filter design.

Two-sided bandwidth of interest (Hz)

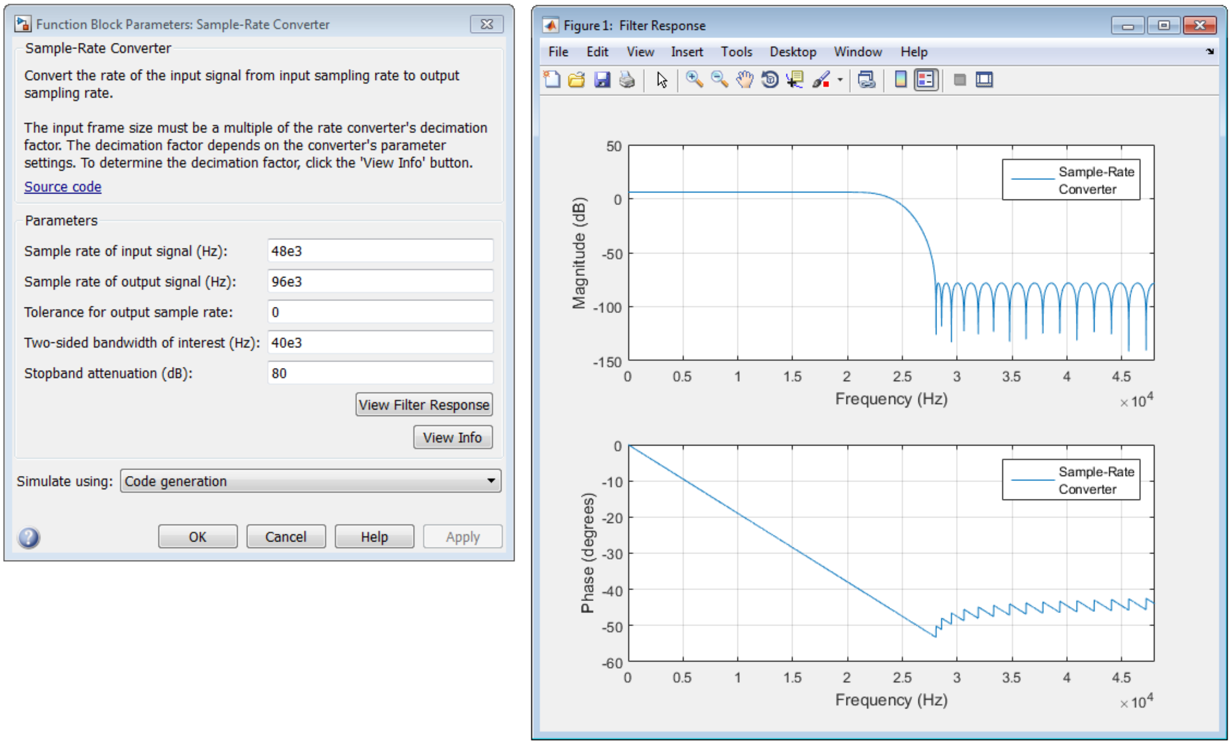
Two-sided bandwidth of interest (after the rate of conversion), specified as a positive scalar in Hz. The default is 40e3.

Stopband attenuation (dB)

Minimum amount of attenuation for aliased components in the stopband, specified as a positive scalar in dB. The default is 80. This parameter is the minimum amount by which any aliasing involved in the process is attenuated.

View Filter Response

Opens the Filter Visualization Tool FVTool and displays the magnitude/phase response of the Sample-Rate Converter. The response is based on the block dialog box parameters. Changes made to these parameters update FVTool.

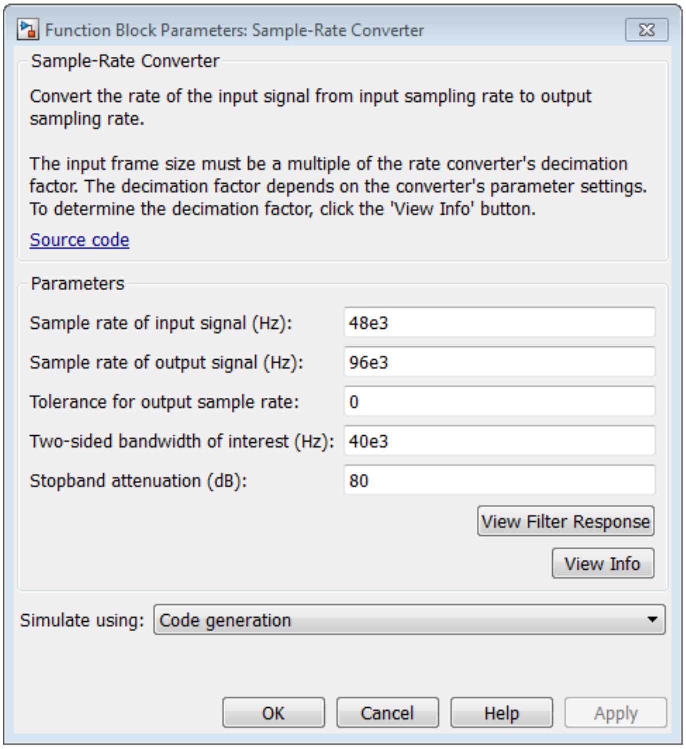


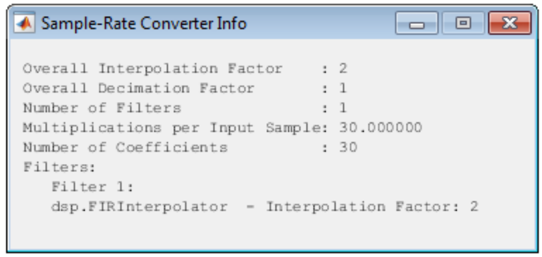
To update the magnitude response while FVTool is running, modify the dialog box parameters and click **Apply**.

View Info

Displays information about the filter system of the Sample-Rate Converter block:

- `Overall Interpolation Factor`
- `Overall Decimation Factor`
- `Number of Filters`
- `Multiplication per Input Sample`
- `Number of Coefficients`
- `Filters`





The button brings the functionality of the `dsp.SampleRateConverter.info` method into the Simulink environment.

Simulate using

Type of simulation to run. You can set this parameter to:

- `Code generation` (default)

  Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`

Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

See Also

dsp.SampleRateConverter	DSP System Toolbox
`Farrow Rate Converter`	DSP System Toolbox

Algorithms

This block brings the capabilities of the dsp.SampleRateConverter System object to the Simulink environment.

For information on the algorithms used by this block, see the "Algorithms" on page 4-1504 section of `dsp.SampleRateConverter`.

Introduced in R2015b

Analysis Methods for Filter System Objects

Analysis Methods for Filter System Objects

Method	Description
Single-rate and Multirate Analysis	
`fvtool`	Filter visualization tool
`info`	Filter information
`freqz`	Frequency response of a discrete-time filter
`phasez`	Phase response of a discrete-time filter
`zerophase`	Zero-phase response of a discrete-time filter
`grpdelay`	Group delay of a discrete-time filter
`phasedelay`	Phase delay of a discrete-time filter
`impz`	Impulse response of a discrete-time filter
`impzlength`	Impulse response length
`stepz`	Step response of a discrete-time filter
`zplane`	Pole/Zero plot
`cost`	Cost estimate
`measure`	Measure filter response
`order`	Filter order
`firtype`	Determine the type (1–4) of a linear phase FIR filter
`coeffs`	Return filter coefficients
Multirate Analysis	
`polyphase`	Polyphase decomposition of multirate filters
`gain`	Gain of a Cascaded Integrator-Comb (CIC) filter
Second-order Sections	
`scale`	Scale second-order sections
`scalecheck`	Check scale of second-order sections

Method	Description
`scaleopts`	Create options object for sos scaling
`cumsec`	Vector of cumulative second-order section filters
`reorder`	Reorder second-order sections
`sos`	Convert IIR filter to Biquad filter
Code Generation	
`realizemdl`	Filter realization diagram
Fixed-point (supported by FIRFilter, IIRFilter, AllpoleFilter, and BiquadFilter only)	
`noisepsd`	Power spectral density of filter output due to roundoff noise
`noisepsdopts`	Create options object for noisepsd computation
`freqrespest`	Frequency response estimate via filtering
`freqrespopts`	Create options object for frequency response estimate
Other	
`isallpass`	True for allpass discrete-time filter
`islinphase`	True for linear discrete-time filter
`ismaxphase`	True if maximum phase
`isminphase`	True if minimum phase
`isreal`	True for discrete-time filter with real coefficients
`isstable`	True if the filter is stable
`isfir`	True if the filter is FIR
`issos`	True if filter is in second order sections form
`specifyall`	Specify all fixed-point properties

System Objects — Alphabetical List

dsp.AdaptiveLatticeFilter System object

Package: dsp

Adaptive lattice filter

Description

The `dsp.AdaptiveLatticeFilter` computes output, error, and coefficients using a Lattice based FIR adaptive filter.

To implement the adaptive FIR filter object:

1. Define and set up your adaptive FIR filter object. See “Construction” on page 4-2.
2. Call step to implement the filter according to the properties of `dsp.AdaptiveLatticeFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`alf = dsp.AdaptiveLatticeFilter` returns a Lattice based FIR adaptive filter System object, `alf`. This System object is used to compute the filtered output and the filter error for a given input and desired signal.

`alf = dsp.AdaptiveLatticeFilter('`*`PropertyName`*`',` *`PropertyValue`*`,...)` returns an `AdaptiveLatticeFilter` System object, `alf`, with each specified property set to the specified value.

`alf =`
`dsp.AdaptiveLatticeFilter(LEN,'`*`PropertyName`*`',`*`PropertyValue`*`,...)`

returns an `AdaptiveLatticeFilter` System object, `alf`, with the Length property set to `LEN` and other specified properties set to the specified values.

Properties

`Method`

Method to calculate filter coefficients

Specify the method used to calculate filter coefficients as one of `'Least-squares Lattice'` |`'QR-decomposition Least-squares Lattice'` |`'Gradient Adaptive Lattice'` . The default value is `'Least-squares Lattice'`. For algorithms used to implement these three different methods, refer to [1] [2]. This property is nontunable.

`Length`

Length of the filter coefficients vector

Specify the length of the FIR filter coefficients vector as a positive integer value. This property is nontunable.

The default value is 32.

`ForgettingFactor`

Least-squares lattice forgetting factor

Specify the Least-squares lattice forgetting factor as a scalar positive numeric value less than or equal to 1. Setting this value to 1 denotes infinite memory during adaptation. This property applies only if the Method property is set to `'Least-squares Lattice'` or `'QR-decomposition Least-squares Lattice'`. The default value is `1`.

This property is tunable.

`StepSize`

Joint process step size of the gradient adaptive filter

Specify the joint process step size of the gradient adaptive lattice filter as a positive numeric scalar less than or equal to 1. This property applies only if the `Method` property is set to `'Gradient Adaptive Lattice'`. The default value is `0.1`.

This property is tunable.

Offset

Offset for denominator of StepSize normalization term

Specify an offset value for the denominator of the `StepSize` normalization term as a nonnegative numeric scalar. A nonzero offset helps avoid a divide-by-near-zero condition when the input signal amplitude is very small. This property applies only if the `Method` property is set to `'Gradient Adaptive Lattice'`. The default value is `1`.

This property is tunable.

ReflectionStepSize

Reflection process step size

Specify the reflection process step size of the gradient adaptive lattice filter as a scalar numeric value between 0 and 1, both inclusive. Use this property only if the `Method` property is set to `'Gradient Adaptive Lattice'`. The default value is the `StepSize` property value.

This property is tunable.

AveragingFactor

Averaging factor of the energy estimator

Specify the averaging factor as a positive numeric scalar less than 1. Use this property to compute the exponentially windowed forward and backward prediction error powers for the coefficient updates. This property applies only if the `Method` property is set to `'Gradient Adaptive Lattice`. The default is the value of `1 - StepSize`.

This property is tunable.

InitialPredictionErrorPower

Initial prediction error power

Specify the initial values for the prediction error vectors as a scalar positive numeric value.

If the `Method` property is set to `'Least-squares Lattice'` or `'QR-decomposition Least-squares Lattice'`, the default value is `1.0`. If the `Method` property is set to `'Gradient Adaptive Lattice'`, the default value is `0.1`.

This property is tunable.

`InitialCoefficients`

Initial coefficients of the filter

Specify the initial values of the FIR adaptive filter coefficients as a scalar or a vector of length equal to the value of the `Length` property. The default value is `0`.

This property is tunable.

`LockCoefficients`

Locked status of the coefficient updates

Specify whether to lock the filter coefficient values. By default, the value of this property is `false`, and the object continuously updates the filter coefficients. If this property is set to `true`, the filter coefficients do not update and their values remain the same.

This property is applicable only if the `Method` property is set to `'Gradient Adaptive Lattice'`.

This property is tunable.

Methods

msesim	Mean-square error for Adaptive Lattice filter
reset	Reset filter states for Adaptive Lattice filter
step	Apply Adaptive Lattice filter to input

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object

Common to All System Objects	
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

QPSK adaptive equalization Using Adaptive Lattice Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create the QPSK signal and the noise, filter them to obtain the received signal, and delay the received signal to obtain the desired signal.

```
D = 16;
b = exp(1i*pi/4)*[-0.7 1];
a = [1 -0.7];
ntr = 1000;
s = sign(randn(1,ntr+D)) + 1i*sign(randn(1,ntr+D));
n = 0.1*(randn(1,ntr+D) + 1i*randn(1,ntr+D));
r = filter(b,a,s) + n;
x = r(1+D:ntr+D);
d = s(1:ntr);
```

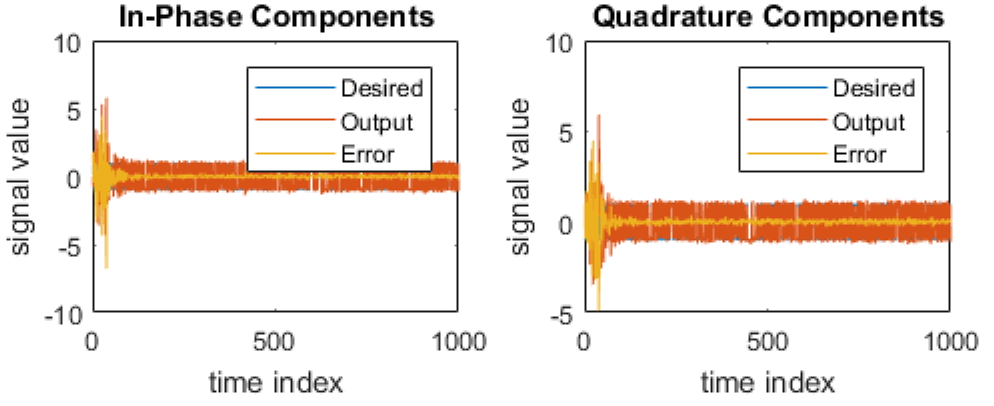
Use the Adaptive Lattice Filter to compute the filtered output and the filter error for the input and desired signal.

```
lam = 0.995;
del = 1;
alf = dsp.AdaptiveLatticeFilter('Length', 32, ...
    'ForgettingFactor', lam, 'InitialPredictionErrorPower', del);
[y,e] = alf(x,d);
```

Plot the In-Phase and the Quadrature components of the desired, output, and the error signals.

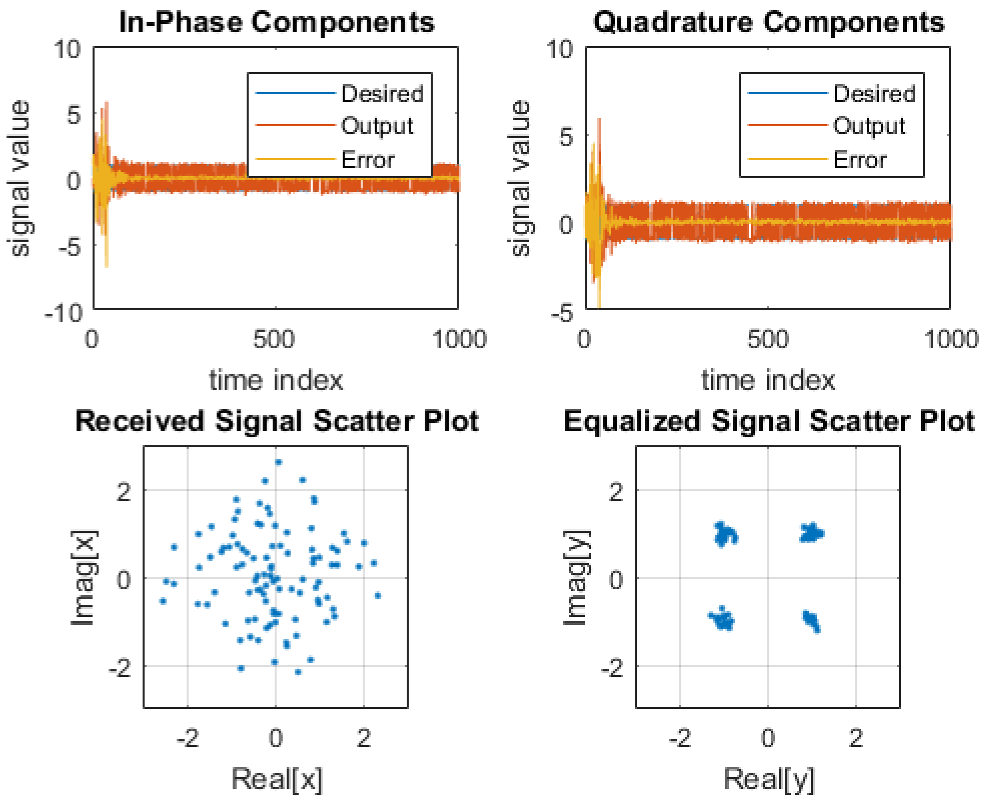
```
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
```

```
xlabel('time index'); ylabel('signal value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('time index'); ylabel('signal value');
```



Plot the received and equalized signals' scatter plots.

```
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```



References

[1] Griffiths, Lloyd J. “A Continuously Adaptive Filter Implemented as a Lattice Structure”. *Proceedings of IEEE Int. Conf. on Acoustics, Speech, and Signal Processing,* Hartford, CT, pp. 683–686, 1977 .

[2] Haykin, S. *Adaptive Filter Theory*, 4th Ed. Upper Saddle River, NJ: Prentice Hall, 1996.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.LMSFilter | dsp.AffineProjectionFilter | dsp.FIRFilter | dsp.RLSFilter | dsp.FrequencyDomainAdaptiveFilter | dsp.FilteredXLMSFilter

Introduced in R2013b

msesim

System object: dsp.AdaptiveLatticeFilter
Package: dsp

Mean-square error for Adaptive Lattice filter

Syntax

```
MSE = msesim(sysObj,X,D)
[MSE,MEANW,W,TRACEK] = msesim(sysObj,X,D)
[...] = msesim(sysObj,X,D,M)
```

Description

`MSE = msesim(sysObj,X,D)` returns a sequence of mean-square errors. This column vector contains estimates of the mean-square error of the adaptive filter at each time instant. The length of `MSE` is equal to `SIZE(X,1)`. The columns of the matrix `X` contain individual input signal sequences, and the columns of the matrix `D` contain corresponding desired response signal sequences.

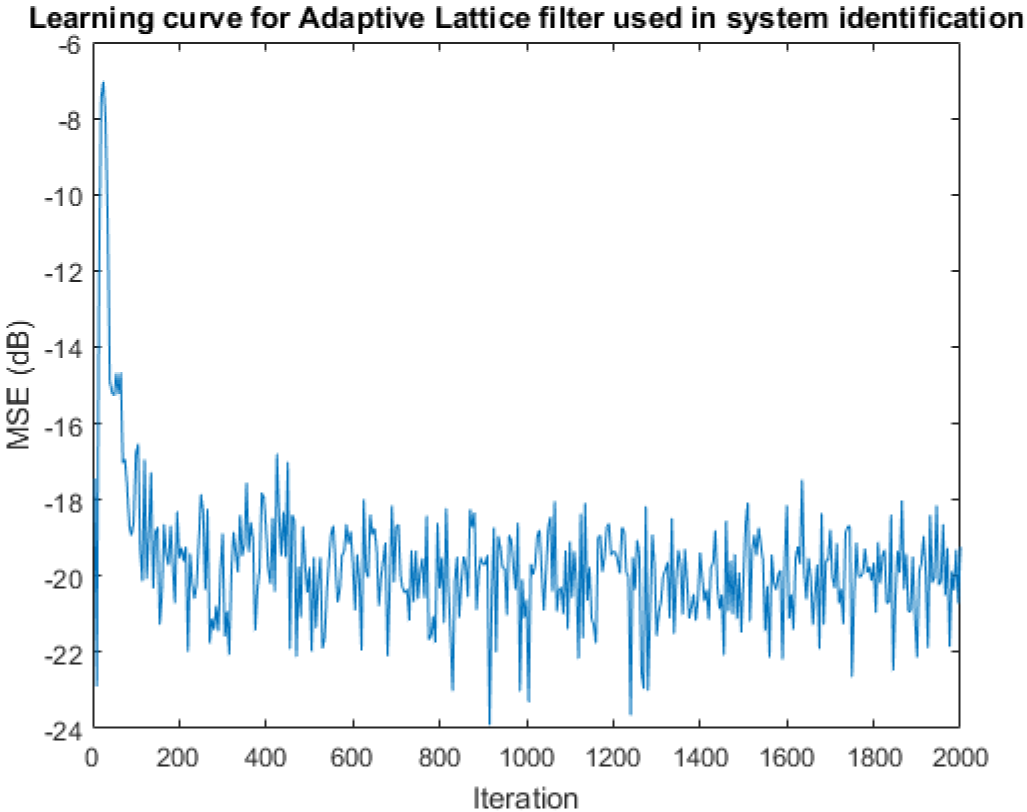
`[MSE,MEANW,W,TRACEK] = msesim(sysObj,X,D)` calculates three parameters corresponding to the simulated behavior of the adaptive filter defined by `sysObj`. `MEANW` is a sequence of coefficient vector means. The columns of this matrix contain estimates of the mean values of the adaptive filter coefficients at each time instant. The dimensions of `MEANW` are `(SIZE(X,1))` by `(sysObj.length)`. `W` is an estimate of the final values of the adaptive filter coefficients for the algorithm corresponding to `sysObj`. `TRACEK` is a sequence of total coefficient error powers. This column vector contains estimates of the total coefficient error power of the adaptive filter at each time instant. The length of `TRACEK` is equal to `SIZE(X,1)`.

`[...] = msesim(sysObj,X,D,M)` specifies an optional decimation factor for computing `MSE`, `MEANW`, and `TRACEK`. If `M` > `1`, every `M`th predicted value of each of these sequences is saved. If omitted, the value of `M` is the default, which is 1.

System identification of an FIR filter Using Adaptive Lattice Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
ha = fir1(31,0.5);
fir = dsp.FIRFilter('Numerator',ha); % FIR system to be identified
iir = dsp.IIRFilter('Numerator',sqrt(0.75),...
        'Denominator',[1 -0.5]);
x = iir(sign(randn(2000,25)));
n = 0.1*randn(size(x));            % Observation noise signal
d = fir(x)+n;                  % Desired signal
l = 32;                            % Filter length
m  = 5;                            % Decimation factor for analysis
                                   % and simulation results
ha = dsp.AdaptiveLatticeFilter(l);
[simmse,meanWsim,Wsim,traceKsim] = msesim(ha,x,d,m);
plot(m*(1:length(simmse)),10*log10(simmse));
xlabel('Iteration'); ylabel('MSE (dB)');
title('Learning curve for Adaptive Lattice filter used in system identification')
```


Learning curve for Adaptive Lattice filter used in system identification
-6
-8
-10
-12
-14
-16
-18
-20
-22
-24
MSE (dB)
0
200
400
600
800
1000
1200
1400
1600
1800
2000
Iteration

reset

System object: dsp.AdaptiveLatticeFilter
Package: dsp

Reset filter states for Adaptive Lattice filter

Syntax

`reset(alf)`

Description

`reset(alf)` resets the internal states of the System object,`alf`, to their initial values. The reset method is always a no-op for unlocked System objects, as the states may not be allocated when the object is not locked.

step

System object: dsp.AdaptiveLatticeFilter
Package: dsp

Apply Adaptive Lattice filter to input

Syntax

```
[Y, ERR] = step(sysObj, x, D)
Y = step(sysObj,x)
[Y1,...,YN] = step(sysObj,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[Y, ERR] = step(sysObj, x, D)` filters the input `x`, using D as the desired signal, and returns the filtered output in `Y` and the filter error in `ERR`. The System object estimates the filter weights needed to minimize the error between the output signal and the desired signal.

`Y = step(sysObj,x)` processes the input data, `x`, to produce the output, `Y`, from the System object, `sysObj`. `[Y1,...,YN] = step(sysObj,x)` produces `N` outputs.

Every System object has a step method. The step method processes the input data according to the object algorithm. The number of input and output arguments depends on the algorithm, and may depend also on one or more property settings. The step method for some objects accepts fixed-point (fi) inputs.

Calling step on an object puts that object into a locked state. When locked, you cannot change nontunable properties or any input characteristics (size, data type and complexity) without reinitializing (unlocking and relocking) the object.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.AffineProjectionFilter System object

Package: dsp

Compute output, error and coefficients using Affine Projection (AP) Algorithm

Description

The `AffineProjectionFilter` object filters each channel of the input using AP filter implementations.

To filter each channel of the input:

1. Define and set up your AP filter. See “Construction” on page 4-16.
2. Call step to filter each channel of the input according to the properties of `dsp.AffineProjectionFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`apf = dsp.AffineProjectionFilter` returns an adaptive FIR filter System object, `apf`. This System object computes the filtered output and the filter error for a given input and desired signal using the Affine Projection (AP) algorithm.

`apf = dsp.AffineProjectionFilter('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns an AP filter System object, `apf`, with each specified property set to the specified value.

`apf = dsp.AffineProjectionFilter(LEN, '`*`PropertyName`*`',` *`PropertyValue`*`, ...)` returns an AP filter System object, `apf`. This System object has the Length property set to `LEN`, and other specified properties set to the specified values.

Properties

Method

Method to calculate the filter coefficients

Specify the method used to calculate filter coefficients as one of | `Direct Matrix Inversion` | `Recursive Matrix Update` | `Block Direct Matrix Inversion`. The default value is `Direct Matrix Inversion`. This property is nontunable.

Length

Length of filter coefficients vector

Specify the length of the FIR filter coefficients vector as a scalar positive integer value. The default value is `32`. This property is nontunable.

ProjectionOrder

Projection order of the affine projection algorithm

Specify the projection order of the affine projection algorithm as a scalar positive integer value greater than or equal to 2. This property defines the size of the input signal covariance matrix. The default value is `2`. This property is nontunable.

StepSize

Affine projection step size

Specify the affine projection step size factor as a scalar non-negative numeric value between 0 and 1, both inclusive. Setting step equal to one provides the fastest convergence during adaptation. The default value is `1`. This property is tunable.

InitialCoefficients

Initial coefficients of the filter

Specify the initial values of the FIR adaptive filter coefficients as a scalar or a vector of length equal to the `Length` property value. The default value is `0`. This property is tunable.

InitialOffsetCovariance

Initial values of the offset input covariance matrix

Specify the initial values for the offset input covariance matrix. This property must be either a scalar positive numeric value or a positive-definite square matrix with each dimension equal to the `ProjectionOrder` property value. If it is a scalar value, the `OffsetCovariance` property is initialized to a diagonal matrix with the diagonal elements equal to that scalar value. If it is a square matrix, the `OffsetCovariance` property is initialized to the value of that square matrix. This property is applicable only if the Method property is set to `Direct Matrix Inversion` or `Block Direct Matrix Inversion`. The default value is `1`. This property is tunable.

InitialInverseOffsetCovariance

Initial values of the offset input covariance matrix inverse

Specify the initial values for the offset input covariance matrix inverse. This property must be either a scalar positive numeric value or a positive-definite square matrix with each dimension equal to the `ProjectionOrder` property value. If it is a scalar value, the `InverseOffsetCovariance` property is initialized to a diagonal matrix with each diagonal element equal to that scalar value. If it is a square matrix, the `InverseOffsetCovariance` property is initialized to the values of that square matrix. This property is applicable only if the `Method` property is set to `Recursive Matrix Update`. The default value is `20`. This property is tunable.

InitialCorrelationCoefficients

Initial correlation coefficients

Specify the initial values of the correlation coefficients of the FIR filter as a scalar or a vector of length equal to `ProjectionOrder` –`1`. This property is applicable only if the `Method` property is set to `Recursive Matrix Update`. The default value is `0`. This property is tunable.

LockCoefficients

Lock coefficient updates

Specify whether the filter coefficient values should be locked. When you set this property to `true`, the filter coefficients are not updated and their values remain the same. The default value is `false` (filter coefficients are continuously updated). This property is tunable.

Methods

msesim	Mean-square error for Affine Projection filter
reset	Reset the internal states of a System object
step	Process inputs using the affine projection filter algorithm

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

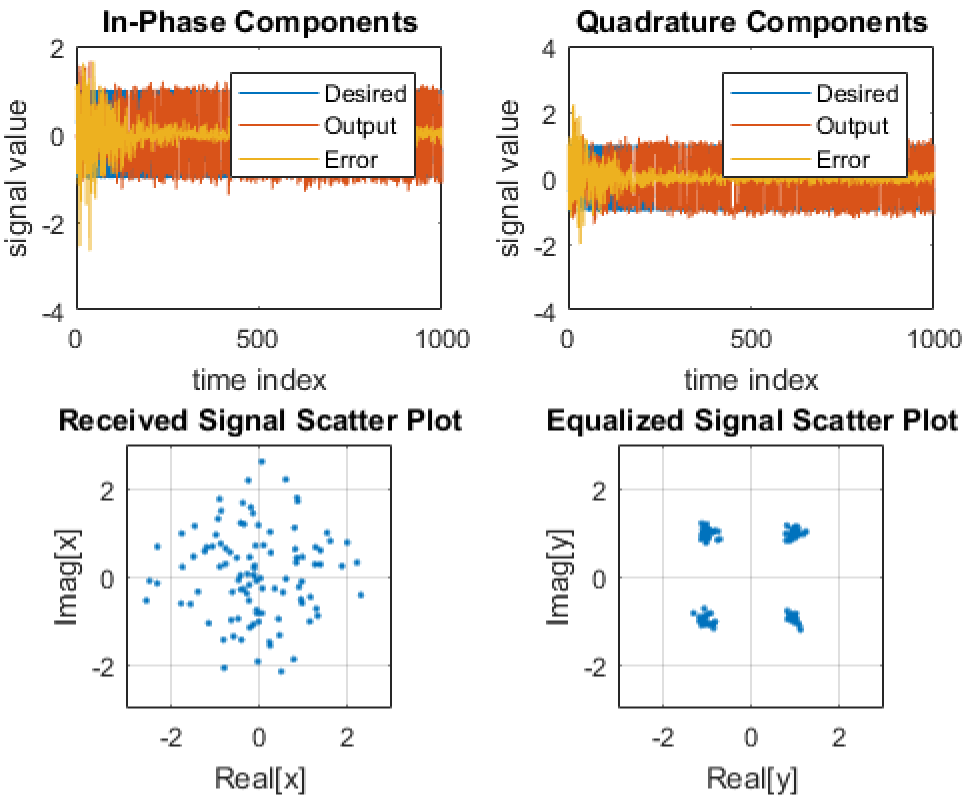
QPSK Adaptive Equalization

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

QPSK Adaptive Equalization Using a 32-Coefficient FIR Filter (1000 Iterations)

```
D = 16;                         % Number of samples of delay
b = exp(1i*pi/4)*[-0.7 1];       % Numerator coefficients of channel
a = [1 -0.7];                   % Denominator coefficients of channel
ntr = 1000;                     % Number of iterations
s = sign(randn(1,ntr+D)) + 1i*sign(randn(1,ntr+D)); % Baseband signal
n = 0.1*(randn(1,ntr+D) + 1i*randn(1,ntr+D));       % Noise signal
r = filter(b,a,s)+n;            % Received signal
x = r(1+D:ntr+D);               % Input signal (received signal)
d = s(1:ntr);                   % Desired signal (delayed QPSK signal)
mu = 0.1;                       % Step size
po = 4;                         % Projection order
offset = 0.05;                  % Offset for covariance matrix
apf = dsp.AffineProjectionFilter('Length', 32, ...
    'StepSize', mu, 'ProjectionOrder', po, ...
```

```
    'InitialOffsetCovariance',offset);
[y,e] = apf(x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('time index'); ylabel('signal value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('time index'); ylabel('signal value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```



Algorithms

The affine projection algorithm (APA), is an adaptive scheme that estimates an unknown system based on multiple input vectors [1]. It is designed to improve the performance of other adaptive algorithms, mainly those that are LMS-based. The affine projection algorithm reuses old data resulting in fast convergence when the input signal is highly correlated, leading to a family of algorithms that can make trade-offs between computation complexity with convergence speed [2].

The following equations describe the conceptual algorithm used in designing AP filters:

$$\mathbf{U}_{ap}(n) = \begin{pmatrix} u(n) & \dots & u(n-L) \\ \vdots & \ddots & \vdots \\ u(n-N) & \cdots & u(n-L-N) \end{pmatrix} = \begin{pmatrix} \mathbf{u}(n) & \mathbf{u}(n-1) & \cdots & \mathbf{u}(n-L) \end{pmatrix}$$

$$\mathbf{y}_{ap}(n) = \mathbf{U}^T{}_{ap}(n)\mathbf{w}(n) = \begin{pmatrix} y(n) \\ \bullet \\ \bullet \\ \bullet \\ y(n-L) \end{pmatrix}$$

$$\mathbf{d}_{ap}(n) = \begin{pmatrix} d(n) \\ \bullet \\ \bullet \\ \bullet \\ d(n-L) \end{pmatrix}$$

$$\mathbf{e}_{ap}(n) = \mathbf{d}_{ap}(n) - \mathbf{y}_{ap}(n) = \begin{pmatrix} e(n) \\ \bullet \\ \bullet \\ \bullet \\ e(n-L) \end{pmatrix}$$

$$\mathbf{w}(n) = \mathbf{w}(n-1) + \mu \mathbf{U}_{ap}(n)(\mathbf{U}_{ap}{}^H(n)\mathbf{U}_{ap}(n) + \mathbf{C})^{-1}\mathbf{e}_{ap}$$

where **C** is either ε**I** if the initial offset covariance is a scalar ε, or **R** if the initial offset covariance is a matrix **R**. The variables are as follows:

Variable	Description
n	The current time index
u(n)	The input sample at step n
$\mathbf{U}_{ap}$(n)	The matrix of the last L+1 input signal vectors

Variable	Description
w(n)	The adaptive filter coefficients vector
y(n)	The adaptive filter output
d(n)	The desired signal
e(n)	The error at step n
L	The projection order
N	The filter order (i.e., filter length = N+1)
μ	The step size

References

[1] K. Ozeki, T. Umeda, "An adaptive Filtering Algorithm Using an Orthogonal Projection to an Affine Subspace and its Properties", Electron. Commun. Jpn. 67-A(5), May 1984, pp. 19–27.

[2] *Paulo S. R. Diniz, Adaptive Filtering: Algorithms and Practical Implementation,* Second Edition. *Boston: Kluwer Academic Publishers, 2002*

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

dsp.FIRFilter | dsp.RLSFilter | dsp.LMSFilter

Introduced in R2013a

msesim

System object: dsp.AffineProjectionFilter
Package: dsp

Mean-square error for Affine Projection filter

Syntax

```
MSE = msesim(sysObj,X,D)
[MSE,MEANW,W,TRACEK] = msesim(sysObj,X,D)
[...] = msesim(sysObj,X,D,M)
```

Description

`MSE = msesim(sysObj,X,D)` returns a sequence of mean-square errors. This column vector contains estimates of the mean-square error of the adaptive filter at each time instant. The length of `MSE` is equal to `SIZE(X,1)`. The columns of the matrix `X` contain individual input signal sequences, and the columns of the matrix `D` contain corresponding desired response signal sequences.

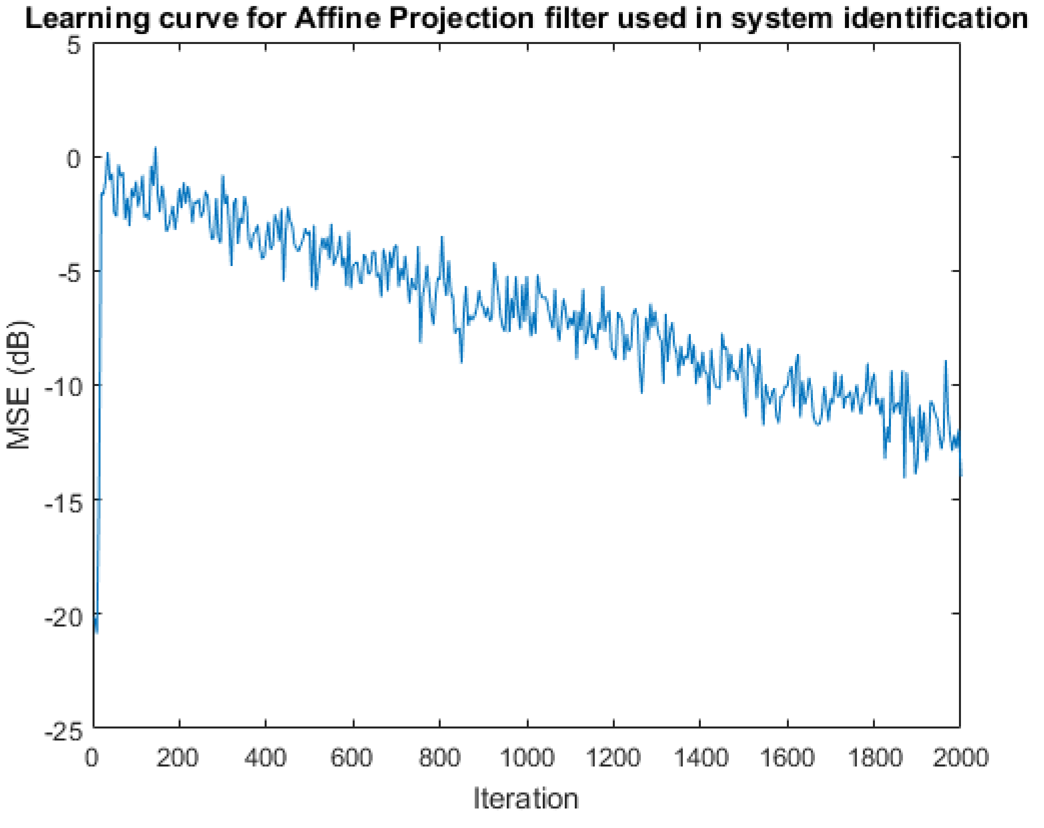
`[MSE,MEANW,W,TRACEK] = msesim(sysObj,X,D)` calculates three parameters corresponding to the simulated behavior of the adaptive filter defined by `sysObj`. `MEANW` is a sequence of coefficient vector means. The columns of this matrix contain estimates of the mean values of the adaptive filter coefficients at each time instant. The dimensions of `MEANW` are `(SIZE(X,1))` by `(sysObj.length)`. `W` is an estimate of the final values of the adaptive filter coefficients for the algorithm corresponding to `sysObj`. `TRACEK` is a sequence of total coefficient error powers. This column vector contains estimates of the total coefficient error power of the adaptive filter at each time instant. The length of `TRACEK` is equal to `SIZE(X,1)`.

`[...] = msesim(sysObj,X,D,M)` specifies an optional decimation factor for computing `MSE`, `MEANW`, and `TRACEK`. If `M` > `1`, every M^{th} predicted value of each of these sequences is saved. If omitted, the value of `M` is the default, which is 1.

System Identification of an FIR Filter Using Affine Projection Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
ha = fir1(31,0.5);
fir = dsp.FIRFilter('Numerator',ha); % FIR system to be identified
iir = dsp.IIRFilter('Numerator',sqrt(0.75),...
        'Denominator',[1 -0.5]);
x = iir(sign(randn(2000,25)));
n = 0.1*randn(size(x));              % Observation noise signal
d = fir(x)+n;                    % Desired signal
l = 32;                              % Filter length
mu = 0.008;                          % Affine Projection filter Step size.
m  = 5;                              % Decimation factor for analysis
                                     % and simulation results
apf = dsp.AffineProjectionFilter(l,'StepSize',mu);
[simmse,meanWsim,Wsim,traceKsim] = msesim(apf,x,d,m);
plot(m*(1:length(simmse)),10*log10(simmse));
xlabel('Iteration'); ylabel('MSE (dB)');
title('Learning curve for Affine Projection filter used in system identification')
```



Learning curve for Affine Projection filter used in system identification
MSE (dB)
5
0
-5
-10
-15
-20
-25
0
200
400
600
800
1000
1200
1400
1600
1800
2000
Iteration

reset

System object: dsp.AffineProjectionFilter
Package: dsp

Reset the internal states of a System object

Syntax

`reset(apf)`

Description

`reset(apf)` resets the internal states of the System object, `apf`, to their initial values.

For many System objects, this method is nonoperational. Objects that have internal states describe in their **help** what the reset method does for that object. The `reset` method is always nonoperational for unlocked System objects, as states may not be allocated when the object is not locked.

step

System object: dsp.AffineProjectionFilter
Package: dsp

Process inputs using the affine projection filter algorithm

Syntax

```
Y = step(apf,x)
[Y1,...,YN] = step(apf,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(apf,x)` processes the input data, `x`, to produce the output, `Y`, for the System object, `apf`. `[Y1,...,YN] = step(apf,x)` produces N outputs.

Every System object has a `step` method. The `step` method processes the input data according to the object algorithm. The number of the input and the output arguments depends on the algorithm, and may depend also on one or more property settings. The step method for some objects accepts fixed-point (fi) inputs.

Calling `step` on an object puts that object into a locked state. When locked, you cannot change nontunable properties or any input characteristics (size, data type and complexity) without reinitializing (unlocking and relocking) the object.

dsp.AllpassFilter System object

Package: dsp

Single section or cascaded allpass filter

Note: MATLAB code with `LatticeCoefficients`, `AllpassCoefficients`, and `WDFCoefficients` properties set to cell arrays will error in a future release. Set these properties to array values instead.

Description

The `AllpassFilter` object filters each channel of the input using Allpass filter implementations. This System object supports code generation. To import this object into Simulink, use the MATLAB System block.

To filter each channel of the input:

1. Define and set up your Allpass filter. See "Construction" on page 4-29.
2. Call step to filter each channel of the input according to the properties of `dsp.AllpassFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`Allpass = dsp.AllpassFilter` returns an Allpass filter System object, `Allpass`, that filters each channel of the input signal independently using an allpass filter, with the default structure and coefficients.

`Allpass = dsp.AllpassFilter('PropertyName',PropertyValue, ...)` returns an Allpass filter System object, `Allpass`, with each property set to the specified value.

Properties

`Structure`

Internal allpass filter structure

You can specify the internal allpass filter implementation structure as one of `| Minimum multiplier | Lattice | Wave Digital Filter`. The default is `Minimum multiplier`. Each structure uses a different set of coefficients, independently stored in the corresponding object property.

`AllpassCoefficients`

Allpass polynomial coefficients

Specify the real allpass polynomial filter coefficients. This property is applicable only when the `Structure` property is set to `Minimum multiplier`. Specify this property as either an *N*-by-`1` or *N*-by-`2` matrix of *N* first-order or second-order allpass sections. The default value of this property is `[-2^(-1/2) 0.5]`. The default value defines a stable second-order allpass filter with poles and zeros located at $\pm\pi/3$ in the Z plane. This property is tunable.

`WDFCoefficients`

Wave Digital Filter allpass coefficients

Specify the real allpass coefficients in the Wave Digital Filter form. This property is only applicable when the `Structure` property is set to `Wave Digital Filter`. Specify this property as either a *N*-by-`1` or *N*-by-`2` matrix of *N* first-order or second-order allpass sections. All elements must have absolute values less than or equal to `1`. The default value for this property is [`1/2, -2^(1/2)/3`]. This value is a transformed version of the default value of `AllpassCoefficients`, computed using `allpass2wdf(AllpassCoefficients)`. These coefficients define the same stable second-order allpass filter as when `Structure` is set to `'Minimum multiplier'`. This property is tunable.

`LatticeCoefficients`

Lattice allpass coefficients

Specify the real or complex allpass coefficients as lattice reflection coefficients. This property is applicable only if the `Structure` property is set to `Lattice`. Specify this property as either a row vector (single-section configuration) or a column vector. The default is `[-2^(1/2)/3, 1/2]`. This value is a transformed and transposed version of the default value of `AllpassCoefficients`, computed using `transpose(tf2latc([1 h.AllpassCoefficients]))`. These coefficients define the same stable second-order allpass filter as when `Structure` is set to `'Lattice'`. This property is tunable.

TrailingFirstOrderSection

Indicate if last section is first order

Indicate if last section is first order or second order. When you set `TrailingFirstOrderSection` to `true`, the last section is considered to be first-order, and the second element of the last row of the N-by-2 matrix is ignored. When you set `TrailingFirstOrderSection` to `false`, the last section is considered to be second-order. The default is `false`.

Methods

reset	Reset internal states of a System object
step	Process inputs using allpass filter

For additional methods, see "Analysis Methods for Filter System Objects" on page 3-2.

For a complete list of analysis methods supported for the `dsp.AllpassFilter` object, enter `dsp.AllpassFilter.helpFilterAnalysis` at the MATLAB command prompt.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Lowpass Filtering using Two Allpass Filters

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Construct the Allpass Filters

```
Fs  = 48000;    % in Hz
FL = 1024;
APF1 = dsp.AllpassFilter('AllpassCoefficients',...
    [-0.710525516540603   0.208818210000029]);
APF2 =  dsp.AllpassFilter('AllpassCoefficients',...
    [-0.940456403667957   0.6;...
    -0.324919696232907   0],...
    'TrailingFirstOrderSection',true);
```

Construct the Transfer Function Estimator to estimate the transfer function between the random input and the Allpass filtered output

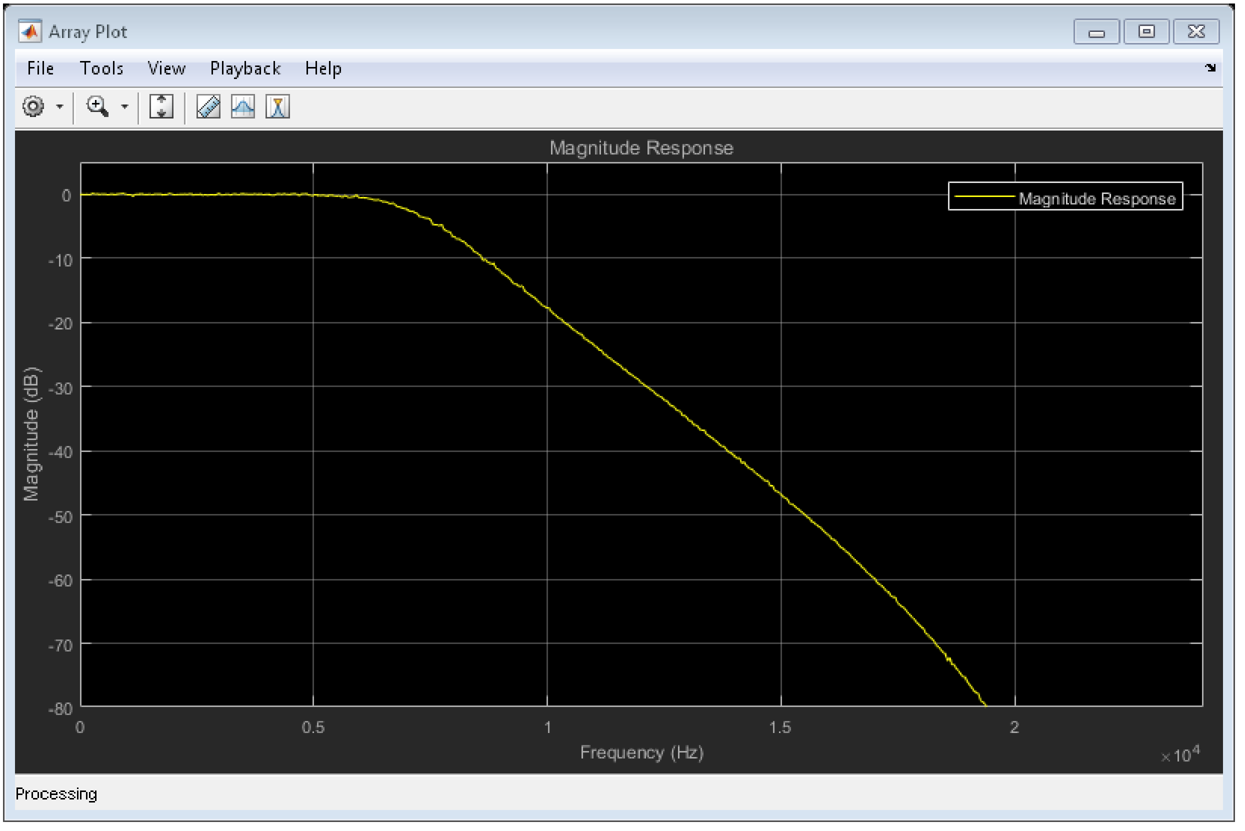
```
TFE = dsp.TransferFunctionEstimator('FrequencyRange',...
    'onesided','SpectralAverages',2);
```

Construct the ArrayPlot to plot the magnitude response

```
AP = dsp.ArrayPlot('PlotType','Line','YLimits', [-80 5],...
    'YLabel','Magnitude (dB)','SampleIncrement', Fs/FL,...
    'XLabel','Frequency (Hz)','Title','Magnitude Response',...
    'ShowLegend', true,'ChannelNames',{'Magnitude Response'});
```

Filter the Input and show the magnitude response of the estimated transfer function between the input and the filtered output

```
tic;
while toc < 5
    in  = randn(FL,1);
    out = 0.5.*(APF1(in) + APF2(in));
    A = TFE(in, out);
    AP(db(A));
end
```



Algorithms

The transfer function of an allpass filter is given by

$$H(z)=\frac{c(n)+c(n-1)z^{-1}+\ldots+z^{-n}}{1+c(1)z^{-1}+\ldots+c(n)z^{-n}}.$$

c is allpass polynomial coefficients vector. The order, n, of the transfer function is the length of vector c.

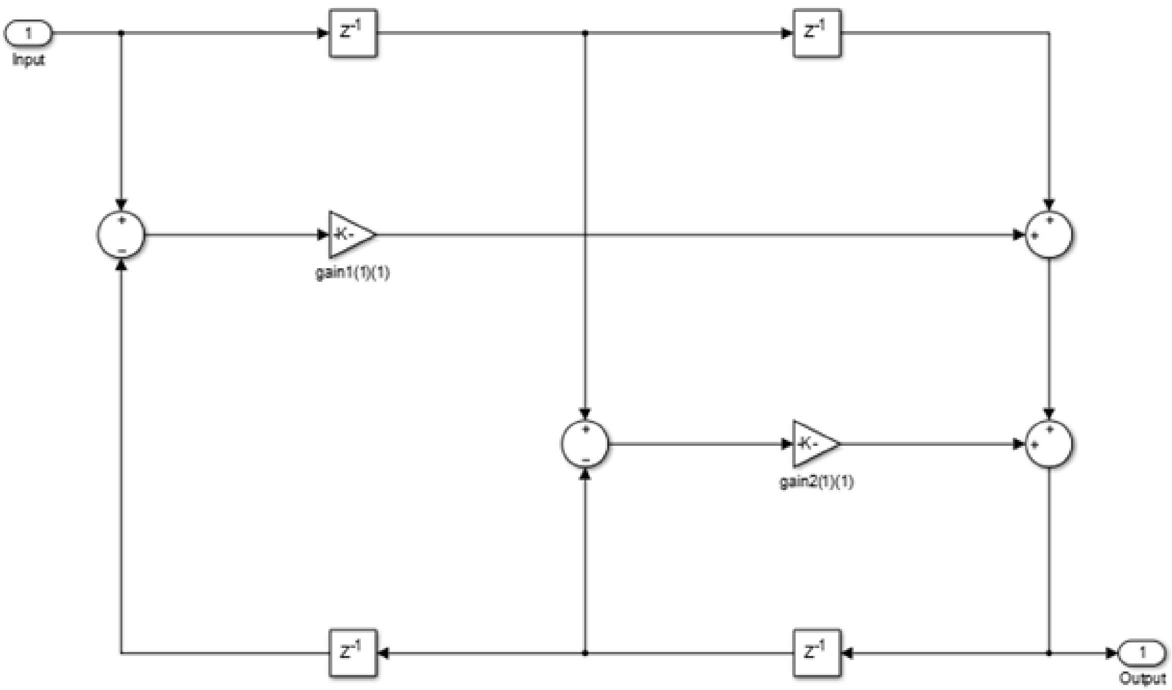
In the minimum multiplier form and wave digital form, the allpass filter is implemented as a cascade of either second-order (biquad) sections or first-order sections. When the coefficients are specified as an N-by-2 matrix, each row of the matrix specifies the coefficients of a second-order filter. The last element of the last row can be ignored based on the trailing first-order setting. When the coefficients are specified as an N-by-1 matrix, each element in the matrix specifies the coefficient of a first-order filter. The cascade of all the filter sections forms the allpass filter.

In the lattice form, the coefficients are specified as a vector.

These structures are computationally more economical and structurally more stable compared to the generic IIR filters, such as df1, df1t, df2, df2t. For all structures, the allpass filter can be a single-section or a multiple-section (cascaded) filter. The different sections can have different orders, but they are all implemented according to the same structure.

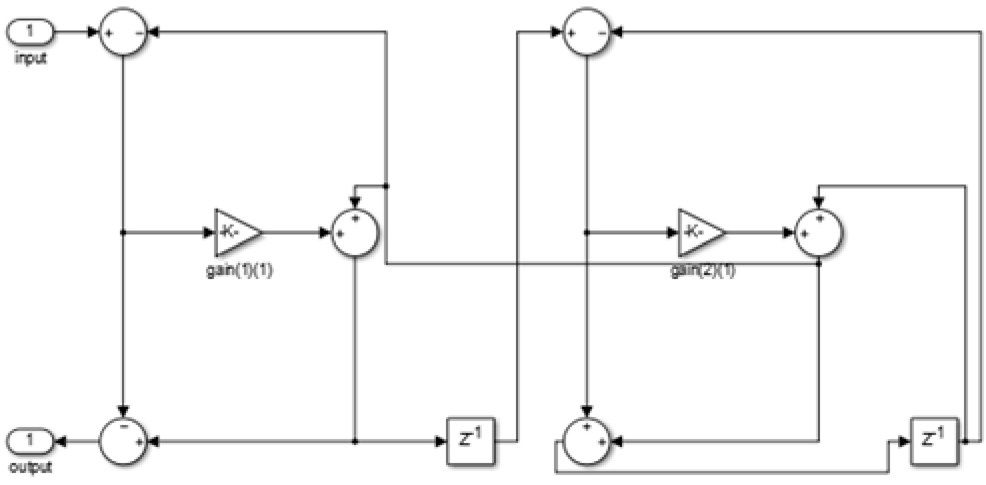
Minimum Multiplier

This structure realizes the allpass filter with the minimum number of required multipliers, equal to the order `n`. It also uses `2n` delay units and `2n` adders. The multipliers uses the specified coefficients, which are equal to the polynomial vector c in the allpass transfer function. In this second-order section of the minimum multiplier structure, the coefficients vector, c, is equal to [`0.1 -0.7`].



Wave Digital Filter

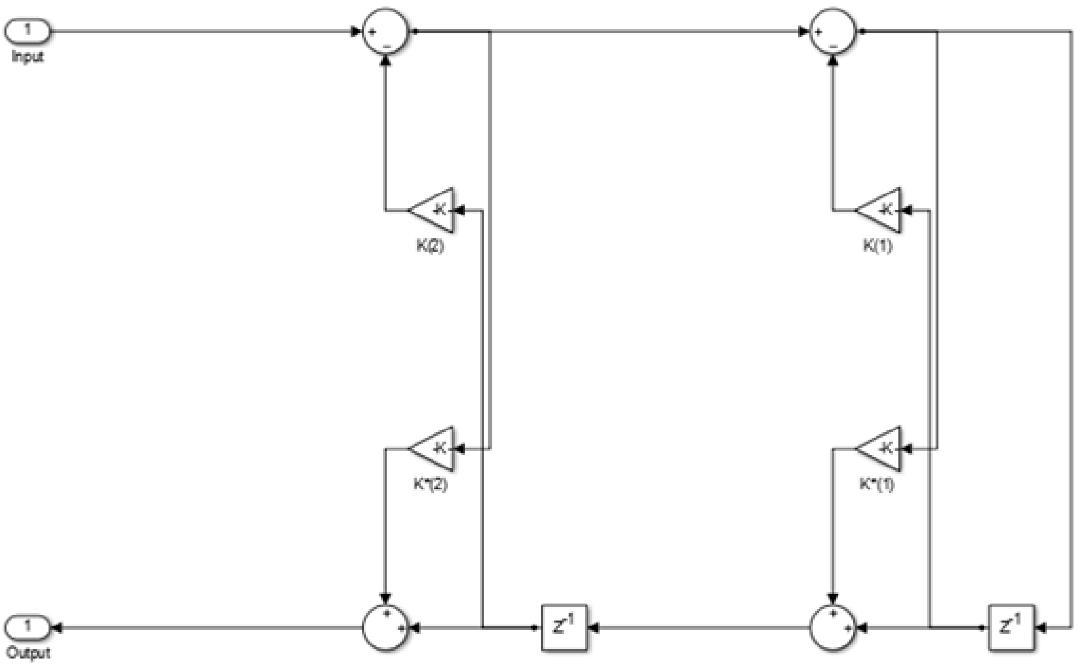
This structure uses `n` multipliers, but only `n` delay units, at the expense of requiring `3n` adders. To use this structure, specify the coefficients in wave digital filter (WDF) form. Obtain the WDF equivalent of the conventional allpass coefficients using `allpass2wdf(allpass_coefficients)`. To convert WDF coefficients into the equivalent allpass polynomial form, use `wdf2allpass(WDF coefficients)`. In this second-order section of the WDF structure, the coefficients vector w is equal to `allpass2wdf([0.1 -0.7])`.



Lattice

This lattice structure uses `2n` multipliers, `n` delay units, and `2n` adders. To use this structure, specify the coefficients as a vector.

You can obtain the lattice equivalent of the conventional allpass coefficients using `transpose(tf2latc(1, [1 allpass_coefficients]))`. In the following second-order section of the lattice structure, the coefficients vector is computed using `transpose(tf2latc(1, [1 0.1 -0.7]))`. Use these coefficients for a filter that is functionally equivalent to the minimum multiplier structure with coefficients [0.1 -0.7].



References

[1] Regalia, Philip A. and Mitra Sanjit K. and Vaidyanathan, P. P. (1988) “The Digital All-Pass Filter: AVersatile Signal Processing Building Block.” *Proceedings of the IEEE*, Vol. 76, No. 1, 1988, pp. 19–37

[2] *M. Lutovac, D. Tosic, B. Evans, Filter Design for Signal Processing Using MATLAB and Mathematica. Prentice Hall, 2001*

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The System object supports code generation only when the `Structure` property is set to `Minimum multiplier` or `Lattice`.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.BiquadFilter | dsp.IIRFilter

Introduced in R2013a

reset

System object: dsp.AllpassFilter
Package: dsp

Reset internal states of a System object

Syntax

`reset(OBJ)`

Description

`reset(OBJ)` resets the internal states of the System object, `OBJ`, to their initial values.

For many System objects, this method is equivalent to no operation being executed. The `reset` method is always nonoperational for unlocked System objects, as states may not be allocated when the object is not locked.

step

System object: dsp.AllpassFilter
Package: dsp

Process inputs using allpass filter

Syntax

```
Y = step(OBJ,x)
[Y1,...,YN]= step(OBJ,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(OBJ,x)` processes the input data, `x`, to produce the output, `Y`, for the System object, `OBJ`. `[Y1,...,YN]= step(OBJ,x)` produces N outputs.

Every System object has a `step` method. The `step` method processes the input data according to the object algorithm. The number of the input and the output arguments depends on the algorithm, and may depend also on one or more property settings. The step method for some objects accepts fixed-point (fi) inputs.

Calling `step` on an object puts that object into a locked state. When locked, you cannot change nontunable properties or any input characteristics (size, data type and complexity) without reinitializing (unlocking and relocking) the object.

dsp.AllpoleFilter System object

Package: dsp

IIR Filter with no zeros

Description

The `AllpoleFilter` object filters each channel of the input using Allpole filter implementations.

To filter each channel of the input:

1. Define and set up your Allpole filter. See "Construction" on page 4-41.
2. Call step to filter each channel of the input according to the properties of `dsp.AllpoleFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`allpole = dsp.AllpoleFilter` returns an Allpole filter System object, `allpole`, which independently filters each channel of the input over successive calls to the `step` method. This System object uses a specified Allpole filter implementation, and it supports variable-size input.

`allpole = dsp.AllpoleFilter('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns an Allpole filter System object, `allpole`, with each property set to the specified value.

Properties

`Structure`

Filter structure

Specify the filter structure.
You can specify the filter structure as one of | `Direct form` | `Direct form transposed` | `Lattice AR`. The default is `Direct form`. Analysis methods are not supported for fixed-point processing if the structure is `Direct form` or `Direct form transposed`. This property is nontunable.

Denominator

Filter denominator coefficients

Specify the denominator coefficients as a real or complex numeric row vector. This property is applicable when the Structure property is set to one of `Direct form` | `Direct form transposed`. The default value of this property is `[1 0.1]`. This property is tunable.

ReflectionCoefficients

Lattice filter coefficients

Specify the lattice filter coefficients as a real or complex numeric row vector. This property is applicable when the Structure property is set to `Lattice AR`. The default value of this property is `[0.2 0.4]`. This property is tunable.

InitialConditions

Initial conditions for the filter states

Specify the initial conditions of the filter states. The default value is `0`.

You can specify the initial conditions as a scalar, vector, or matrix. If you specify a scalar value, this System object initializes all delay elements in the filter to that value. You can also specify a vector whose length equals the number of delay elements in the filter. When you do so, each vector element specifies a unique initial condition for the corresponding delay element. The object applies the same vector of initial conditions to each channel of the input signal.

You can also specify a matrix with the same number of rows as the number of delay elements in the filter and one column for each channel of the input signal. In this case, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel. This property is tunable.

CoefficientsDataType

Denominator coefficients word- and fraction-length designations

Specify the denominator coefficients fixed-point data type as one of `Same word length as input` | `Custom`. The default is `Same word length as input`. This property is nontunable.

`ReflectionCoefficientsDataType`

Reflection coefficients word- and fraction-length designations

Specify the reflection coefficients fixed-point data type as one of `Same word length as input` | `Custom`. The default is `Same word length as input`. This property is nontunable.

Fixed-Point Properties

`ProductDataType`

Product word- and fraction-length designations

Specify the product fixed-point data type as one of | `Full precision` | `Same as input` | `Custom` |. The default is `Full precision`. This property is nontunable.

`AccumulatorDataType`

Accumulator word- and fraction-length designations

Specify the accumulator fixed-point data type to one of | `Full precision` | `Same as input` | `Same as product` | `Custom` |. The default is `Full precision`. This property is nontunable.

`OutputDataType`

Output word- and fraction-length designations

Specify the output fixed-point data type as one of | `Same as accumulator` | `Same as input` | `Custom` |. The default is `Same as input`. This property is nontunable.

`StateDataType`

State word- and fraction-length designations

Specify the state fixed-point data type as one of | `Same as input` | `Same as accumulator` | `Custom`. The default is `Same as accumulator`. This property is nontunable.

CustomCoefficientsDataType

Custom denominator word- and fraction-lengths

Specify the denominator coefficients fixed-point type as an autosigned numerictype (Fixed-Point Designer) object. This property is applicable when the `CoefficientsDataType` on page 4-42 property is `Custom`. The default value of this property is numerictype ([],16,15). This property is nontunable.

CustomReflectionCoefficientsDataType

Custom reflection coefficients word- and fraction-lengths

Specify the denominator coefficients fixed-point type as an autosigned numerictype (Fixed-Point Designer) object. This property is applicable when the `ReflectionCoefficientsDataType` on page 4-43 property is `Custom`. The default value of this property is numerictype ([],16,15). This property is nontunable.

CustomProductDataType

Custom Product word- and fraction-lengths

Specify the product fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `ProductDataType` on page 4-43 property to `Custom`. The default is `numerictype ([],32,30)`. This property is nontunable.

CustomAccumulatorDataType

Custom accumulator word- and fraction-lengths

Specify the accumulator fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `AccumulatorDataType` on page 4-43 property to `Custom`. The default is `numerictype([],32,30)`. This property is nontunable.

CustomStateDataType

Custom state word- and fraction-lengths

Specify the state fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `StateDataType` on page 4-43 property to `Custom`. The default is `numerictype([],16,15)`. This property is nontunable.

CustomOutputDataType

Custom output word- and fraction-lengths

Specify the output fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `OutputDataType` on page 4-43 property to `Custom`. The default is `numerictype([],16,15)`. This property is nontunable.

Methods

freqz	Frequency response
fvtool	Open filter visualization tool
impz	Impulse response
phasez	Unwrapped phase response
reset	Reset internal states of Allpole filter
step	Filter input with Allpole filter object

More "Analysis Methods for Filter System Objects" on page 3-2.

Note: In AllpoleFilter, analysis methods are not supported for fixed-point processing if the structure is `Direct form` or `Direct form transposed`.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

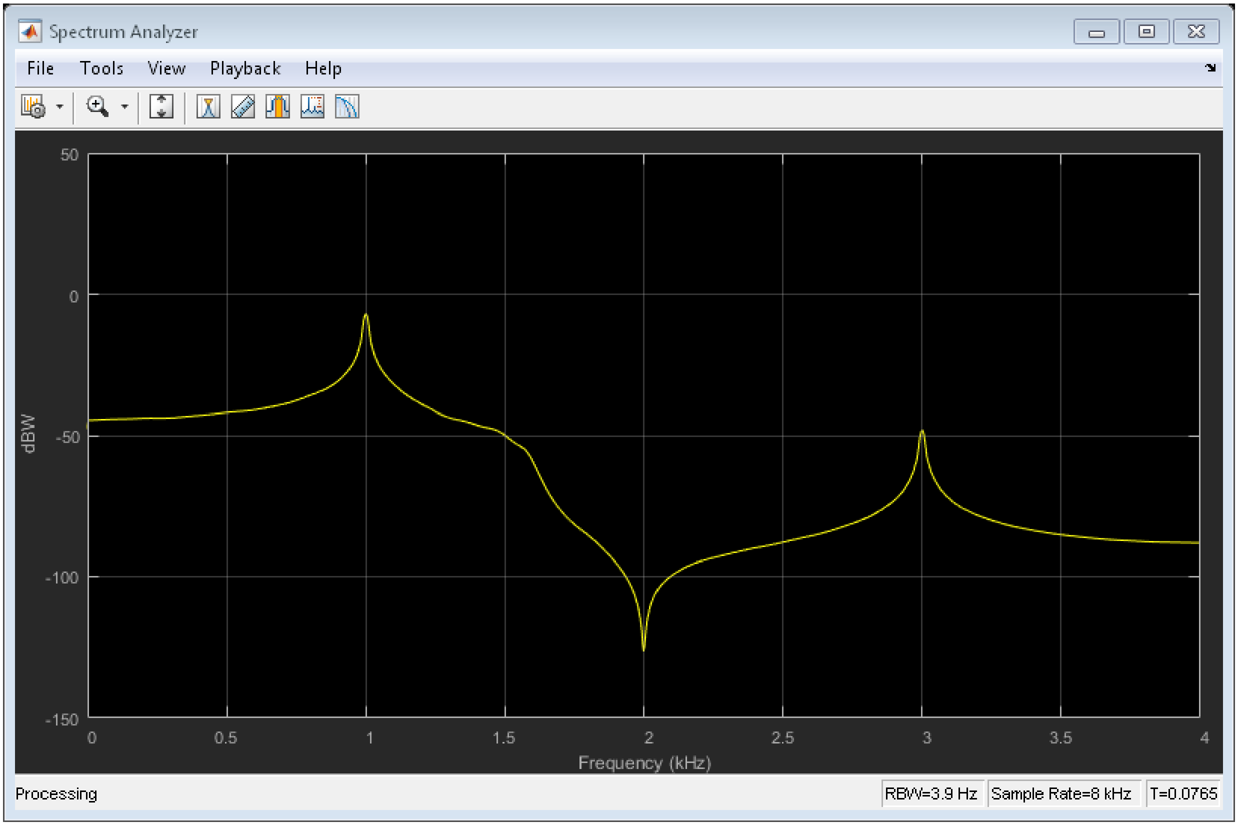
Examples

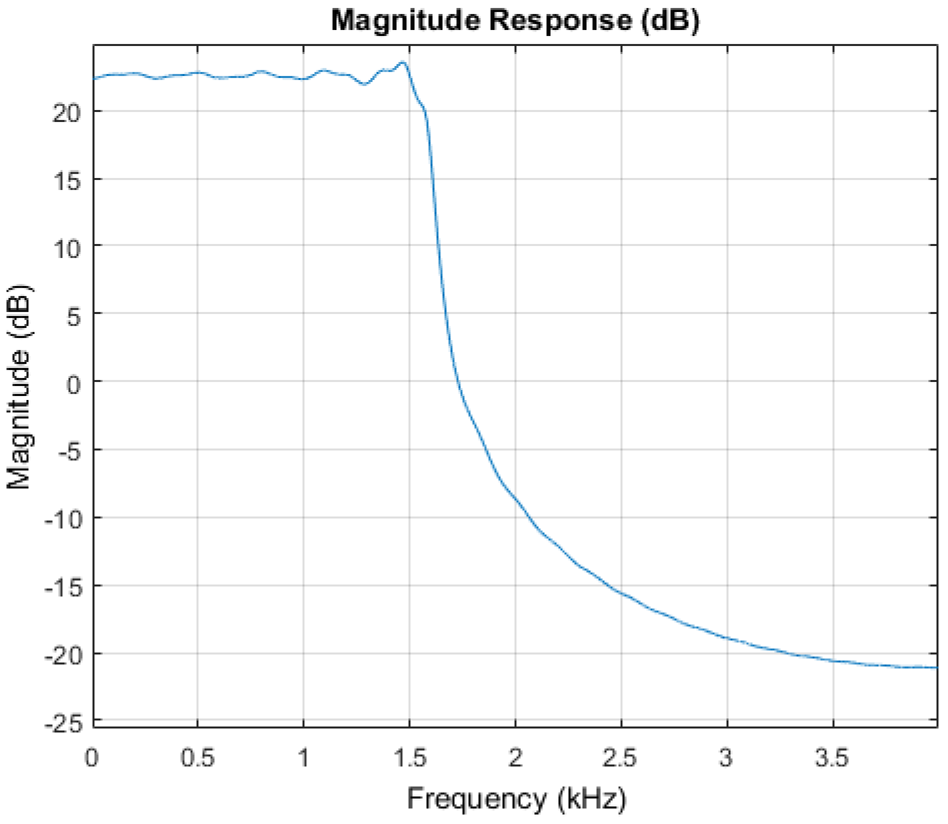
Lowpass filtering a waveform with two frequencies

Use an Allpole filter to apply a lowpass filter to a waveform with two sinusoidal frequencies.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
 t = (0:1000)./8e3;
 xin = sin(2*pi*1e3*t)+sin(2*pi*3e3*t);

 src = dsp.SignalSource(xin', 4);
 sink = dsp.SignalSink;
 allpole = dsp.AllpoleFilter;
 tt = (-25:25)';
 xsinc = 0.4*sinc(0.4*tt);
 asinc = lpc(xsinc,51);
 allpole.Denominator = asinc;

 sa = dsp.SpectrumAnalyzer('SampleRate',8e3,...
    'PlotAsTwoSidedSpectrum',false,...
    'OverlapPercent', 80,'PowerUnits','dBW',...
    'YLimits', [-150 50]);

while ~isDone(src)
      input = src();
      filteredOutput = allpole(input);
      sink(filteredOutput);
      sa(filteredOutput)
 end

 filteredResult = sink.Buffer;
 fvtool(allpole,'Fs',8000)
```


Spectrum Analyzer
File Tools View Playback Help
50
0
-50
-100
-150
dBW
0 0.5 1 1.5 2 2.5 3 3.5 4
Frequency (kHz)
Processing
RBW=3.9 Hz
Sample Rate=8 kHz
T=0.0765



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Allpole Filter` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `Denominator` property is tunable for code generation.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.FIRFilter | dsp.BiquadFilter | dsp.IIRFilter

Introduced in R2012b

freqz

System object: dsp.AllpoleFilter
Package: dsp

Frequency response

Syntax

```
[h,w] = freqz(allpole)
[h,w] = freqz(allpole,n)
[h,w] = freqz(allpole,Name,Value)
freqz(allpole)
```

Description

`[h,w] = freqz(allpole)` returns the complex, 8192–element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample.

`[h,w] = freqz(allpole,n)` returns the complex, n-element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[h,w] = freqz(allpole,Name,Value)` returns the frequency response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`freqz(allpole)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the filter System object `allpole`.

fvtool

System object: dsp.AllpoleFilter
Package: dsp

Open filter visualization tool

Syntax

```
fvtool(allpole)
fvtool(allpole,'Arithmetic',ARITH,...)
```

Description

`fvtool(allpole)` performs an analysis and computes the magnitude response of the filter System object `allpole`.

`fvtool(allpole,'Arithmetic',ARITH,...)` analyzes the filter System object `allpole`, based on the arithmetic specified in the ARITH input. ARITH can be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The 'Arithmetic' input is only relevant for the analysis of filter System objects.

impz

System object: dsp.AllpoleFilter
Package: dsp

Impulse response

Syntax

```
[h,t] = impz(allpole)
[h,t] = impz(allpole,Name,Value)
impz(allpole)
```

Description

`[h,t] = impz(allpole)` returns the impulse response `h`, and the corresponding time points `t` at which the impulse response of`allpole` is computed.

`[h,t] = impz(allpole,Name,Value)` returns the impulse response `h`, and the corresponding time points`t` , with additional options specified by one or more Name, Value pair arguments.

`impz(allpole)` uses FVTool to plot the impulse response of the filter System object `allpole`.

Note: You can use `impz` for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

---

phasez

System object: dsp.AllpoleFilter
Package: dsp

Unwrapped phase response

Syntax

```
[phi,w] = phasez(allpole)
[phi,w] = phasez(allpole,n)
[phi,w] = phasez(allpole,Name,Value)
phasez(allpole)
```

Description

`[phi,w] = phasez(allpole)` returns the 8192–element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample.

`[phi,w] = phasez(allpole,n)` returns the n-element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[phi,w] = phasez(allpole,Name,Value)` returns the phase response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`phasez(allpole)` uses FVTool to plot the phase response of the filter System object `allpole`.

reset

System object: dsp.AllpoleFilter
Package: dsp

Reset internal states of Allpole filter

Syntax

```
reset(allpole)
```

Description

`reset(allpole)` resets the filter states of the Allpole filter object, `allpole`, to their initial values of 0. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the allpole filter object to nonzero input data, the states may be nonzero. Invoking the `step` method again without first invoking the `reset` method may produce different outputs for an identical input.

Reset an Allpole Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
n = 0:100;
x = cos(0.2*pi*n)+sin(0.8*pi*n);
allpole = dsp.AllpoleFilter;
a = lpc(x,20);
allpole.Denominator = a;
y = allpole(x);
```

Filter states are nonzero. Filter the input again without resetting the filter states.

```
y1 = allpole(x);
isequal(y,y1)
```

```
ans =

  logical

   0
```

Now reset filter states to 0.

```
reset(allpole)
```

Filter the input.

```
y2 = allpole(x);
isequal(y,y2)
```

```
ans =

  logical

   1
```

step

System object: dsp.AllpoleFilter
Package: dsp

Filter input with Allpole filter object

Syntax

```
Y = step(allpole,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(allpole,X)` filters the real or complex input signal `X` using the Allpole filter, `allpole`, to produce the output `Y` . When the input data is of a fixed-point type, it must be signed. The Allpole filter object operates on each channel of the input signal independently over successive calls to step method.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.AnalyticSignal System object

Package: dsp

Analytic signals of discrete-time inputs

Description

The `AnalyticSignal` object computes analytic signals of discrete-time inputs. The real part of the analytic signal in each channel is a replica of the real input in that channel, and the imaginary part is the Hilbert transform of the input. In the frequency domain, the analytic signal doubles the positive frequency content of the original signal while zeroing-out negative frequencies and retaining the DC component. The object computes the Hilbert transform using an equiripple FIR filter.

To compute the analytic signal of a discrete-time input:

1 Define and set up your analytic signal calculation. See "Construction" on page 4-57.
2 Call step to compute the analytic signal according to the properties of `dsp.AnalyticSignal`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`anaSig = dsp.AnalyticSignal` returns an analytic signal object, `anaSig`, that computes the complex analytic signal corresponding to each channel of a real M-by-N input matrix.

`anaSig = dsp.AnalyticSignal('PropertyName',PropertyValue,...)` returns an analytic signal object, `anaSig`, with each specified property set to the specified value.

`anaSig = dsp.AnalyticSignal(order,'PropertyName',PropertyValue,...)` returns an analytic signal object, `anaSig`, with the `FilterOrder` on page 4-58 property set to `order` and other specified properties set to the specified values.

Properties

`FilterOrder`

Filter order used to compute Hilbert transform

Specify the order of the equiripple FIR filter used in computing the Hilbert transform as an even integer scalar. The default is `100`.

Methods

reset	Reset internal states of analytic signal object
step	Analytic signal

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Compute The Analytic Signal

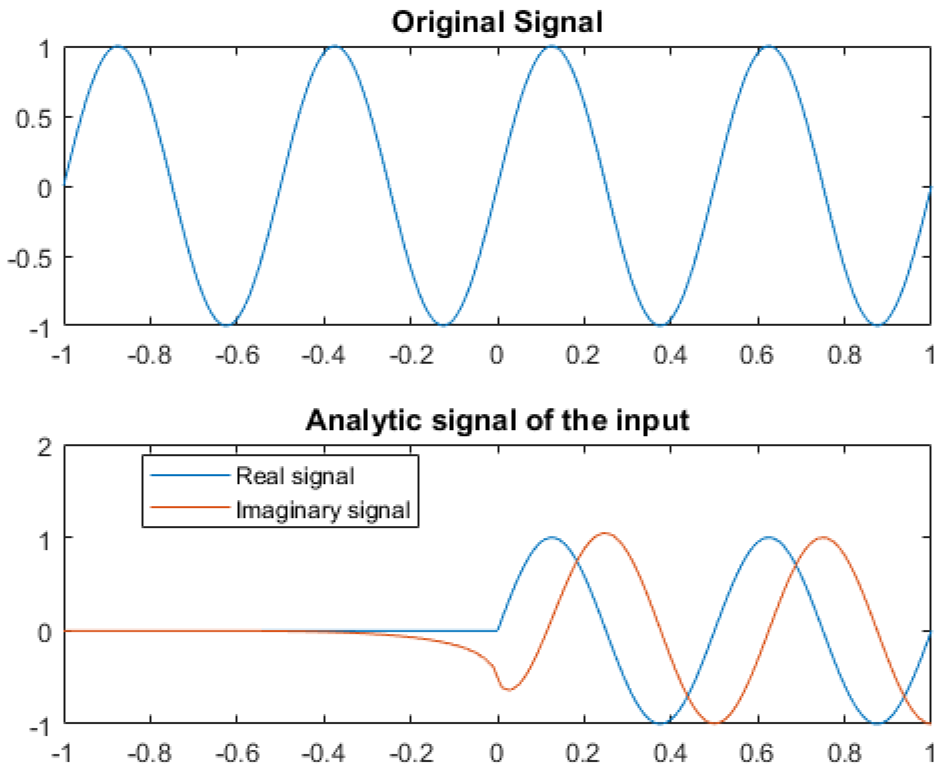
Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Compute the analytic signal of a sinusoidal input.

```
t = (-1:0.01:1)';
x = sin(4*pi*t);
anaSig = dsp.AnalyticSignal(200);
y = anaSig(x);
```

View the analytic signal.

```
subplot(2,1,1), plot(t, x);
title('Original Signal');
subplot(2,1,2), plot(t, [real(y) imag(y)]);
title('Analytic signal of the input')
legend('Real signal','Imaginary signal',...
    'Location','best');
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Analytic Signal` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.IFFT | dsp.FFT

Introduced in R2012a

reset

System object: dsp.AnalyticSignal
Package: dsp

Reset internal states of analytic signal object

Syntax

`reset(anaSig)`

Description

`reset(anaSig)` sets the internal states of the `AnalyticSignal` object `anaSig` to their initial values.

step

System object: dsp.AnalyticSignal
Package: dsp

Analytic signal

Syntax

`Y = step(anaSig,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(anaSig,X)` computes the analytic signal, `Y`, of the M-by-N input matrix `X`, according to the equation

$$\mathbf{Y} = \mathbf{X} + jH\{\mathbf{X}\}$$

where j is the imaginary unit and $H\{\mathbf{X}\}$ denotes the Hilbert transform.

Each of the N columns in `X` contains M sequential time samples from an independent channel. The method computes the analytic signal for each channel.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change

nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.ArrayPlot System object

Package: dsp

Display vectors or arrays

Description

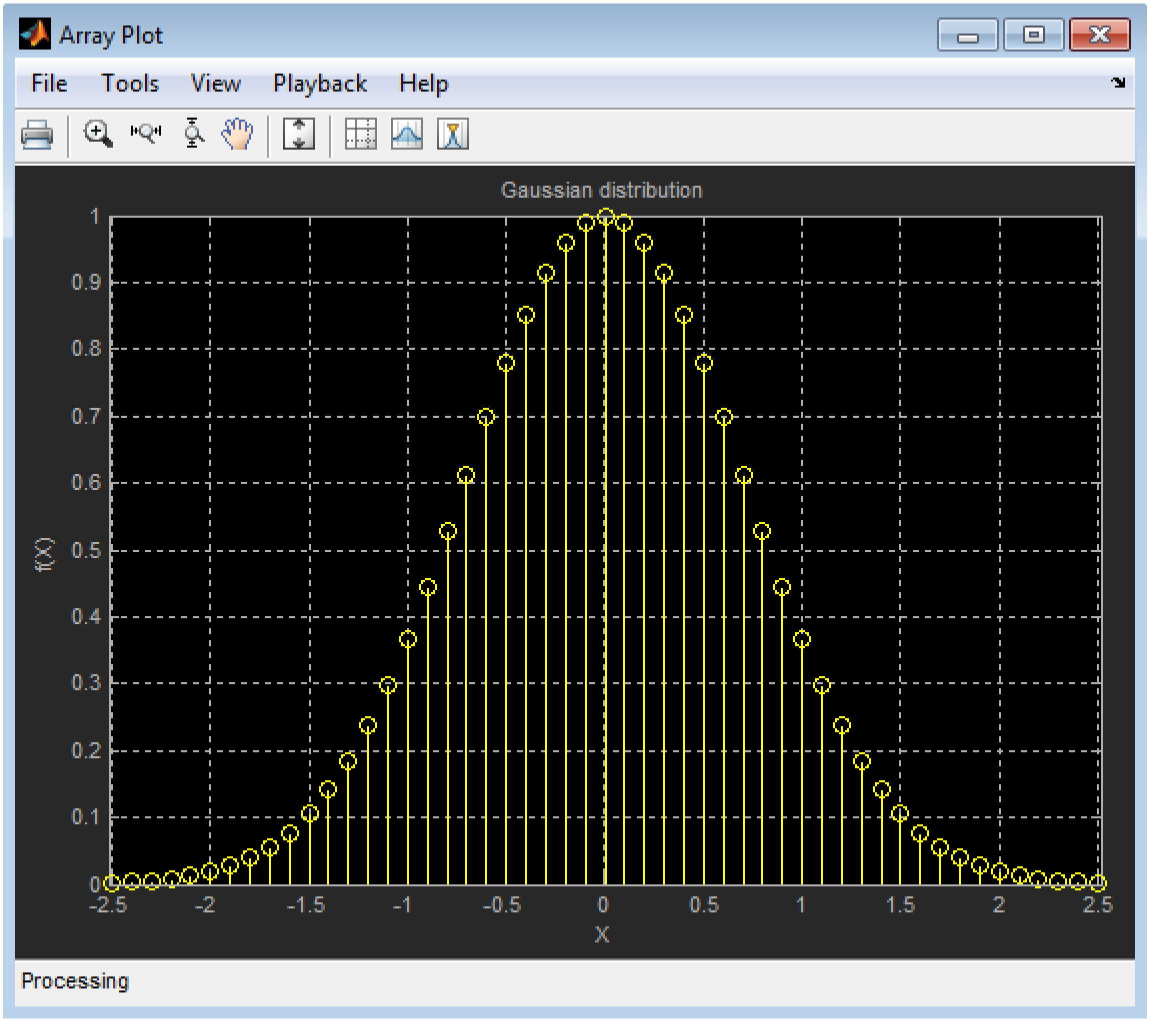
The `ArrayPlot` object displays vectors or arrays.

To display vectors or array in Array Plot:

1. Define and set up your Array Plot. See "Construction" on page 4-67.
2. Call step to display the vectors or arrays in the Array Plot figure. The behavior of `step` is specific to each object in the toolbox.

Use the MATLAB `clear` function to close the Array Plot figure window and clear its associated data. Use the hide method to hide the ArrayPlot window and the show method to make it visible.

Use the MATLAB `mcc` function to compile code containing an Array Plot.



See the following sections for more information on the Array Plot Graphical User Interface:

- "Signal Display" on page 4-76
- "Toolbar" on page 4-79
- "Measurements Panels" on page 4-83
- "Visuals — Array Plot Properties" on page 4-90

- “Style Dialog Box” on page 4-96
- “Tools — Plot Navigation Properties” on page 4-97

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`plot = dsp.ArrayPlot` creates an Array Plot System object, `plot`. This object displays vectors or arrays.

`plot = dsp.ArrayPlot('Name',Value,...)` creates an Array Plot System object, `plot`, with each specified property *`Name`* set to the specified value. You can specify *`Name`*–*`Value`* arguments in any order.

Properties

`ChannelNames`

Input channel names

Specify as a cell array of character vectors the input channels names to display in the legend, which is visible only when the `ShowLegend` property is `true`. The channel names are also used in the **Style** dialog, and in some **Measurements** panels. The default is an empty cell array, which labels the channels as `Channel 1`, `Channel 2`, etc. This property is Tunable (Simulink).

Default: empty cell array

`CustomXData`

X-data values

Specify the desired *x*-data values as a row or column vector of length equal to the frame length of the inputs. This property is used only if the `XDataMode` property is set to

'Custom'. If you use the default (empty vector) value, the x-data is uniformly spaced and set to (0:L-1), where L is the frame length.

An example of a Name-Value pair to specify logarithmic x-data scaling is 'CustomXData', logspace(0, log10(44100/2), 1024).

Default: empty vector

MaximizeAxes

Maximize axes control

Specify whether to display the scope in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. To conserve space, labels do not appear in the display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- Auto — In this mode, the axes appear maximized only if the Title and YLabel properties are empty. If you enter any value for either of these properties, the axes are not maximized.
- On — In this mode, the axes appear maximized. Any values entered into the Title and YLabel properties are hidden.
- Off — In this mode, none of the axes appear maximized.

This property is Tunable (Simulink).

Default: 'Auto'

Name

Caption to display on Scope window

Specify as a character vector the caption to display on the scope window. This property is Tunable (Simulink).

Default: 'Array Plot'

NumInputPorts

Number of input signals

Specify the number of input signals to display on the scope as a positive integer. You must invoke the `step` method with the same number of inputs as the value of this property.

Default: `1`

`PlotAsMagnitudePhase`

Plot signal as Magnitude-and-Phase

When you set this property to `true`, the scope plots the magnitude and phase of the input signal on two separate axes. When you set this property to `false`, the scope plots the real and imaginary parts of the input signal on two separate axes. This property is particularly useful for complex-valued input signals. When the input signal is real-valued and you select this check box, the phase will be 0 degrees when the amplitude of the input signal is nonnegative and 180 degrees when the input signal is negative. This property is Tunable (Simulink).

Default: `false`

`PlotType`

Option to control the type of plot

Specify the type of plot to use for all the input signals displayed in the scope window.

- When you set this property to `'Stem'` , the scope displays the input signal as circles with vertical lines extending down to the *x*-axis at each of the sampled values . This approach is similar to the functionality of the MATLAB `stem` function.
- When you set this property to `'Line'` , the scope displays the input signal as lines connecting each of the sampled values. This approach is similar to the functionality of the MATLAB `line` or `plot` function.
- When you set this property to `'Stairs'`, the scope displays the input signal as a *stairstep* graph. A stairstep graph is made up of only horizontal lines and vertical lines. Each horizontal line represents the signal value for a discrete sample period and is connected to two vertical lines. Each vertical line represents a change in values occurring at a sample. This approach is equivalent to the MATLAB `stairs` function. Stairstep graphs are useful for drawing time history graphs of digitally sampled data.

This property is Tunable (Simulink).

Default: `'Stem'`

Position

Scope window position in pixels

Specify, in pixels, the size and location of the scope window as a 4-element double vector of the form [left bottom width height]. You can place the scope window in a specific position on your screen by modifying the values to this property. This property is Tunable (Simulink).

Default: The default depends on your screen resolution. By default, the Scope window appears in the center of your screen with a width of `800` pixels and height of `450` pixels.

ReduceUpdates

Reduce updates to improve performance

When you set this property to `true`, the scope logs data for later use and updates the window periodically. When you set this property to `false`, the scope updates every time the `step` method is called. The simulation speed is faster when this property is set to `true`. This property is Tunable (Simulink).

You can also modify this property from the scope window. Opening the **Playback** menu and clearing the **Reduce Updates to Improve Performance** check box is the same as setting this property to `false`.

Default: `true`

SampleIncrement

Sample increment of input

Specify the spacing between samples along the *x*-axis as a finite numeric scalar. The input signal is only *y*-axis data. *x*-axis data is set automatically based on the `Xoffset` and `SampleIncrement` properties. For example, when `Xoffset` is 0 and `SampleIncrement` is 1, the x-data for the input signal is set to 0, 1, 2, 3, 4, etc. If you set `SampleIncrement` to 0.25, the x-axis data becomes 0, 0.25, 0.5, 0.75, 1, etc.

Default: `1`

ShowGrid

Option to enable or disable grid display

When you set this property to `true`, the grid appears. When you set this property to `false`, the grid is hidden. This property is Tunable (Simulink).

Default: `true`

`ShowLegend`

Source of legend

When you set this property to `true`, the scope displays a legend with the input channels labels. When you set this property to `false`, the scope does not display a legend. This property is Tunable (Simulink).

Default: `false`

`Title`

Display title

Specify the display title as a character vector. Enter `%<SignalLabel>` to use the signal labels in the Simulink Model as the axes titles. This property is Tunable (Simulink).

Default: `''`

`XDataMode`

Source of the *x*-data spacing

Specify whether to use the `SampleIncrement` and `XOffset` values, or specify your own custom spacing. Valid values are `'SampleIncrement and XOffset'` and `'Custom'`. If you specify `'Custom'`, you also must specify the `CustomXData` property values.

Default: `'SampleIncrement and XOffset'`

`XLabel`

The label for the *x*-axis

Specify the text for the scope to display below the *x*-axis. This property is Tunable (Simulink).

Default: `''`

XOffset

x-axis display offset

Specify the offset to apply to the *x*-axis. This property is a numeric scalar. This property is Tunable (Simulink).

Default: `0`

XScale

x-axis scale

Specify `'Linear'` or `'Log'` as the *x*-axis scale . If `XOffset` is a negative value, you cannot set `XScale` to `'Log'`. Tunable (Simulink)

Default: `'Linear'`

YLabel

The label for the *y*-axis

Specify the text for the scope to display to the left of the *y*-axis. Tunable (Simulink)

Default: `'Amplitude'`

YLimits

The limits for the *y*-axis

Specify the *y*-axis limits as a 2-element numeric vector, `[ymin ymax]`. This property is Tunable (Simulink).

Default: `[-10, 10]`

Methods

hide	Hide scope window

reset	Reset internal states of scope object
show	Make scope window visible
step	Display signal in scope figure

Examples

The following examples illustrate how to use the Array Plot object to view a variety of input signals.

Plot a Gaussian Distribution

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

View a *Gaussian*, or normal, distribution on the Array Plot figure.

Create a new Array Plot object.

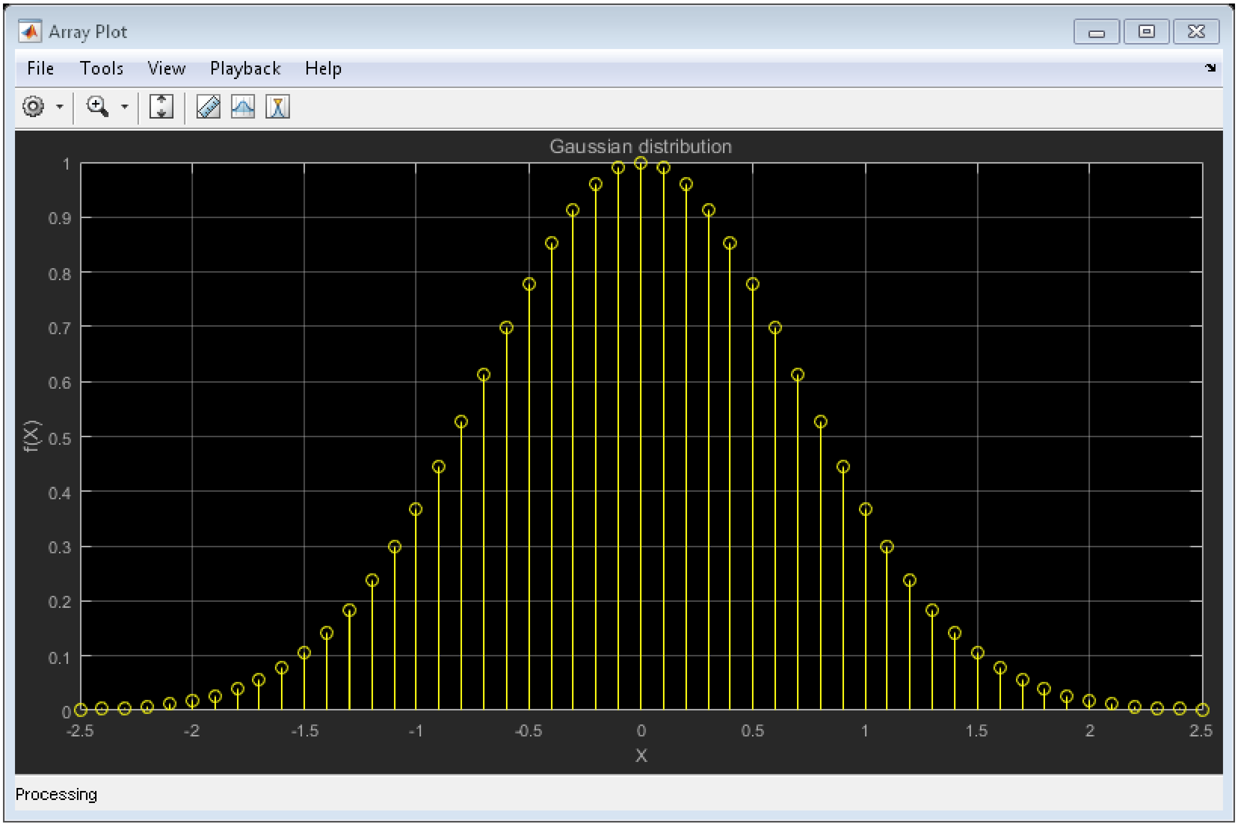
```
plot = dsp.ArrayPlot;
```

Configure the properties of the Array Plot object for a Gaussian distribution.

```
plot.YLimits = [0 1];
plot.XOffset = -2.5;
plot.SampleIncrement = 0.1;
plot.Title = 'Gaussian distribution';
plot.XLabel = 'X';
plot.YLabel = 'f(X)';
```

Call the `step (dsp.ArrayPlot)` method to plot a Gaussian distribution.

```
plot(exp(-(-2.5:.1:2.5).*(-2.5:.1:2.5))');
```



Plot Changing Filter Weights

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

View Least Mean Squares (LMS) adaptive filter weights on the Array Plot figure. Watch the filter weights change as they adapt to filter a noisy input signal.

Create an LMS adaptive filter System object.

```
lmsFilter = dsp.LMSFilter(40,'Method', 'Normalized LMS', 'StepSize', .002);
```

Create and configure an audio file reader System object to read the input signal from the specified audio file.

```
signalSource = dsp.AudioFileReader('dspafxf_8000.wav',  ...
    'SamplesPerFrame', 40, ...
    'PlayCount', Inf,...
    'OutputDataType', 'double');
```

Create and configure a fixed-point Finite Impulse Response (FIR) digital filter System object to filter random white noise, creating colored noise.

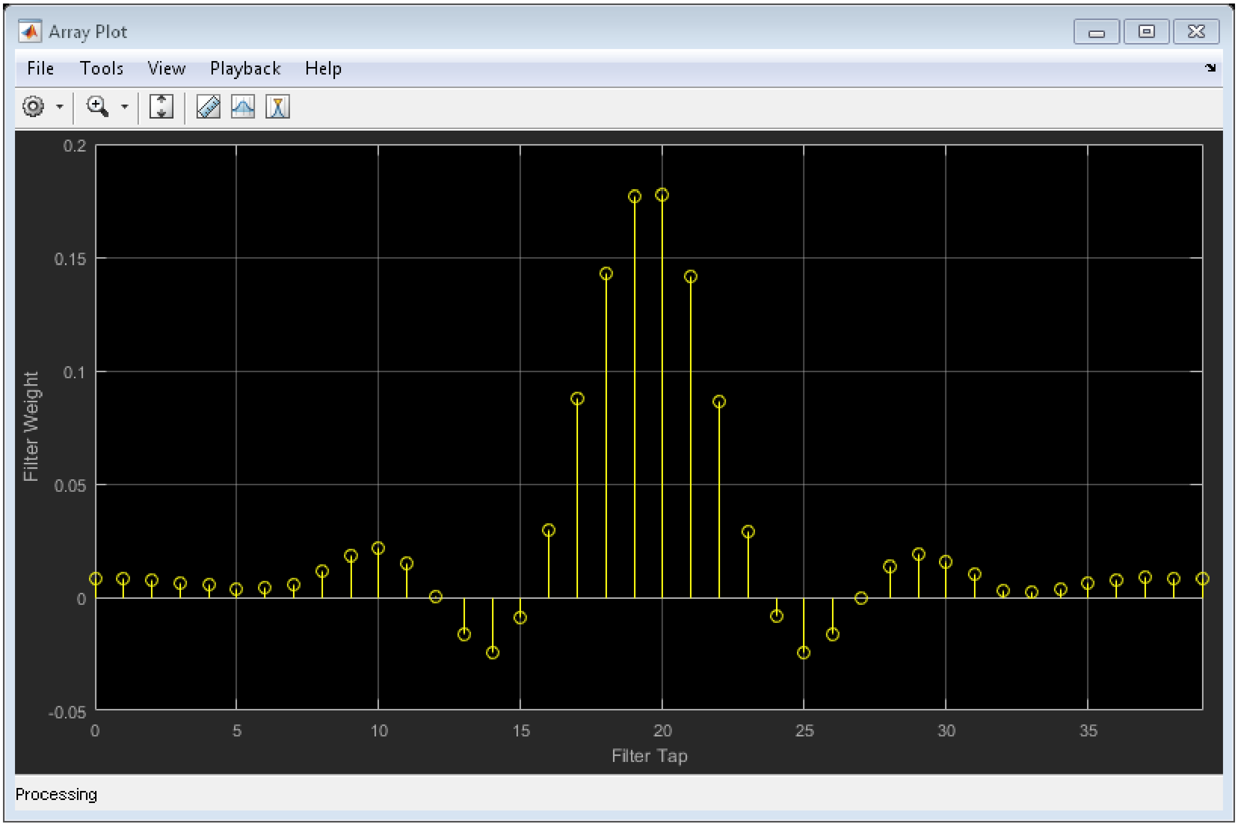
```
firFilter = dsp.FIRFilter('Numerator', fir1(39, .25));
```

Create and configure an Array Plot System object to display the adaptive filter weights.

```
plot = dsp.ArrayPlot('XLabel', 'Filter Tap', ...
    'YLabel', 'Filter Weight', ...
    'YLimits', [-.05 .2]');
```

Plot the LMS filter weights as they adapt to a desired signal. Read from the audio file, produce random data, and filter the random data. Update the filter weights and plot the filter weights.

```
numplays = 0;
while numplays < 3
    [y, eof]    = signalSource();
    noise       = rand(40,1);
    noisefilt   = firFilter(noise);
    desired     = y + noisefilt;
    [~, ~, wts] = lmsFilter(noise, desired);
    plot(wts);
    numplays = numplays + eof;
end
```

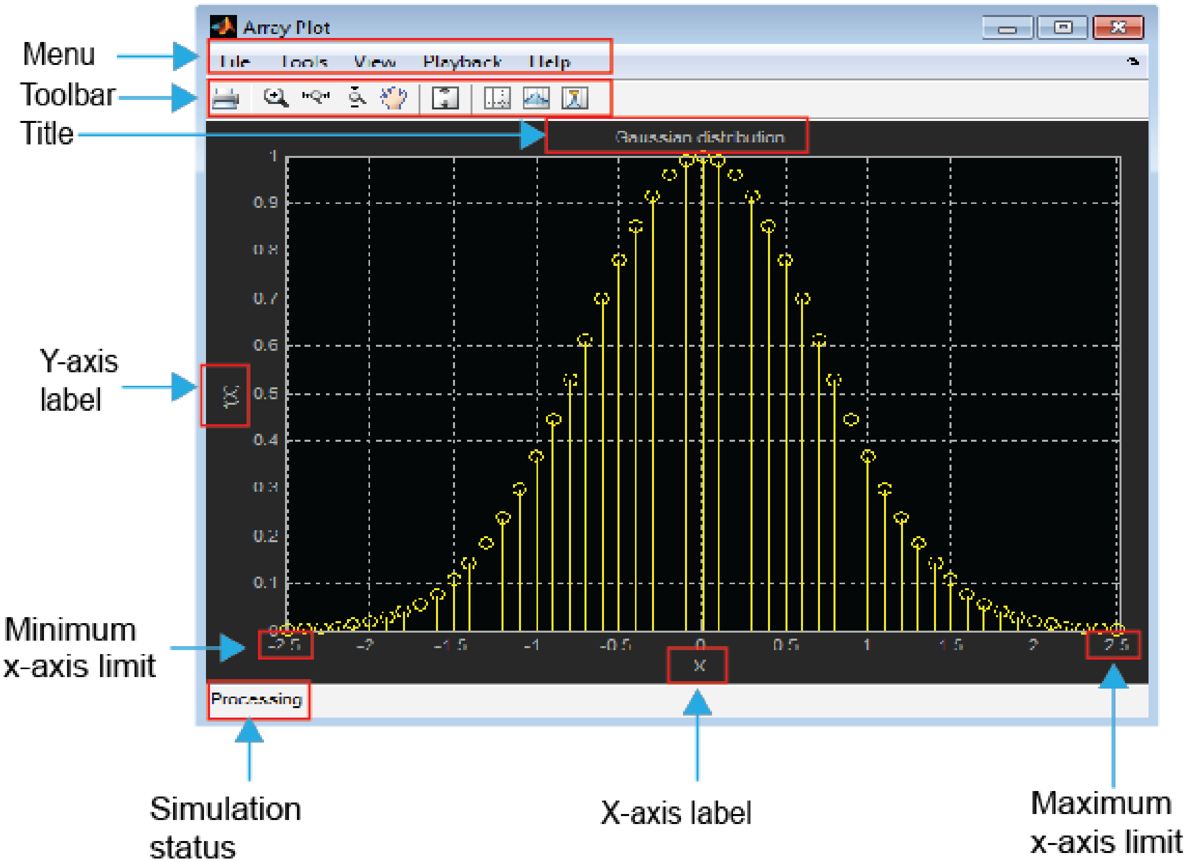


Signal Display

Array Plot uses the `XOffset` on page 4-72 and `SampleIncrement` on page 4-70 properties to determine the time range. To change the **X-offset** setting in the Array Plot figure, select **View** > **Configuration Properties**. The Visuals — Array Plot Properties dialog box appears. Go to the **Main** tab, and modify the value for the **X-offset**.

The span of the x-axis data is related to the `SampleIncrement` property by the equation $x_{span} = \texttt{SampleIncrement} \times (\text{length}(x) - 1)$. Suppose you set `SampleIncrement` on page 4-70 to `0.1` and the input signal data has 51 samples. The scope displays values on the x-axis from `0` to `5`. If you also set the **X-offset** to `–2.5`, the scope displays values on the x-axis from `-2.5` to `2.5`. The values on the x-axis of the scope display remain the same throughout simulation.

The following figure highlights the important aspects of the Array Plot window.



Indicators

- **Minimum x-axis limit** — Array Plot sets the minimum x-axis limit using the value of the **X-offset** parameter on the **Main** tab of the Visuals—Array Plot Properties dialog box.

- **Maximum x-axis limit** — Array Plot sets the maximum x-axis limit by summing the value of **X-offset** parameter with the span of x-axis values. The equation $x_{span} = \texttt{SampleIncrement} \times (\text{length}(x) - 1)$ defines the relationship of the span of the x-axis data to the `SampleIncrement` on page 4-70 property.

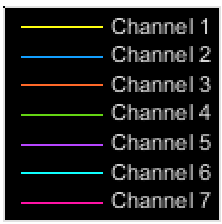
- *Simulation status* — Provides the current status of the model simulation. The status can be either of the following conditions:

  - `Processing` — Occurs after you run the `step` method and before you run the `release` method.
  - `Stopped` — Occurs after you construct the scope object and before you first run the `step` method. This status also occurs after you run the `release` method.

  The *Simulation status* is part of the *Status Bar* in the scope window. You can choose to hide or display the entire *Status Bar*. From the scope menu, select **View** > **Status Bar**.

Multiple Signal Names and Colors

By default, if the input signal has multiple channels, the scope uses an index number to identify each channel of that signal. For example, a 2-channel signal would have the following default names in the channel legend: `Channel 1`, `Channel 2`. To show the legend, select **View** > **Configuration Properties**, click the **Display** tab, and select the **Show Legend** check box. If there are a total of 7 input channels, the following legend appears in the display.



By default, the scope has a black axes background and chooses line colors for each channel in a manner similar to the Simulink Scope block. When the scope axes background is black, it assigns each channel of each input signal a line color in the order shown in the above figure.

If there are more than 7 channels, then the scope repeats this order to assign line colors to the remaining channels. To choose line colors for each channel, change the axes background color to any color except black. To change the axes background color to white, select **View** > **Style**, click the Axes background color button (), and select white from the color palette. Run the simulation again. The following legend appears in the display. This is the color order when the background is not black.

Toolbar

The Array Plot toolbar contains the following buttons.

Print, Settings, and Properties Buttons

Button	Menu Location	Shortcut Keys	Description
	**File** > **Print**	**Ctrl+P**	Print the current scope window. To print the current scope window to a figure rather than sending it to your printer, select **File** > **Print to figure**.

Zoom and Axes Control Buttons

Button	Menu Location	Shortcut Keys	Description
	**Tools** > **Zoom In**	N/A	When this tool is active, you can zoom in on the scope window. To do so, click in the center of your area of interest, or click and drag your cursor to draw a rectangular area of interest inside the scope window.

Button	Menu Location	Shortcut Keys	Description
	**Tools >** **Zoom X**	N/A	You access the Zoom X button from the menu under the Zoom In icon. When this tool is active, you can zoom in on the *x*-axis. To do so, click inside the scope window, or click and drag your cursor along the *x*-axis over your area of interest.
	**Tools >** **Zoom Y**	N/A	You access the Zoom Y button from the menu under the Zoom In icon. When this tool is active, you can zoom in on the *y*-axis. To do so, click inside the scope window, or click and drag your cursor along the *y*-axis over your area of interest.
	**Tools >** **Pan**	N/A	You access the Pan button from the menu under the Zoom In icon. When this tool is active, you can pan on the scope window. To do so, click in the center of your area of interest and drag your cursor to the left, right, up, or down, to move the position of the display.
	**Tools >** **Scale Y-Axis Limits**	**Ctrl+A**	Click this button to scale the axes in the active scope window. Alternatively, you can enable automatic axes scaling by selecting one of the following options from the **Tools** menu: • **Automatically Scale Axes Limits** — When you select this option, the scope scales the axes as needed during simulation. • **Scale Axes Limits after 10 Updates** — When you select this option, the scope scales the axes after 10 updates. The scope does not scale the axes again during the simulation. • **Scale Axes Limits at Stop** — When you select this option, the scope scales the axes each time the simulation is stopped.

Button	Menu Location	Shortcut Keys	Description
	**Tools > Scale X-Axis Limits**	N/A	You access the Scale X-Axis Limits button from the menu under the current Axis Limits icon. Click this button to scale the axes in the X direction in the active scope window. Alternatively, you can enable automatic axes scaling by selecting one of the following options from the **Tools** menu: • **Automatically Scale Axes Limits** — When you select this option, the scope scales the axes as needed during simulation. • **Scale Axes Limits after 10 Updates** — When you select this option, the scope scales the axes after 10 updates. The scope does not scale the axes again during the simulation. • **Scale Axes Limits at Stop** — When you select this option, the scope scales the axes each time the simulation is stopped.

Button	Menu Location	Shortcut Keys	Description
	**Tools** > **Scale X & Y Axes Limits**	N/A	You access the Scale X & Y Axes Limits button from the menu under the current Axis Limits icon. Click this button to scale the axes in both the X and Y directions in the active scope window. Alternatively, you can enable automatic axes scaling by selecting one of the following options from the **Tools** menu: • **Automatically Scale Axes Limits** — When you select this option, the scope scales the axes as needed during simulation. • **Scale Axes Limits after 10 Updates** — When you select this option, the scope scales the axes after 10 updates. The scope does not scale the axes again during the simulation. • **Scale Axes Limits at Stop** — When you select this option, the scope scales the axes each time the simulation is stopped.

Measurements Buttons

	**Tools** > **Measurements** > **Cursor Measurements**	N/A	Open or close the **Cursor Measurements** panel. This panel puts screen cursors on the display. See the "Cursor Measurements Panel" on page 4-85 section for more information.
	**Tools** > **Measurements** > **Signal Statistics**	N/A	Open or close the **Signal Statistics** panel. This panel displays the maximum, minimum, peak-to-peak difference, mean, median, RMS values of a selected signal, and the times at which the maximum and minimum occur. See the "Signal Statistics Panel" on page 4-86 section for more information.

	**Tools** > **Measurements** > **Peak Finder**	N/A	Open or close the **Peak Finder** panel. This panel displays maxima and the frequencies at which they occur, allowing the settings for peak threshold, maximum number of peaks, and peak excursion to be modified. See the "Peak Finder Panel" on page 4-88 section for more information.

Note: The zoom buttons do not change the settings related to span for the scope. These buttons are purely graphical.

You can control whether this toolbar appears in the Array Plot window. From the menu, select **View** > **Toolbar**.

Measurements Panels

The Measurements panels are the panels that appear to the right side of the Array Plot figure. These panels are labeled **Trace selection**, **Cursor Measurements**, **Signal Statistics**, and **Peak Finder**.

Measurements Panel Buttons

Each of the Measurements panels contains the following buttons that enable you to modify the appearance of the current panel.

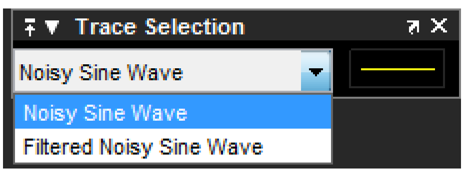
Button	Description
	Move the current panel to the top. When you are displaying more than one panel, this action moves the current panel above all the other panels.
	Collapse the current panel. When you first enable a panel, by default, it displays one or more of its panes. Click this button to hide all of its panes to conserve space. After you click this button, it becomes the expand button .
	Expand the current panel. This button appears after you click the collapse button to hide the panes in the current panel. Click this button to display

Button	Description
	the panes in the current panel and show measurements again. After you click this button, it becomes the collapse button again.
	Undock the current panel. This button lets you move the current panel into a separate window that can be relocated anywhere on your screen. After you click this button, it becomes the dock button in the new window.
	Dock the current panel. This button appears only after you click the undock button. Click this button to put the current panel back into the right side of the Scope window. After you click this button, it becomes the undock button again.
	Close the current panel. This button lets you remove the current panel from the right side of the Scope window.

Some panels have their measurements separated by category into a number of panes. Click the pane expand button to show each pane that is hidden in the current panel. Click the pane collapse button to hide each pane that is shown in the current panel.

Trace Selection Panel

When you use the scope to view multiple signals, the Trace Selection panel appears if you have more than one signal displayed and you click any of the other Measurements panels. The Measurements panels display information about only the signal chosen in this panel. Choose the signal name for which you would like to display time domain measurements. See the following figure.



You can choose to hide or display the **Trace Selection** panel. In the Scope menu, select **Tools** > **Measurements** > **Trace Selection**.

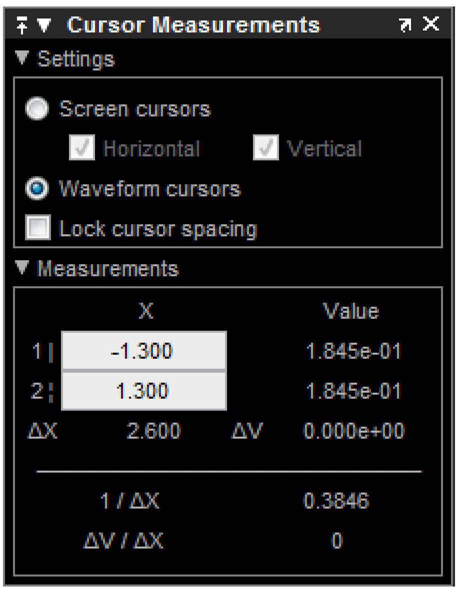
Cursor Measurements Panel

The **Cursor Measurements** panel displays screen cursors. The panel provides two types of cursors for measuring signals. Waveform cursors are vertical cursors that track along the signal. Screen cursors are both horizontal and vertical cursors that you can place anywhere in the display.

Note: If a data point in your signal has more than one value, the cursor measurement at that point is undefined and no cursor value is displayed.

In the Scope menu, select **Tools** > **Measurements** > **Cursor Measurements**.

Alternatively, in the Scope toolbar, click the Cursor Measurements  button.



The **Cursor Measurements** panel is separated into two panes, labeled **Settings** and **Measurements**. You can expand each pane to see the available options.

You can use the mouse or the left and right arrow keys to move vertical or waveform cursors and the up and down arrow keys for horizontal cursors.

Settings Pane

The **Settings** pane enables you to modify the type of screen cursors used to calculate *x*-axis and *y*-axis value measurements.

- **Screen Cursors**— Shows screen cursors.
- **Horizontal**— Shows horizontal screen cursors.
- **Vertical**— Shows vertical screen cursors.
- **Waveform Cursors**— Shows cursors that attach to the input signals.
- **Lock Cursor Spacing**— Locks the time difference between the two cursors.

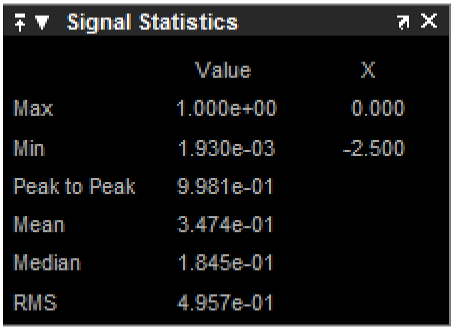
Measurements Pane

The **Measurements** pane shows the time and value measurements.

- **1 |**— Shows or enables you to modify the time or value at cursor number one, or both.
- **2 :**— Shows or enables you to modify the time or value at cursor number two, or both.
- **ΔX**— Shows the absolute value of the difference in the*x*-axis values between cursor number one and cursor number two.
- **ΔV**— Shows the absolute value of the difference in the *y*-axis values, or signal amplitudes, between cursor number one and cursor number two.
- **1/ΔX**— Shows the rate, the reciprocal of the absolute value of the difference in the *x*-axis values between cursor number one and cursor number two.
- **ΔV/ΔX**— Shows the slope, the ratio of the absolute value of the difference in the *y*-axis values between cursors to the absolute value of the difference in the *x*-axis values between cursors.

Signal Statistics Panel

The **Signal Statistics** panel displays the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal. It also shows the *x*-axis indices at which the maximum and minimum values occur. You can choose to hide or display the **Signal Statistics** panel. In the scope menu, select **Tools** > **Measurements** > **Signal Statistics**. Alternatively, in the scope toolbar, click the Signal Statistics  button.



Signal Statistics Measurements

The **Signal Statistics** panel shows statistics about the portion of the input signal within the *x*-axis and *y*-axis limits of the active display. The statistics shown are:

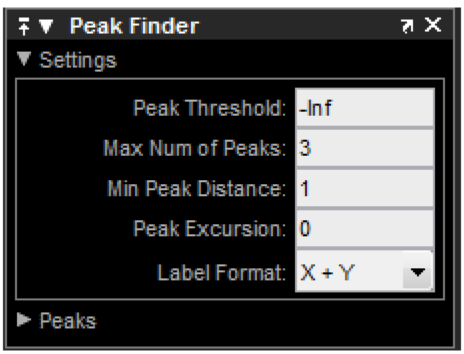
- **Max** — Shows the maximum or largest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `max` function reference.
- **Min** — Shows the minimum or smallest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `min` function reference.
- **Peak to Peak** — Shows the difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `peak2peak` function reference.
- **Mean** — Shows the average or mean of all the values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `mean` function reference.
- **Median** — Shows the median value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `median` function reference.
- **RMS** — Shows the difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `rms` function reference.

When you use the zoom options in the scope, the Signal Statistics measurements automatically adjust to the time range shown in the display. In the scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the *x*-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one pulse to make the **Signal Statistics** panel display information about only that particular pulse.

The Signal Statistics measurements are valid for any units of the input signal. The letter after the value associated with each measurement represents the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Peak Finder Panel

The **Peak Finder** panel displays the maxima, showing the *x*-axis values at which they occur. Peaks are defined as a local maximum where lower values are present on both sides of a peak. Endpoints are not considered to be peaks. This panel allows you to modify the settings for peak threshold, maximum number of peaks, and peak excursion. You can choose to hide or display the **Peak Finder** panel. In the scope menu, select **Tools** > **Measurements** > **Peak Finder**. Alternatively, in the scope toolbar, select the Peak Finder button.

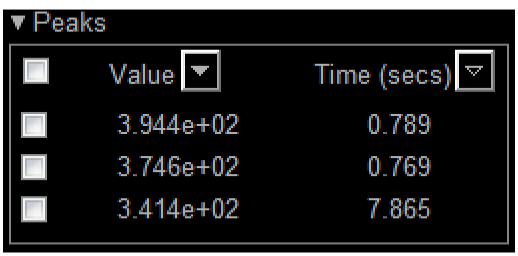


The **Peak finder** panel is separated into two panes, labeled **Settings** and **Peaks**. You can expand each pane to see the available options.

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input signal. For more information on the algorithms this pane uses, see the Signal Processing Toolbox `findpeaks` function reference.

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the `MINPEAKHEIGHT` parameter, which you can set when you run the `findpeaks` function.
- **Max Num of Peaks** — The maximum number of peaks to show. The value you enter must be a scalar integer from 1 through 99. This setting is equivalent to the `NPEAKS` parameter, which you can set when you run the `findpeaks` function.
- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the `MINPEAKDISTANCE` parameter, which you can set when you run the `findpeaks` function.
- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples The peak excursion setting is equivalent to the `THRESHOLD` parameter, which you can set when you run the `findpeaks` function.
- **Label Format** — The coordinates to display next to the calculated peak values on the plot. To see peak values, expand the **Peaks** pane and select the check boxes associated with individual peaks of interest. By default, both *x*-axis and *y*-axis values are displayed on the plot. Select which axes values you want to display next to each peak symbol on the display.
  - `X+Y` — Display both *x*-axis and *y*-axis values.
  - `X` — Display only *x*-axis values.
  - `Y` — Display only *y*-axis values.

The **Peaks** pane displays all of the largest calculated peak values. It also shows the coordinates at which the peaks occur, using the parameters you define in the **Settings** pane. You set the **Max Num of Peaks** parameter to specify the number of peaks shown in the list.



The numerical values displayed in the **Value** column are equivalent to the `pks` output argument returned when you run the `findpeaks` function. The numerical values

displayed in the second column are similar to the `locs` output argument returned when you run the `findpeaks` function.

The Peak Finder displays the peak values in the **Peaks** pane. By default, the **Peak Finder** panel displays the largest calculated peak values in the **Peaks** pane in decreasing order of peak height. Use the sort descending button () to rearrange the category and order by which Peak Finder displays peak values. Click this button again to sort the peaks in ascending order instead. When you do so, the arrow changes direction to become the sort ascending button (). A filled sort button indicates that the peak values are currently sorted in the direction of the button arrow. If the sort button is not filled (), then the peak values are sorted in the opposite direction of the button arrow. The **Max Num of Peaks** parameter still controls the number of peaks listed.

Use the check boxes to control which peak values are shown on the display. By default, all check boxes are cleared and the **Peak Finder** panel hides all the peak values. To show all the peak values on the display, select the check box in the top-left corner of the **Peaks** pane. To hide all the peak values on the display, clear this check box. To show an individual peak, select the check box directly to the left of its **Value** listing. To hide an individual peak, clear the check box directly to the left of its **Value** listing.

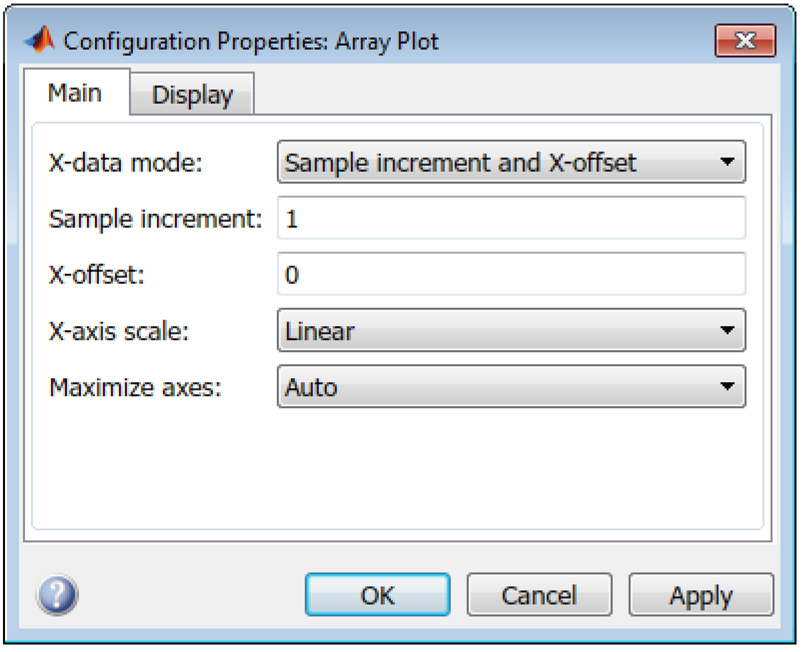
The Peaks are valid for any units of the input signal. The letter after the value associated with each measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Visuals — Array Plot Properties

The Visuals—Array Plot Properties dialog box controls various properties about the Array Plot display. From the Array Plot menu, select **View** > **Configuration Properties** to open this dialog box.

Main Pane

The **Main** pane of the Visuals—Array Plot Properties dialog box appears as follows.



Sample increment

Specify the spacing between samples along the *x*-axis as a finite numeric scalar value.

X-offset

Specify the offset to apply to the *x*-axis. This property is a numeric scalar. This property is Tunable (Simulink).

X-axis scale

Specify `'Linear'` or `'Log'` as the *x*-axis scale . If `XOffset` is a negative value, you cannot set `XScale` to `'Log'`. Tunable (Simulink)

Maximize axes

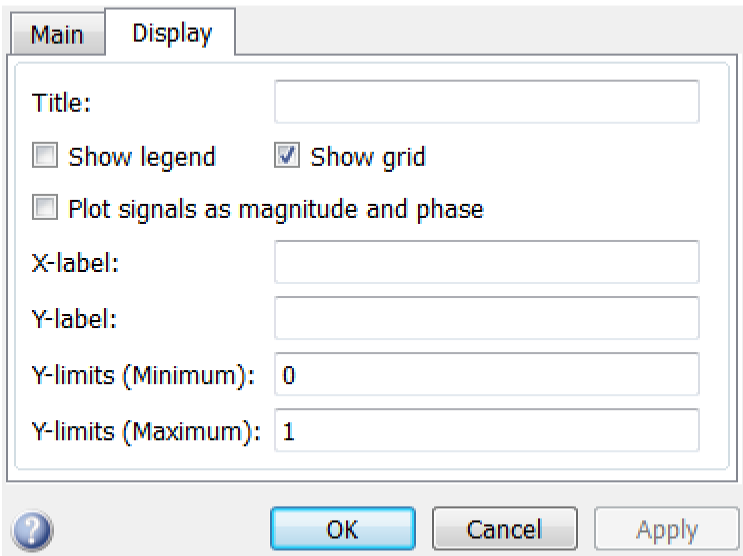
Specify whether to display the scope in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. To conserve space, labels do not appear in the display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- Auto — In this mode, the axes appear maximized only if the Title and YLabel properties are empty. If you enter any value for either of these properties, the axes are not maximized.
- On — In this mode, the axes appear maximized. Any values entered into the Title and YLabel properties are hidden.
- Off — In this mode, none of the axes appear maximized.

This property is Tunable (Simulink).

Display Pane

The **Display** pane of the Visuals—Array Plot Properties dialog box appears as follows.



Title

Specify the display title as a character vector. Enter %<SignalLabel> to use the signal labels in the Simulink Model as the axes titles. This property is Tunable (Simulink).

Show legend

Select this check box to show the legend in the display. The channel legend displays a name for each channel of each input signal. When the legend appears, you can place it anywhere inside the scope window. To turn off the legend, clear the **Show legend** check box. If the signal has multiple channels, the scope uses an index number to identify each signal channel.

You can use the legend to show and hide signals. Right-clicking a signal name toggles the signal visibility, shown or hidden. Clicking a signal name shows only the selected signal and hides the other signal.

You can edit the name of any channel by double-clicking the current channel name and entering a new name.

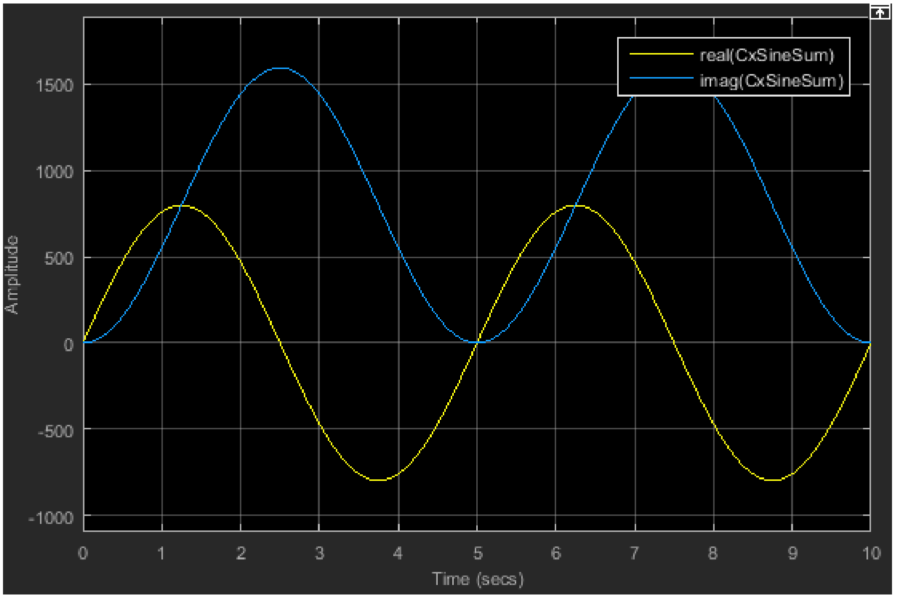
Tunable (Simulink)

Show grid

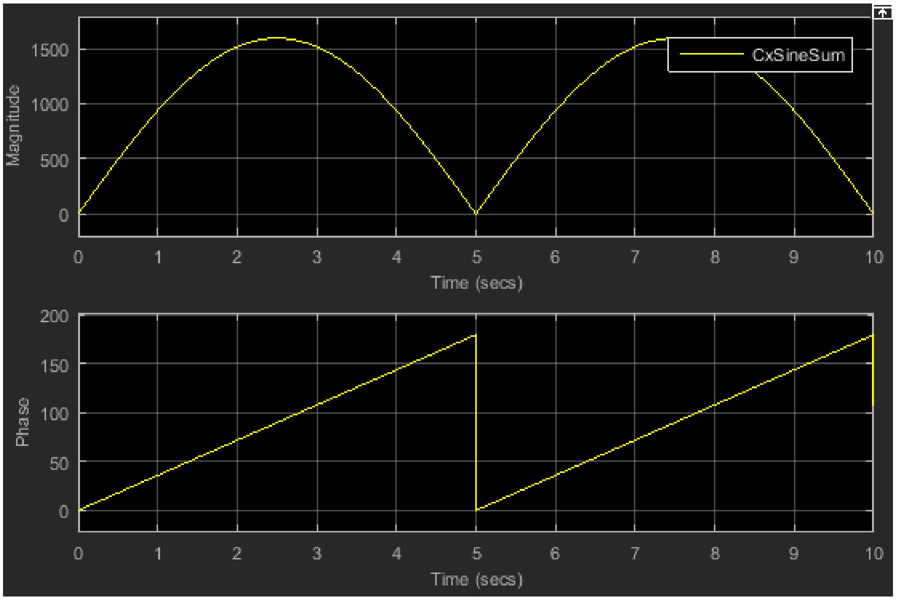
When you select this check box, a grid appears in the display of the scope figure. To hide the grid, clear this check box. Tunable (Simulink)

Plot signals as magnitude and phase

When you select this check box, the scope splits the display into a magnitude plot and a phase plot. By default, this check box is cleared. If the input signal has complex values, the scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes, as shown in the following figure.



Selecting this check box and clicking the **Apply** or **OK** button changes the display. The magnitude of the input signal appears on the top axes and its phase, in degrees, appears on the bottom axes. See the following figure.



This feature is useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box returns the absolute value of the signal for the magnitude. The phase is 0 degrees for nonnegative input and 180 degrees for negative input. Tunable (Simulink)

X-label

Specify the text for the scope to display below the *x*-axis. This property is Tunable (Simulink).

Y-label

Specify the text for the scope to display to the left of the *y*-axis. Tunable (Simulink)

Y-limits (Minimum)

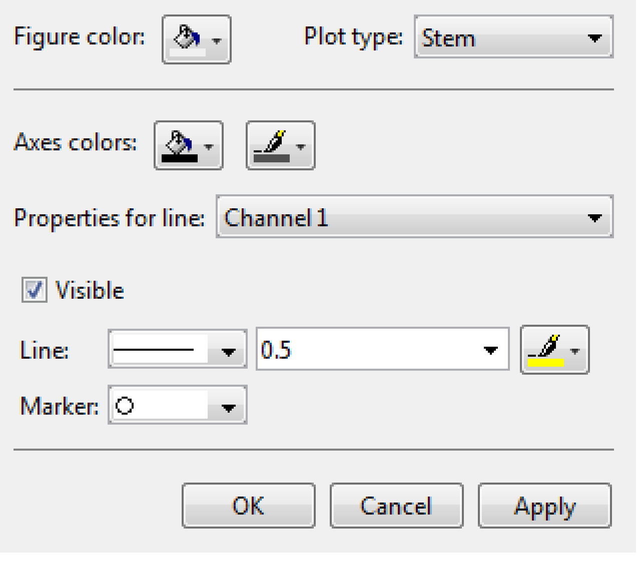
Specify the minimum value of the *y*-axis. Tunable (Simulink)

Y-limits (Maximum)

Specify the maximum value of the *y*-axis. Tunable (Simulink)

Style Dialog Box

In the **Style** dialog box, you can customize the style of displays. You are able to change the color of the figure containing the displays, the background and foreground colors of display axes, and properties of lines in a display. From the scope menu, select **View** > **Style** to open this dialog box.



Properties

The **Style** dialog box allows you to modify the following properties of the scope figure:

Figure color

Specify the color that you want to apply to the background of the scope figure. By default, the figure color is gray.

Plot type

Specify the type of plot to use for all the input signals displayed in the scope window.

- When you set this property to `'Stem'` , the scope displays the input signal as circles with vertical lines extending down to the x-axis at each of the sampled values . This approach is similar to the functionality of the MATLAB `stem` function.

- When you set this property to `'Line'` , the scope displays the input signal as lines connecting each of the sampled values. This approach is similar to the functionality of the MATLAB `line` or `plot` function.
- When you set this property to `'Stairs'`, the scope displays the input signal as a *stairstep* graph. A stairstep graph is made up of only horizontal lines and vertical lines. Each horizontal line represents the signal value for a discrete sample period and is connected to two vertical lines. Each vertical line represents a change in values occurring at a sample. This approach is equivalent to the MATLAB `stairs` function. Stairstep graphs are useful for drawing time history graphs of digitally sampled data.

This property is Tunable (Simulink).

The default setting is `Stem`.

Axes colors

Specify the color that you want to apply to the background of the axes.

Properties for line

Specify the channel for which you want to modify the visibility, line properties, and marker properties.

Visible

Specify whether the selected channel should be visible. If you clear this check box, the line disappears.

Line

Specify the line style, line width, and line color for the selected channel.

Marker

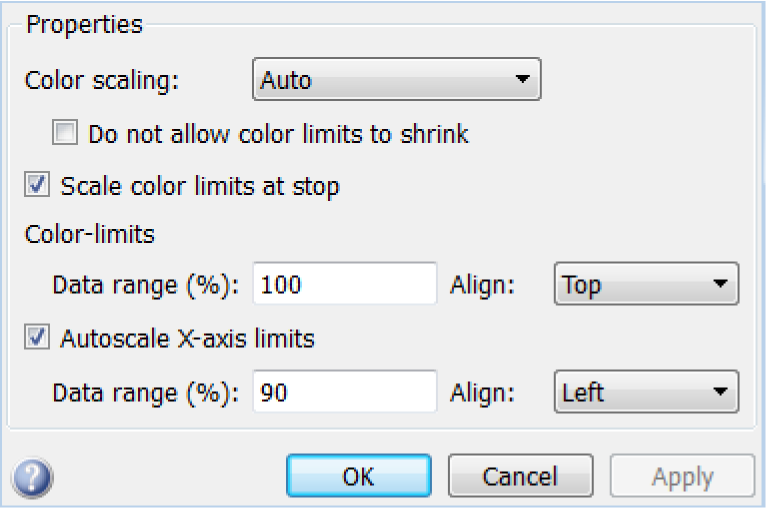
Specify marks for the selected channel to show at its data points. This parameter is similar to the `Marker` property for plot objects. You can choose any of the marker symbols from the dropdown.

Tools — Plot Navigation Properties

The Tools—Plot Navigation Properties dialog box allows you to automatically zoom in on and zoom out of your data. You can also scale the axes of the scope. In the scope menu, select **Tools > Axes Scaling Properties** to open this dialog box.

Properties

The Tools—Plot Navigation Properties dialog box appears as follows.



Axes Scaling

Specify when the scope automatically scales the axes. You can select one of the following options:

- `Manual` — When you select this option, the scope does not automatically scale the axes. You can manually scale the axes in any of the following ways:
  - Select **Tools** > **Axes Scaling Properties**.
  - Press one of the **Scale Axis Limits** toolbar buttons.
  - When the scope figure is the active window, press **Ctrl** and **A** simultaneously.
- `Auto` — When you select this option, the scope scales the axes as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** check box.
- `After N Updates` — Selecting this option causes the scope to scale the axes after a specified number of updates. This option is useful and more efficient when your scope display starts with one axis scale, but quickly reaches a different steady state axis scale. Selecting this option shows the **Number of updates** edit box.

By default, this property is set to `Auto`. This property is Tunable (Simulink).

By default, this parameter is set to `Manual`.

Do not allow Y-axis limits to shrink

When you select this property, the *y*-axis is allowed only to grow during axes scaling operations. If you clear this check box, the *y*-axis or color limits may shrink during axes scaling operations.

This property appears only when you select `Auto` for the **Axis scaling** property. When you set the **Axes scaling** property to `Manual` or `After N Updates`, the *y*-axis or color limits are allowed to shrink. Tunable (Simulink).

Number of updates

Specify as a positive integer the number of updates after which to scale the axes, or for the spectrogram, the color. This property appears only when you select `After N Updates` for the **Axes scaling** or **Color scaling** property. Tunable (Simulink).

Scale axes limits at stop

Select this check box to scale the axes when the simulation stops. The *y*-axis is always scaled. The *x*-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

Y-axis Data range (%)

Set the percentage of the *y*-axis that the scope uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the Scope scales the *y*-axis limits such that your data uses the entire *y*-axis range. If you then set this property to `30`, the scope increases the *y*-axis range such that your data uses only 30% of the *y*-axis range. Tunable (Simulink).

Y-axis Align

Specify where the scope aligns your data along the *y*-axis when it scales the axes. You can select `Top`, `Center`, or `Bottom`. Tunable (Simulink).

Autoscale X-axis limits

Check this box to allow the scope to scale the *x*-axis limits when it scales the axes. If **Axes scaling** is set to `Auto`, checking **Autoscale X-axis limits** only scales the data currently within the axes, not the entire signal in the data buffer. If **Autoscale X-axis limits** is on and the resulting axis is greater than the span of the scope, trigger position markers will not be displayed. Triggers are controlled using the Trigger Measurements panel. Tunable (Simulink).

X-axis Align

Specify how the scope aligns your data along the *x*-axis: `Left`, `Center`, or `Right`. This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supports MEX code generation by treating the calls to the object as extrinsic. Does not support code generation for standalone applications.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.TimeScope | dsp.SpectrumAnalyzer | dsp.LogicAnalyzer

Introduced in R2013a

hide

System object: dsp.ArrayPlot
Package: dsp

Hide scope window

Syntax

`hide(plot)`

Description

`hide(plot)` hides the scope window associated with System object, `plot`.

See Also

dsp.ArrayPlot.show

reset

System object: dsp.ArrayPlot
Package: dsp

Reset internal states of scope object

Syntax

```
reset(plot)
```

Description

`reset(plot)` sets the internal states of the scope object `plot` to their initial values.

You should call the `reset` method after calling the `step` method when you want to clear the scope figure displays, prior to releasing system resources. This action enables you to start a simulation from the beginning. When you call the `reset` method, the displays will become blank again. In this sense, its functionality is similar to that of the MATLAB `clf` function. Do not call the `reset` method after calling the `release` method.

Algorithms

In operation, the `reset` method is similar to a consecutive execution of the `mdlTerminate` function and the `mdlInitializeConditions` function.

See Also

See Also

dsp.ArrayPlot

show

System object: dsp.ArrayPlot
Package: dsp

Make scope window visible

Syntax

`show(plot)`

Description

`show(plot)` makes the scope window associated with System object, `plot`, visible.

See Also

dsp.ArrayPlot.hide

step

System object: dsp.ArrayPlot
Package: dsp

Display signal in scope figure

Syntax

```
step(plot,sig1)
step(plot,sig1,sig2,...,sigN)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`step(plot,sig1)` displays the signal, `sig1`, in the Array Plot.

`step(plot,sig1,sig2,...,sigN)` displays the signals `sig1`, `sig2`, ... ,`sigN` in the Array Plot `plot`. You must set the `NumInputPorts` property to `N` to enable more than one input signal. Each of the input signal matrices must have the same number of rows and cannot be scalars.

dsp.ArrayVectorAdder System object

Package: dsp

Add array to vector along specified dimension

Description

The `ArrayVectorAdder` object adds an N-D array to a vector along a specified dimension. The length of the vector must equal the size of the N-D array along the specified dimension.

To add an N-D array to a vector along a specified dimension:

1 Define and set up your array-vector addition object. See “Construction” on page 4-105.
2 Call step to add the N-D array according to the properties of `dsp.ArrayVectorAdder`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ava = dsp.ArrayVectorAdder` returns an array-vector addition object, `ava`, that adds a vector to an N-D array along the first dimension.

`ava = dsp.ArrayVectorAdder('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns an array-vector addition object, `ava`, with each property set to the specified value.

Properties

`Dimension`

Dimension along which to add vector elements to input

Specify the dimension along which to add the input array to the elements of the vector as a positive integer. The length of the vector must match the size of the N-D array along the specified dimension. The default is `1`.

`VectorSource`

Source of vector

Specify the source of the vector values as `|Input port | Property |`. The default is `Input port`.

`Vector`

Vector values

Specify the vector values. This property applies only when you set the `VectorSource` on page 4-106 property to `Property`. The default is `[0.5 0.25]`. This property is tunable.

Fixed-Point Properties

`FullPrecisionOverride`

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero` |. The default is `Floor`. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as | `Wrap` | `Saturate` |. The default is `Wrap`. This property applies only if the object is not in full precision mode.

VectorDataType

Vector word and fraction lengths

Specify the vector fixed-point data type as | `Same word length as input` | `Custom` |. This property applies when you set the `VectorSource` on page 4-106 property to `Property`. The default is `Same word length as input`.

CustomVectorDataType

Vector word and fraction lengths

Specify the vector fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `VectorSource` on page 4-106 property to `Property` and the `VectorDataType` on page 4-107 property to `Custom`. The default is `numerictype([],16,15)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as | `Full precision` | `Same as first input` | `Custom` |. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `AccumulatorDataType` on page 4-107 property is `Custom`. The default is `numerictype([],32,30)`.

`OutputDataType`

Output word and fraction lengths

Specify the output fixed-point data type as | `Same as accumulator` | `Same as first input` | `Custom` |.

`CustomOutputDataType`

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `OutputDataType` on page 4-108 property is `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Add vector to N-D array

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Add a Vector To a Matrix

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Add a 2-by-1 vector to a 2-by-2 matrix along the first dimension of the array.

```
ava = dsp.ArrayVectorAdder;
a = ones(2);
x = [1 2]';
y = ava(a, x);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Array-Vector Add` block reference page. The object properties correspond to the block parameters, except:

The array-vector addition object does not have **Minimum** or **Maximum** options for data output.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.ArrayVectorMultiplier | dsp.ArrayVectorDivider | dsp.ArrayVectorSubtractor

Introduced in R2012a

step

System object: dsp.ArrayVectorAdder
Package: dsp

Add vector to N-D array

Syntax

```
Y = step(ava,A)
Y = step(ava,A,V)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(ava,A)` returns `Y` , the result of adding the input array `A` to the elements of the vector specified in the `Vector` property along the specified dimension when the `VectorSource` property is `Property`. The length of the vector specified in the `Vector` property must equal the length of the specified dimension of `A`.

`Y = step(ava,A,V)` returns `Y` , the result of adding the input array `A` to the elements of the input vector `V` along the specified dimension when the `VectorSource` property is `Input port`. The length of the input `V` must equal the length of the specified dimension of `A`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change

nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.ArrayVectorDivider System object

Package: dsp

Divide array by vector along specified dimension

Description

The `ArrayVectorDivider` object divides an array by a vector along a specified dimension.

To divide an array by a vector along a specified dimension:

1. Define and set up your array-vector division object. See "Construction" on page 4-112.
2. Call step to divide the array according to the properties of `dsp.ArrayVectorDivider`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`avd = dsp.ArrayVectorDivider` returns an array-vector division object, `avd`, that divides an input array by the elements of a vector along the first dimension of the array.

`avd = dsp.ArrayVectorDivider('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an array-vector division object, `avd`, with each property set to the specified value.

Properties

`Dimension`

Dimension along which to divide input by vector elements

Specify the dimension along which to divide the input array by the elements of a vector as a positive integer. The default is `1`.

`VectorSource`

Source of vector

Specify the source of the vector values as | `Input port` | `Property` |. The default is `Input port`.

`Vector`

Vector values

Specify the vector values. This property applies when you set the `VectorSource` on page 4-113 property to `Property`. The default is `[0.5 0.25]`. This property is tunable.

Fixed-Point Properties

`RoundingMethod`

Rounding method for fixed-point operations

Specify the rounding method as | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero` |. The default is `Floor`.

`OverflowAction`

Overflow action for fixed-point operations

Specify the overflow action as | `Wrap` | `Saturate` |. The default is `Wrap`.

`VectorDataType`

Vector word and fraction lengths

Specify the vector fixed-point mode as | `Same word length as input` | `Custom` |. This property applies when you set the `VectorSource` on page 4-113 property to `Property`. The default is `Same word length as input`.

`CustomVectorDataType`

Vector word and fraction lengths

Specify the vector fixed-point data type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `VectorSource` on page 4-113 property to `Property` and the `VectorDataType` on page 4-113 property to `Custom`. The default is `numerictype([],16,15)`.

`OutputDataType`

Output word and fraction lengths

Specify the output fixed-point data type as | `Same as first input` | `Custom` |. The default is `Same as first input`.

`CustomOutputDataType`

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` on page 4-114 property to `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Divide array by vector

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Divide a Matrix By a Vector

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
avd = dsp.ArrayVectorDivider;
a = ones(2)
x = [2 3]'
y = avd(a, x)


a =

     1     1
     1     1


x =

     2
     3


y =

    0.5000    0.5000
    0.3333    0.3333
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Array-Vector Divide` block reference page. The object properties correspond to the block parameters, except:

The array-vector division object does not have **Minimum** or **Maximum** options for data output.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.ArrayVectorAdder | dsp.ArrayVectorMultiplier | dsp.ArrayVectorSubtractor

Introduced in R2012a

step

System object: dsp.ArrayVectorDivider
Package: dsp

Divide array by vector

Syntax

```
Y = step(avd,A,V)
Y = step(avd,A)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(avd,A,V)` returns `Y` , the result of dividing the input array `A` by the elements of input vector `V` along the specified dimension when the `VectorSource` property is `Input port`. The length of the input `V` must equal the length of the specified dimension of `A`.

`Y = step(avd,A)` returns `Y` , the result of dividing the input array `A` by the elements of the vector specified in the `Vector` property along the specified dimension when the `VectorSource` property is `Property`. The length of the vector specified in the `Vector` property must equal the length of the specified dimension of `A`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change

nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.ArrayVectorMultiplier System object

Package: dsp

Multiply array by vector along specified dimension

Description

The `ArrayVectorMultiplier` object multiplies an array by a vector along a specified dimension.

To multiply an array by a vector along a specified:

1. Define and set up your array-vector multiplication object. See "Construction" on page 4-119.
2. Call step to multiply the array according to the properties of `dsp.ArrayVectorMultiplier`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`avm = dsp.ArrayVectorMultiplier` returns an array-vector multiplication object, `avm`, that multiplies an input N-D array by the elements of a vector along the second dimension.

`avm = dsp.ArrayVectorMultiplier('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an array-vector multiplication object, `avm`, with each property set to the specified value.

Properties

Dimension

Dimension along which to multiply input by vector elements

Specify the dimension along which to multiply the input array by the elements of vector as a positive integer. The default is `2`.

VectorSource

Source of vector

Specify the source of the vector values as one of `Input port` or `Property`. The default is `Input port`.

Vector

Vector to multiply array

Specify the vector by which to multiply the array. This property applies when you set the `VectorSource` on page 4-127 property to `Property`. The default is `[0.5 0.25]`. This property is tunable.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor` , `Nearest`, `Round`, Simplest', `Zero`. The default is `floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

VectorDataType

Vector word and fraction lengths

Specify the vector fixed-point data type as `Same word length as input` , `Custom`. This property applies when you set the `VectorSource` on page 4-120 property to `Property`. The default is `Same word length as input`.

CustomVectorDataType

Vector word and fraction lengths

Specify the vector fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `VectorSource` on page 4-120 property to `Property` and the `VectorDataType` on page 4-120 property to `Custom`. The default is `numerictype([],16,15)`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as `Full precision` , `Same as first input`, or `Custom`. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` on page 4-121 property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Full precision` , `Same as product`, `Same as first input`, or `Custom`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` on page 4-121 property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as `Same as product`, `Same as first input`, or `Custom`. The default is `Same as product`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` on page 4-122 property to `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Multiply array by vector

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Multiply a Matrix by a Vector

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
avm = dsp.ArrayVectorMultiplier;
a = ones(2);
```

```
x = [2 3]';
y = avm(a, x);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Array-Vector Multiply` block reference page. The object properties correspond to the block parameters, except:

- The array-vector multiplication object does not have **Minimum** or **Maximum** options for data output.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.ArrayVectorAdder | dsp.ArrayVectorDivider | dsp.ArrayVectorSubtractor

Introduced in R2012a

step

System object: dsp.ArrayVectorMultiplier
Package: dsp

Multiply array by vector

Syntax

```
Y = step(avm,A,V)
Y = step(avm,A)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(avm,A,V)` returns `Y` , the result of multiplying the input array `A` by the elements of input vector `V` along the specified dimension when the `VectorSource` property is `Input port`. The length of the input `V` must equal the length of the specified dimension of `A`.

`Y = step(avm,A)` returns `Y` , the result of multiplying the input array `A` by the elements of vector specified in `Vector` property along the specified dimension when the `VectorSource` property is set to `Property`. The length of the vector specified in `Vector` property must equal the length of the specified dimension of `A`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change

nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.ArrayVectorSubtractor System object

Package: dsp

Subtract vector from array along specified dimension

Description

The `ArrayVectorSubtractor` object subtracts a vector from an N-D array along a specified dimension.

To subtract a vector from an N-D array along a specified dimension:

1. Define and set up your array-vector subtraction object. See "Construction" on page 4-126.
2. Call step to subtract the vector according to the properties of `dsp.ArrayVectorSubtractor`. The behavior of `step` is specified to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`avs = dsp.ArrayVectorSubtractor` returns an array-vector subtraction object, `avs`, that subtracts the elements of a vector from an N-D input array along the first dimension.

`avs = dsp.ArrayVectorSubtractor('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an array-vector subtraction object, `avs`, with each property set to the specified value.

Properties

`Dimension`

Dimension along which to subtract vector elements from input

Specify the dimension along which to subtract the elements of the vector from the input array as an integer-valued scalar greater than `0`. The default is `1`.

`VectorSource`

Source of vector

Specify the source of the vector values as one of `Input port` or `Property`. The default is `Input port`.

`Vector`

Vector values

Specify the vector values. This property applies when you set the `VectorSource` on page 4-127 property to `Property`. The default is `[0.5 0.25]`. This property is tunable.

Fixed-Point Properties

`FullPrecisionOverride`

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

`RoundingMethod`

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor` , `Nearest`, `Round`, `Simplest`, or `Zero`. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` or `Saturate`. The default is `Wrap`. This property applies only if the object is not in full precision mode.

VectorDataType

Vector word and fraction lengths

Specify the vector fixed-point data type as `Same word length as input` or `Custom`. This property applies when you set the `VectorSource` on page 4-127 property to `Property`. The default is `Same word length as input`.

CustomVectorDataType

Vector word and fraction lengths

Specify the vector fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `VectorSource` on page 4-127 property to `Property` and the `VectorDataType` on page 4-128 property to `Custom`. The default is `numerictype([],16,15)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Full precision` , `Same as first input`, or `Custom`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` on page 4-128 property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as `Same as accumulator`, `Same as first input`, or `Custom`. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` on page 4-129 property to `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Subtract vector from array along specified dimension

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Subtract Vector From Matrix

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
avs = dsp.ArrayVectorSubtractor;
```

```
a = ones(2);
x = [1 2]';
y = avs(a, x)
```

```
y =

     0     0
    -1    -1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Array-Vector Subtract` block reference page. The object properties correspond to the block parameters, except:

The array-vector subtraction object does not have **Minimum** or **Maximum** options for data output.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.ArrayVectorAdder | dsp.ArrayVectorMultiplier | dsp.ArrayVectorDivider

Introduced in R2012a

step

System object: dsp.ArrayVectorSubtractor
Package: dsp

Subtract vector from array along specified dimension

Syntax

```
Y = step(avs,A,V)
Y = step(avs,A)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(avs,A,V)` returns `Y`. The value of `Y` results from subtracting the elements of input vector `V` from the input array `A` along the specified dimension when the `VectorSource` property is `Input port`. The length of the input `V` must equal the length of the specified dimension of `A`.

`Y = step(avs,A)` returns `Y`. The value of `Y` results from subtracting the elements of the vector specified in the `Vector` property from the input array `A` along the specified dimension when the `VectorSource` property is `Property`. The length of the vector specified in the `Vector` property must equal the length of the specified dimension of `A`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change

nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.AsyncBuffer System object

Package: dsp

FIFO buffer

Description

The `dsp.AsyncBuffer` System object writes samples to and reads samples from a first-in, first-out (FIFO) buffer. The `write` method writes data to the buffer and the `read` method reads data from the buffer. When creating the object, you can set the number of samples (rows) of the buffer using the `Capacity` property. The number of channels (columns) is set during the first call to `write`. Initialize the buffer by calling `write` or `setup` before the first call to `read`.

The data that you write occupies the next available space in the buffer. If the buffer is full and all the data within it is unread, that is, if `asyncBuff.NumUnreadSamples == asyncBuff.Capacity`, the object overwrites the oldest data with any new data that comes in. The buffer removes data only when the data is overwritten, so you can reread data from the past. The `dsp.AsyncBuffer` object supports writing and reading variable frame size signals. For examples, see “Read Variable Frame Sizes from Buffer” on page 4-135 and “Write Variable Frame Sizes to Buffer” on page 4-137.

To write and read samples from a FIFO buffer:

1 Create a `dsp.AsyncBuffer` object and set the properties of the object.
2 Call write to write samples to the buffer.
3 Call read to read samples from the buffer.

Construction

`asyncBuff = dsp.AsyncBuffer` returns an `AsyncBuffer` object, `asyncBuff`, using the default properties.

`asyncBuff = dsp.AsyncBuffer(cap)` sets the `Capacity` property to `cap`.

`asyncBuff = dsp.AsyncBuffer(Name,Value)` specifies properties using `Name,Value` pairs. Unspecified properties have default values.

Example:

```
asyncBuff = dsp.AsyncBuffer(200000);
```

Properties

`Capacity` — Number of writable/readable rows in buffer

`192000` (default) | positive integer greater than or equal to 2

Number of writable/readable rows in buffer. The number of rows during each write to the buffer must not exceed the capacity of the buffer. If the buffer is full and all the data within is unread, the object overwrites the oldest data with any new data that comes in. The `CumulativeOverrun` property returned by the `info` method gives the number of samples overrun per channel since the last call to `reset`. The number of samples overrun is the number of unread samples overwritten.

`NumUnreadSamples` — Number of unread samples in each channel

`0` (default) | scalar greater than or equal to 0

This property is read-only.

Number of unread samples in each channel (column) of the buffer. The total number of unread samples in the buffer is `NumUnreadSamples` × `numChann`, where `numChann` is the number of channels in the buffer.

The `CumulativeUnderrun` property returned by the `info` method gives the number of samples underrun per channel since the last call to `reset`. Underrun occurs if you attempt to read more samples than available.

Limitations

Before calling the `read` method, you must initialize the buffer by calling either the `write` or `setup` method. For an example, see “Why Does the dsp.AsyncBuffer Object Error When You Call read Before write?”

Methods

info	Get cumulative overrun and underrun

read	Read data from buffer
reset	Reset internal states of System object
write	Write data to buffer

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

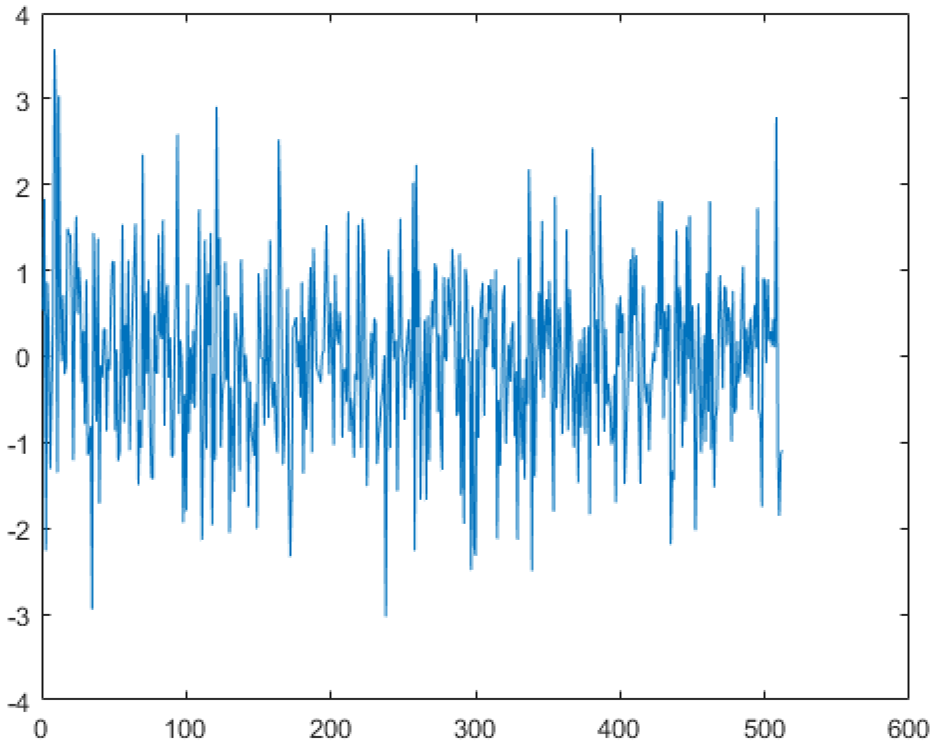
Examples

Read Variable Frame Sizes from Buffer

The `dsp.AsyncBuffer` System object™ supports reading variable frame sizes from the buffer.

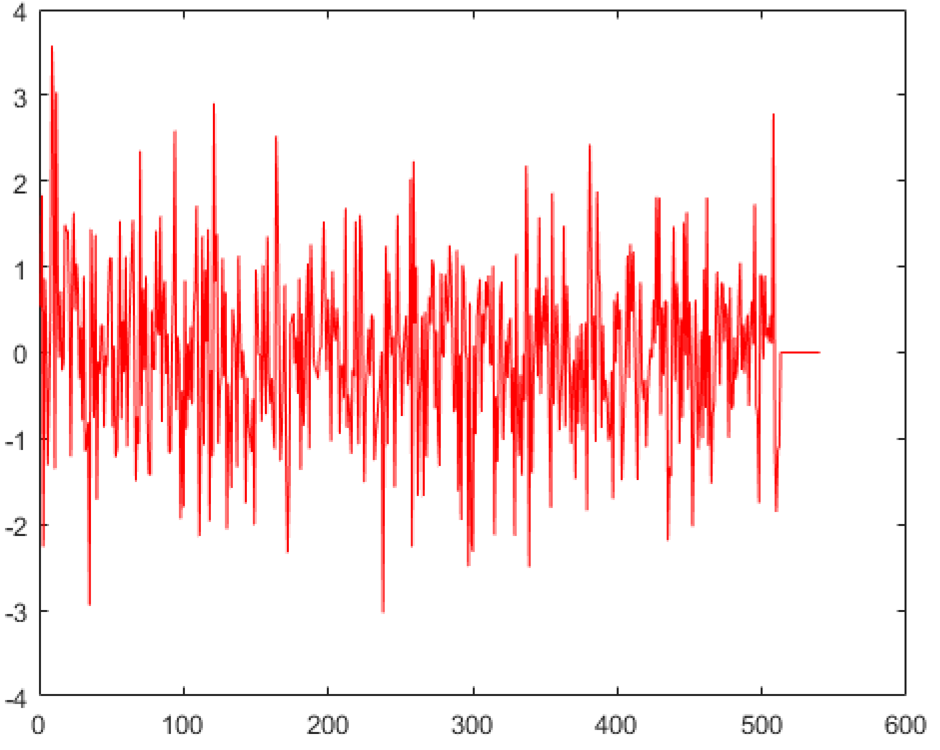
Create a `dsp.AsyncBuffer` System object. The input is white Gaussian noise with zero mean, a standard deviation of 1, and a frame size of 512 samples. Write the input to the buffer using the `write` method. Store the data that is read from the buffer in `outTotal`.

```
asyncBuff = dsp.AsyncBuffer;
input = randn(512,1);
write(asyncBuff,input);

outTotal = [];
plot(input)
hold on
```



Plot the input signal and data that is read from the buffer in the same plot. Read data from the buffer until all samples are read. In each loop of iteration, `randi` determines the number of samples to read. Therefore, the signal is read in as a variable-size signal.

```
while asyncBuff.NumUnreadSamples ~= 0
    numToRead = randi([1,64]);
    out = read(asyncBuff,numToRead);
    outTotal = [outTotal;out];
    plot(outTotal,'r');
    pause(0.2)
end
hold off
```



Write Variable Frame Sizes to Buffer

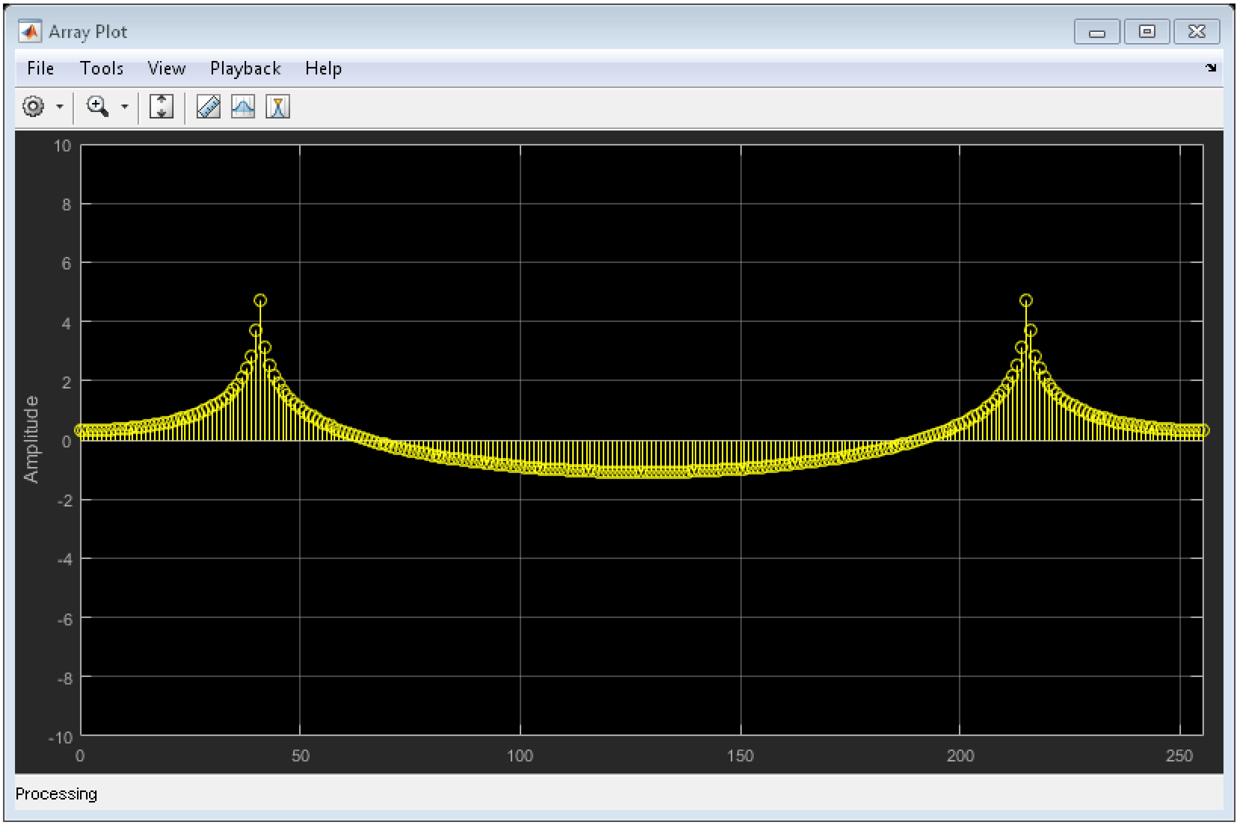
Write a sine wave of variable frame size to the buffer. Compute the FFT of the sine wave and visualize the result on an array plot.

Initialize the `dsp.AsyncBuffer`, `dsp.ArrayPlot`, and `dsp.FFT` System objects.

```
asynBuff = dsp.AsyncBuffer;
plotter = dsp.ArrayPlot;
fftObj = dsp.FFT('FFTLengthSource','Property','FFTLength',256);
```

The sine wave is generated using the `sin` function in MATLAB®. The `start` and `finish` variables mark the start and finish indices of each frame. If enough data is cached, read from the buffer and perform the FFT. View the FFT on an array plot.

```
start = 1;

for Iter = 1 : 2000
    numToWrite = randi([200,800]);
    finish = start + numToWrite;

    inputData = sin(start:finish)';
    start = finish + 1;

    write(asynBuff,inputData);
    while asynBuff.NumUnreadSamples >= 256
        x = read(asynBuff,256);
        X = abs(fftObj(x));
        plotter(log(X));
    end
end
```



- "Why Does the dsp.AsyncBuffer Object Error When You Call read Before write?"
- "High Resolution Spectral Analysis"

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

System Objects
dsp.Buffer | dsp.DelayLine | dsp.Delay

Blocks
`Buffer` | `Delay Line` | `Queue`

Topics
“Why Does the dsp.AsyncBuffer Object Error When You Call read Before write?”
“High Resolution Spectral Analysis”
“Why Does Reading Data from the dsp.AsyncBuffer Object Give a Dimension Mismatch Error in the MATLAB Function Block?”

Introduced in R2017a

info

System object: dsp.AsyncBuffer
Package: dsp

Get cumulative overrun and underrun

Syntax

```
S = info(asyncBuff)
```

Description

`S = info(asyncBuff)` returns a structure, `S`, containing the cumulative overrun and underrun information of the `dsp.AsyncBuffer` System object, `asyncBuff`.

The `CumulativeOverrun` property gives the number of samples overrun per channel since the last call to `reset`. The number of samples overrun is the number of unread samples overwritten. The `CumulativeUnderrun` property gives the number of samples underrun per channel since the last call to `reset`. Underrun occurs if you attempt to read more samples than available.

Introduced in R2017a

read

System object: dsp.AsyncBuffer
Package: dsp

Read data from buffer

Syntax

```
out = read(asyncBuff)
out = read(asyncBuff,NumRows)
out = read(asyncBuff,NumRows,Overlap)
[out,nUnderrun] = read(asyncBuff,___)
```

Description

`out = read(asyncBuff)` returns all unread samples from the buffer, `asyncBuff`.

`out = read(asyncBuff,NumRows)` returns `NumRows` samples from each channel (column) of the buffer. If the requested number of samples is greater than the number of unread samples, the output is zero-padded.

`out = read(asyncBuff,NumRows,Overlap)` returns `NumRows` samples from each channel and overlaps previously read samples by `Overlap`. The total number of samples read is `NumRows` × `NumChann`, where `NumChann` is the number of channels in the buffer. The total number of new samples read is (`NumRows` – `Overlap`) × `NumChann`. If the overlap portion contains samples that are overwritten, and are therefore not contiguously written, the output is zero-padded.

`[out,nUnderrun] = read(asyncBuff,___)` additionally returns the number of rows zero-padded if underrun occurred, using any of the previous arguments. `Underrun` occurs if you attempt to read more samples than available. Samples that are zero-padded in overlapped portions are not counted as underrun.

Introduced in R2017a

reset

System object: dsp.AsyncBuffer
Package: dsp

Reset internal states of System object

Syntax

`reset(asyncBuff)`

Description

`reset(asyncBuff)` returns the buffer to its initial condition, and resets the `CumulativeOverrun` and `CumulativeUnderrun` properties.

Introduced in R2017a

write

System object: dsp.AsyncBuffer
Package: dsp

Write data to buffer

Syntax

```
nOverrun = write(asyncBuff,x)
```

Description

`nOverrun = write(asyncBuff,x)` writes the input array, `x`, to the buffering object, `asyncBuff`. The object returns the number of samples overrun in the current call to `write`. The number of samples overrun is the number of unread samples overwritten. If `x` is a multichannel input, then `nOverrun` is the number of rows of data overrun.

Introduced in R2017a

audioDeviceWriter System object

Play to sound card

Description

The `audioDeviceWriter` System object writes audio samples to an audio output device. See "Audio Device Writer System Interaction" on page 4-152 for a visualization of how the `audioDeviceWriter` System object plays audio samples.

To stream data to an audio device:

1. Define and set up your audio device writer. See "Construction" on page 4-145.
2. Call step or play to stream data to an audio device.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`aDW = audioDeviceWriter` returns a System object, `aDW` that writes audio samples to an audio output device in real time.

`aDW = audioDeviceWriter(sampleRateValue)` sets the `SampleRate` property to `sampleRateValue`.

`aDW = audioDeviceWriter(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `aDW = audioDeviceWriter(12000,'BitDepth','8-bit integer')` creates a System object, `aDW`, with the `SampleRate` property set to `12000` and the `BitDepth` property set to `'8-bit integer'`.

Properties

`Driver` — Driver used to access audio device (Windows only)

`'DirectSound'` (default) | `'ASIO'` | `'WASAPI'`

Driver used to access your audio device, specified as `'DirectSound'`, `'ASIO'`, or `'WASAPI'`.

- ASIO drivers do not come pre-installed on Windows machines. To use the `'ASIO'` driver option, install an ASIO driver outside of MATLAB.

  Note: If `Driver` is specified as `'ASIO'`, open the ASIO UI outside of MATLAB to set the sound card buffer size to the `BufferSize` value of your `audioDeviceWriter` System object. See the documentation of your ASIO driver for more information.

- WASAPI drivers are supported for exclusive-mode only.

ASIO and WASAPI drivers do not provide sample rate conversion. For ASIO and WASAPI drivers, set `SampleRate` to a sample rate supported by your audio device.

This property applies only on Windows machines. Linux machines always use the ALSA driver. Mac machines always use the CoreAudio driver.

To specify nondefault `Driver` values, you must install Audio System Toolbox. If the toolbox is not installed, specifying nondefault `Driver` values returns an error.

`Device` — Device used to play audio samples

default audio device (default) | character vector

Device used to play audio samples, specified as a character vector. Use the `getAudioDevices` method to list available devices.

`SampleRate` — Sample rate of signal sent to audio device (Hz)

`44100` (default) | positive integer

Sample rate of signal sent to audio device, in Hz, specified as a positive integer. The range of `SampleRate` depends on your audio hardware.

`BitDepth` — Data type used by the device

`'16-bit integer'` (default) | `'8-bit integer'` | `'24-bit integer'` | `'32-bit float'`

Data type used by the device, specified as a character vector. Before performing digital-to-analog conversion, the input data is cast to a data type specified by `BitDepth`.

To specify a nondefault `BitDepth`, you must install Audio System Toolbox. If the toolbox is not installed, specifying a nondefault `BitDepth` returns an error.

`SupportVariableSizeInput` — Option to support variable frame size
`false` (default) | `true`

Option to support variable frame size, specified as `true` or `false`.

- `false` — If the `audioDeviceWriter` object is locked, the input must have the same frame size at each call to `step` or `play`. The buffer size of your audio device is the same as the input frame size.
- `true` — If the `audioDeviceWriter` object is locked, the input frame size can change at each call to `step` or `play`. The buffer size of your audio device is specified through the `BufferSize` property.

`BufferSize` — Buffer size of audio device
`4096` (default) | positive integer

Buffer size of audio device, specified as a positive integer.

Note: If `Driver` is specified as `'ASIO'`, open the ASIO UI to set the sound card buffer size to the `BufferSize` value of your `audioDeviceWriter` System object.

This property is available when you set `SupportVariableSizeInput` to `true`.

`ChannelMappingSource` — Source of mapping between columns of input matrix and channels of output device
`'Auto'` (default) | `'Property'`

Source of mapping between columns of input matrix and channels of audio output device, specified as `'Auto'` or `'Property'`.

- `'Auto'` — Default settings determine the mapping between columns of input matrix and channels of audio output device. For example, suppose that your input is a matrix with four columns, and your audio device has four channels available. Column 1 of your input data writes to channel 1 of your device, column 2 of your input data writes to channel 2 of your device, and so on.

- `'Property'` — The `ChannelMapping` property determines the mapping between columns of input matrix and channels of audio output device.

`ChannelMapping` — Nondefault mapping between columns of input matrix and channels of output device
`[1:MaximumOutputChannels]` (default) | scalar | vector

Nondefault mapping between columns of input matrix and channels of output device, specified as a scalar or vector of valid channel indices.

This property is available when you set `ChannelMappingSource` to `'Property'`.

To selectively map between columns of the input matrix and your sound card's output channels, you must install Audio System Toolbox. If the toolbox is not installed, specifying a nondefault `ChannelMapping` returns an error.

Methods

getAudioDevices	List available audio input devices
info	Get information about selected device
play	Stream audio data to output device
step	Stream audio data to output device

Examples

Read from File and Write to Audio Device

Read an MP3 audio file and play it through your default audio output device.

Create a `dsp.AudioFileReader` System object™ with default settings. Use the `audioinfo` function to return a structure containing information about the audio file.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
fileInfo = audioinfo('speech_dft.mp3');
```

Create an `audioDeviceWriter` System object and specify the sample rate. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
deviceWriter = audioDeviceWriter(...
```

```
    'SampleRate',fileInfo.SampleRate);
setup(deviceWriter,...
    zeros(fileReader.SamplesPerFrame,fileInfo.NumChannels));
```

In an audio stream loop, read an audio signal frame from the file, and write the frame to your device.

```
while ~isDone(fileReader)
    audioData = fileReader();
    deviceWriter(audioData);
end
```

Close the input file and release the device.

```
release(fileReader);
release(deviceWriter);
```

Reduce Latency due to Output Device Buffer

Modify default properties of your `audioDeviceWriter` System object™ to reduce latency due to device buffer size.

Create a `dsp.AudioFileReader` System object to read an audio file with default settings.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
```

Create an `audioDeviceWriter` System object and specify the sample rate to match that of the audio file reader.

```
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
```

Calculate the latency due to your device buffer, in seconds.

```
bufferLatency = fileReader.SamplesPerFrame/deviceWriter.SampleRate
```

```
bufferLatency =

    0.0464
```

Set the `SamplesPerFrame` property of your `dsp.AudioFileReader` System object to 256. Calculate the buffer latency in seconds.

```
fileReader.SamplesPerFrame = 256;
bufferLatency = fileReader.SamplesPerFrame/deviceWriter.SampleRate
```

```
bufferLatency =

    0.0116
```

Determine and Decrease Underrun

Underrun refers to output signal silence, which occurs when the audio stream loop does not keep pace with the output device. Determine the underrun of an audio stream loop, add artificial computational load to the audio stream loop, and then modify properties of your `audioDeviceWriter` System object™ to decrease underrun. Your results depend on your computer.

Create a `dsp.AudioFileReader` System object, and specify the file to read. Use the `audioinfo` function to return a structure containing information about the audio file.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
fileInfo = audioinfo('speech_dft.mp3');
```

Create an `audioDeviceWriter` System object. Use the `SampleRate` of the file reader as the `SampleRate` of the device writer. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
setup(deviceWriter,...
    zeros(fileReader.SamplesPerFrame,fileInfo.NumChannels));
```

Run your audio stream loop with input from file and output to device. Print the total samples underrun and the underrun in seconds.

```
totalUnderrun = 0;
while ~isDone(fileReader)
    input = fileReader();
    numUnderrun = deviceWriter(input);
    totalUnderrun = totalUnderrun + numUnderrun;
end
fprintf('Total samples underrun: %d.\n',...
    totalUnderrun);
fprintf('Total seconds underrun: %d.\n',...
```

```
    double(totalUnderrun)/double(deviceWriter.SampleRate));
```

```
Total samples underrun: 0.
Total seconds underrun: 0.
```

Release your `dsp.AudioFileReader` and `audioDeviceWriter` System objects and set your counter variable to zero.

```
release(fileReader);
release(deviceWriter);
totalUnderrun = 0;
```

Use a pause to mimic an algorithm that takes 0.075 seconds to process. The pause causes the audio stream loop to go slower than the device, which results in periods of silence in the output audio signal.

```
while ~isDone(fileReader)
    input = fileReader();
    numUnderrun = deviceWriter(input);
    totalUnderrun = totalUnderrun + numUnderrun;
    pause(0.075)
end
fprintf('Total samples underrun: %d.\n',...
    totalUnderrun);
fprintf('Total seconds underrun: %d.\n',...
    double(totalUnderrun)/double(deviceWriter.SampleRate));
```

```
Total samples underrun: 70656.
Total seconds underrun: 3.204354e+00.
```

Release your `audioDeviceReader` and `dsp.AudioFileWriter` and set the counter variable to zero.

```
release(fileReader);
release(deviceWriter);
totalUnderrun = 0;
```

Set the frame size of your audio stream loop to 2048. Because the `SupportVariableSizeInput` property of your `audioDeviceWriter` System object is set to `false`, the buffer size of your audio device is the same size as the input frame size. Increasing your device buffer size decreases underrun.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
fileReader.SamplesPerFrame = 2048;
fileInfo = audioinfo('speech_dft.mp3');
```

```
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
setup(deviceWriter,...
    zeros(fileReader.SamplesPerFrame,fileInfo.NumChannels));
```

Calculate the total underrun.

```
while ~isDone(fileReader)
    input = fileReader();
    numUnderrun = deviceWriter(input);
    totalUnderrun = totalUnderrun + numUnderrun;
    pause(0.075)
end
fprintf('Total samples underrun: %d.\n',...
    totalUnderrun);
fprintf('Total seconds underrun: %d.\n',...
    double(totalUnderrun)/double(deviceWriter.SampleRate));
```

```
Total samples underrun: 0.
Total seconds underrun: 0.
```

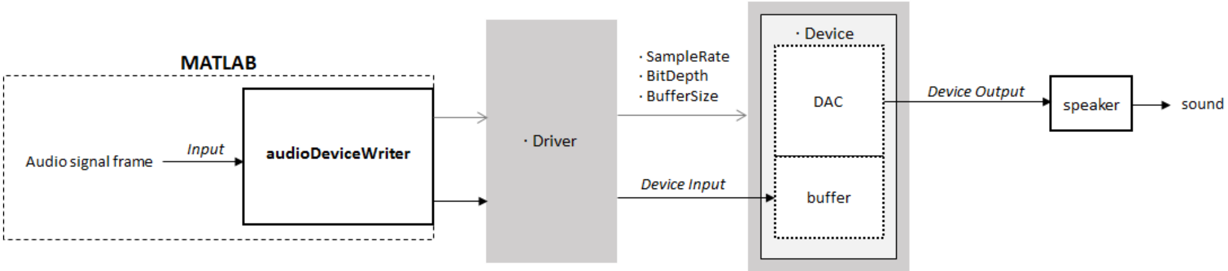
The increased frame size reduces the total underrun of your audio stream loop. However, increasing the frame size also increases latency. Other approaches to reduce underrun include:

- Increase the buffer size independent of input frame size. To increase buffer size independent of input frame size, you must first set `SupportVariableSizeInput` to `true`. This approach also increases latency.
- Decrease the sample rate. Decreasing the sample rate reduces both latency and underrun at the cost of signal resolution.
- Choose an optimal driver and device for your system.

More About

Audio Device Writer System Interaction

Properties of the audio device writer specify the driver, the device, and device attributes such as sample rate, bit depth, and buffer size.



Data Flow of Audio Device Writer

- Call `step` or `play` to input an audio signal frame to the `audioDeviceWriter`.
- The `audioDeviceWriter` uses the specified driver to pass the frame (device input) to the buffer of your specified audio device.
- The audio device performs digital-to-analog conversion at the specified sample rate and bit depth.
- The audio device outputs an analog chunk to your speaker.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- "System Objects in MATLAB Code Generation" (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "How To Run a Generated Executable Outside MATLAB".

See Also

See Also

Blocks

`Audio Device Writer`

System Objects

dsp.AudioFileWriter | dsp.AudioFileReader

Topics

“How To Run a Generated Executable Outside MATLAB”

Introduced in R2016a

getAudioDevices

System object: audioDeviceWriter

List available audio input devices

Syntax

```
devices = getAudioDevices(aDW)
```

Description

`devices = getAudioDevices(aDW)` returns a cell array listing available audio output devices. The list of available output devices depends on the specified `Driver` property of your `audioDeviceWriter` object.

Introduced in R2016a

info

System object: audioDeviceWriter

Get information about selected device

Syntax

```
aDWInfo = info(aDW)
```

Description

`aDWInfo = info(aDW)` returns a structure containing information about your `audioDeviceWriter` System object. The structure contains information about the driver, device, and maximum number of input channels for your `audioDeviceWriter` System object.

Introduced in R2016a

play

System object: audioDeviceWriter

Stream audio data to output device

Syntax

`numUnderrun = play(aDW,x)`

Description

`numUnderrun = play(aDW,x)` writes one frame of audio samples, `x`, to the audio output device specified by the audio device writer System object, `aDW`. The number of samples underrun since the last call to `play` is returned.

If `x` is of data type `'double'` or `'single'`, the audio device writer clips values outside the range [–1, 1]. For other data types, the allowed input range is [min, max] of the specified data type.

When you call the `play` method of an `audioDeviceWriter` System object, the audio device specified by the `Device` property is locked. An audio device can be locked by only one `audioDeviceWriter` at a time. Call the `release` method of the `audioDeviceWriter` System object to release the audio device.

Note: The System object performs an internal initialization the first time you execute `play`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

step

System object: audioDeviceWriter

Stream audio data to output device

Syntax

```
numUnderrun = step(aDW,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`numUnderrun = step(aDW,x)` writes one frame of audio samples, `x`, to the audio output device specified by the audio device writer System object, `aDW`. The number of samples underrun since the last call to `step` is returned.

If `x` is of data type `'double'` or `'single'`, the audio device writer clips values outside the range [–1, 1]. For other data types, the allowed input range is [min, max] of the specified data type.

When you call the `step` method of an `audioDeviceWriter` System object, the audio device specified by the `Device` property is locked. An audio device can be locked by only one `audioDeviceWriter` at a time. Call the `release` method of the `audioDeviceWriter` System object to release the audio device.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change

nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

dsp.AudioFileReader System object

Package: dsp

Stream from audio file

Description

The `AudioFileReader` object reads audio samples from an audio file.

To read audio samples from an audio file:

1. Define and set up your audio file reader object. See “Construction” on page 4-160.
2. Call step to read audio samples according to the properties of `dsp.AudioFileReader`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`afr = dsp.AudioFileReader` returns an audio file reader System object, `afr` that reads audio from an audio file.

`afr = dsp.AudioFileReader('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an audio file reader System object, `afr`, with each specified property set to the specified value.

`afr = dsp.AudioFileReader(Filename,'`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an audio file reader object, `afr`, with `Filename` property set to *`FILENAME`* and other specified properties set to the specified values.

Properties

Filename

Name of audio file from which to read

Specify the name of an audio file as a character vector. Specify the full path for the file only if the file is not on the MATLAB path. The default is `speech_dft.mp3`.

PlayCount

Number of times to play file

Specify a positive integer as the number of times to play the file. The default is `1`.

SampleRate

Sampling rate of the audio file

This read-only property displays the sampling rate, in Hz, of the audio file.

SamplesPerFrame

Number of samples in audio frame

Specify the number of samples in an audio frame as a positive, scalar integer value. The default value is `1024`.

OutputDataType

Data type of output

Set the data type of the audio data output from the audio file reader object. Specify the data type as `double| single | int16 | uint8`. The default is `double`.

Supported Platforms and File Types

The following table lists the supported audio file formats:

Platforms	File Name Extensions
Windows	`.wav`, `.wma`, `.avi`, `.aif`, `.aifc`, `.aiff`, `.mp3`, `.au`, `.snd, .mp4, .m4a, .flac, .ogg`

Platforms	File Name Extensions
Non-Windows	`.avi, .mp3, .mp4, .m4a, .wav, .flac, .ogg, .aif, .aifc, .aiff,`

Methods

info	Information about specific audio file
isDone	End-of-file status (logical)
reset	Reset internal states of audio file reader to read from beginning of file
step	Read audio samples from audio file

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Read and Play back an Audio File

This example shows how to read and play back an audio file using the standard audio output device.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For `dsp.AudioFileReader` System object™, myObject() becomes step(myObject). For all other objects, myObject(x) becomes step(myObject,x).

```
afr = dsp.AudioFileReader('speech_dft.mp3');
adw = audioDeviceWriter('SampleRate', afr.SampleRate);

while ~isDone(afr)
```

```
    audio = afr();
    adw(audio);
end
release(afr);
release(adw);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `From Multimedia File` block reference page. The object properties correspond to the block parameters, except:

- The object has no corresponding property for the **Inherit sample time from file** block parameter. The object always inherits the sample time from the file.
- The object has no corresponding property for the **Output end-of-file indicator** parameter. The object always outputs `EOF` as the last output.
- The object has no corresponding property for the **Multimedia Outputs** parameter because audio is the only supported output.
- The object has no corresponding property for the **Image signal** block parameter.
- The object has no corresponding property for the **Output color format** parameter.
- The object has no corresponding property for the **Video output data type** parameter.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).
- The executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file.

Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

See Also

See Also

System Objects
dsp.AudioFileWriter

Topics
“How To Run a Generated Executable Outside MATLAB”

Introduced in R2012a

info

System object: dsp.AudioFileReader
Package: dsp

Information about specific audio file

Syntax

```
S = info(afr)
```

Description

`S = info(afr)` returns a MATLAB structure, `S`, with information about the audio file specified in the `Filename` property. The number of fields in `S` varies depending on the audio content of the file. For possible fields and values for the structure `S`, see the following table.

Field	Value
SampleRate	Audio sampling rate of the audio file in Hz.
NumBits	Number of bits used to encode the audio stream.
NumChannels	Number of audio channels.

isDone

System object: dsp.AudioFileReader
Package: dsp

End-of-file status (logical)

Syntax

```
STATUS = isDone(afr)
```

Description

`STATUS = isDone(afr)` returns a logical value, `STATUS` , when the file is read `PlayCount` number of times. If you set the `PlayCount` property to a value greater than `1` , `STATUS` is `true` when `EOF` is reached `PlayCount` number of times, and it is `false` otherwise. `STATUS` is the same as the `PlayCount` output value in the `step` method syntax.

reset

System object: dsp.AudioFileReader
Package: dsp

Reset internal states of audio file reader to read from beginning of file

Syntax

```
reset(afr)
```

Description

`reset(afr)` resets the `AudioFileReader` object to read from the beginning of the file.

step

System object: dsp.AudioFileReader
Package: dsp

Read audio samples from audio file

Syntax

```
AUDIO = step(afr)
[AUDIO,EOF] = step(afr)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`AUDIO = step(afr)` outputs one frame of audio samples, `AUDIO`. You can specify the number of times to play the file using the `PlayCount` property. After playing the file for the number of times you specify, `AUDIO` contains silence.

`[AUDIO,EOF] = step(afr)` returns an end-of-file indicator, `EOF`. `EOF` is true each time the output `AUDIO` contains the last audio sample in the file.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.AudioFileWriter System object

Package: dsp

Stream to audio file

Description

The `AudioFileWriter` object writes audio samples to an audio file.

To write audio samples to an audio file:

1. Define and set up your audio file writer object. See "Construction" on page 4-169.
2. Call step to write audio samples according to the properties of `dsp.AudioFileWriter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`afw = dsp.AudioFileWriter` returns an audio file writer System object, `afw`. This object writes audio samples to an audio file.

The following platform-specific restrictions apply when writing these files:

Windows 7	Mac OS X
• Only sample rates of `44,100` Hz and `48,000` Hz are supported for MPEG-4 AAC file format. For other file formats, there is no restriction on the sample rate.	• Only mono or stereo outputs are allowed for MPEG-4 AAC file format. For all other formats, more than two audio output channels are allowed.
• Only mono or stereo outputs are allowed for MPEG-4 AAC file format. For all	

Windows 7	Mac OS X
other formats, more than two audio output channels are allowed.	
• The output data is padded on both the front and back of the signal, with extra samples of silence. Windows AAC encoder places sharp fade-in and fade-out on audio signal, causing signal to be slightly longer in samples when written to disk.	• Not all sampling rates are supported, although the Mac Audio Toolbox API do not explicitly specify a restriction.
• A minimum of 1025 samples per channel must be written to the MPEG-4 AAC file.	

`afw = dsp.AudioFileWriter('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an audio file writer System object, `afw`, with each specified property set to the specified value.

`afw = dsp.AudioFileWriter(`*`FILENAME`*`,'`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an audio file writer System object, `afw`. This object has the `Filename` property set to *`FILENAME`* and other specified properties set to the specified values.

Properties

`Filename`

Name of audio file to which to write

Specify the name of the audio file as a character vector. The default is `output.wav`.

`FileFormat`

Audio file format

Specify which audio file format the object writes. On Microsoft platforms, select one of `AVI | WAV | FLAC | OGG | MPEG4 | WMA`. On Linux platforms, select one of `AVI | WAV | FLAC | OGG`. On Mac OS X platforms, select one of `AVI | WAV | FLAC | OGG | MPEG4`. These abbreviations correspond to the following file formats:

- `AVI`: Audio-Video Interleave
- `WAV`: Microsoft WAVE Files
- `WMA`: Windows Media Audio
- `FLAC`: Free Lossless Audio Codec
- `OGG`: Ogg/Vorbis Compressed Audio File
- `MPEG4`: MPEG-4 AAC File — You can use both `.m4a` and `.mp4` extensions

The default is `WAV`.

SampleRate

Sampling rate of audio data stream

Specify the sampling rate of the input audio data as a positive, numeric scalar value. The default is `44100`.

Compressor

Algorithm that compresses audio data

Specify the type of compression algorithm the audio file writer uses to compress the audio data. Compression reduces the size of the audio file. Select `None (uncompressed)` to save uncompressed audio data to the file. The other options available reflect the audio compression algorithms installed on your system. You can use tab completion to query valid `Compressor` options for your computer by typing `H.Compressor = '` and then pressing the tab key. This property applies when writing `WAV` or `AVI` files on Windows platforms.

DataType

Data type of the uncompressed audio

Specify the type of uncompressed audio data written to the file as one of `inherit`, `int16`, `int24`, `single`, or `uint8`. This property only applies when writing uncompressed `WAV` files. The default value of this property is `int16`.

Methods

step	Write one frame of audio output samples

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Write an Audio Signal to a WAV File

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Decimate an audio signal, and write it to disk as a WAV file.

```
afr = dsp.AudioFileReader('OutputDataType',...
   'double');
firdec = dsp.FIRDecimator; % decimate by 2
afw = dsp.AudioFileWriter...
   ('speech_dft.wav', ...
   'SampleRate', afr.SampleRate/2);

while ~isDone(afr)
     audio = afr();
     audiod = firdec(audio);
     afw(audiod);
end

release(afr);
release(afw);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `To Multimedia File` block reference page. The object properties correspond to the block parameters, except:

- The object `FileFormat` property does not support video-only file formats.
- The object has no corresponding property for the **Write** parameter. The object writes only audio content to files.
- The object has no corresponding property for the **Video compressor** parameter.
- The object has no corresponding property for the **File color format** parameter.
- The object has no corresponding property for the **Image signal** parameter.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See "System Objects in MATLAB Code Generation" (MATLAB Coder).
- The executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "How To Run a Generated Executable Outside MATLAB".

See Also

See Also

System Objects

dsp.AudioFileReader

Topics

"How To Run a Generated Executable Outside MATLAB"

Introduced in R2012a

step

System object: dsp.AudioFileWriter
Package: dsp

Write one frame of audio output samples

Syntax

`step(afw,AUDIO)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`step(afw,AUDIO)` writes one frame of audio samples, `AUDIO` , to the output file. `AUDIO` is either a vector or an M-by-N matrix for mono or N-channel audio inputs respectively.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.AudioPlayer System object

Package: dsp

Play audio data using computer's audio device

Note: The `dsp.AudioPlayer` object will be removed in a future release. Existing instances of the object continue to run. For new code, use the audioDeviceWriter object instead.

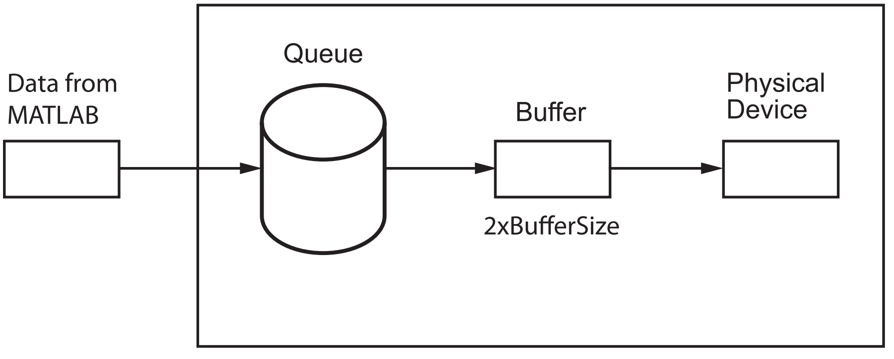
Description

The `AudioPlayer` object plays audio data using the computer's audio device.

To play audio data using the computer's audio device:

1. Define and set up your audio player object. See "Construction" on page 4-176.
2. Call step to play audio data according to the properties of `dsp.AudioPlayer`. The behavior of `step` is specific to each object in the toolbox.

This System object buffers the data from the audio device using the process illustrated by the following figure.



Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = dsp.AudioPlayer` returns an audio player object, `H`, that plays audio samples using an audio output device in real-time.

`H = dsp.AudioPlayer('PropertyName',PropertyValue, ...)` returns an audio player object, `H`, with each property set to the specified value.

`H = dsp.AudioPlayer(SAMPLERATE,'PropertyName',PropertyValue, ...)` returns an audio player object, `H`, with the `SampleRate` property set to `SAMPLERATE` and other specified properties set to the specified values. This System object supports variable-size input. If you use variable-size signals with this System object, you may experience sound dropouts when the size of the input frame increases. To avoid this behavior, use a signal of maximum expected size when you first call `step` to start running through this System object.

Properties

`DeviceName`

Device to which to send audio data

Specify the device to which to send the audio data. The default is `Default`, which is the computer's standard output device. You can use tab completion to query valid `DeviceName` assignments for your computer by typing `H.DeviceName = '` and then pressing the tab key.

`SampleRate`

Number of samples per second sent to audio device

Specify the number of samples per second in the signal as an integer. The default is `44100`. This property is tunable.

DeviceDataType

Data type used by device

Specify the data type used by the audio device to acquire audio data as `Determine from input data type`, `8-bit integer`, `16-bit integer`, `24-bit integer`, or `32-bit float`. The default is `Determine from input data type`.

BufferSizeSource

Source of Buffer Size

Specify how to determine the buffer size as `Auto` or `Property`. The default is `Auto`. When this property is set to `Auto`, an appropriate buffer size based on the `SampleRate` gets computed.

BufferSize

Buffer size

Specify the size of the buffer that the audio player object uses to communicate with the audio device as an integer. `BufferSize` is half the size of the sound card buffer. A frame of data cannot be passed to the queue until the device empties the buffer, which introduces *latency*. Latency is the time it takes the device to empty the queue and the buffer. `BufferSize` has to be smaller than the effective queue duration. This property is tunable. Tuning this property involves a balance between device latency and the possibility of dropping data (buffer underrun).

This property applies when you set the `BufferSizeSource` property to `Property`. The default is `4096`. To set the `BufferSize` to a value other than the default, first change the `BufferSizeSource` to `'Property'`. You can select `BufferSize` in the list of properties.

QueueDuration

Size of queue in seconds

Specify the length of the audio queue, in seconds. The default is `1.0`. This property is tunable. The purpose of the queue is to control the trade-off between latency and data dropout. Latency is calculated by the following equation:

$$latency = \frac{QueueDuration \times SampleRate + 2 \times BufferSize}{SampleRate}.$$

To minimize latency, lower the `QueueDuration` or set it to `0`. However, be aware that data dropouts or loss of system robustness may result. The `QueueDuration` property specifies the duration of the signal, in seconds, that can be buffered during the simulation. This value is the maximum length of time that the System object's data supply can lag the device's data demand. If the MATLAB data throughput rate is lower than the device throughput rate, a buffer underrun occurs. You can use `OutputNumUnderrunSamples` to monitor underrun. To correct the underrun, make the queue duration larger than the buffer. If the MATLAB data throughput rate is higher than the device throughput rate, a buffer overrun occurs, causing the System object to wait before writing data to the queue. To minimize the chance of dropouts, the System object checks to verify the queue duration is at least as large as the maximum of the buffer size and the frame size. If it is not, the queue duration is automatically set to this maximum value. At the start of the simulation, the queue is filled with silence. At each time step, the System object sends a buffer of samples from the top of the queue to the audio device. If the queue does not contain enough data to completely fill the buffer, the System object fills the remaining portion of the buffer with zeros.

OutputNumUnderrunSamples

Enable output of underrun count

Set to `true` to output the number of zero samples inserted due to queue underrun since the last call to the `step` method. The default is `false`.

ChannelMappingSource

Source of device channel mapping

Specify whether to determine the channel mapping as `'Auto'` or as `'Property'`. If you set the value of `ChannelMappingSource` to `'Auto'`, the `ChannelMapping` field is rendered inactive. If you set this property to `'Property'`, the vector specified in the `ChannelMapping` field is used to route the output.

ChannelMapping

Data-to-device channel mapping

Vector of valid channel indices to represent the mapping between data and device output channels. The term *Channel Mapping* refer to a 1-to-1 mapping that associates channels on the selected audio device to channels of the data. When you play audio, channel mapping allows you to specify which channel of the audio data to output a specific channel of audio data. By default, the `ChannelMapping` field is [1:MAXOUTPUTCHANNELS], where MAXOUTPUTCHANNELS depends upon the selected device.

Methods

step	Write audio to audio output device

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Read and Play an Audio File

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Read in an AVI audio file, and play the file back using the standard audio output device.

```
AFR = dsp.AudioFileReader; % points to a default audio file
AP = audioDeviceWriter('SampleRate',AFR.SampleRate);
while ~isDone(AFR)
  audio = AFR();
  nUnderrun = AP(audio);
  if nUnderrun > 0
    fprintf('Audio player queue underrun by %d samples.\n'...
      ,nUnderrun);
  end
end
release(AFR);            % close the input file
```

```
release(AP);                % close the audio output device
```

Troubleshooting

Running an Executable Outside MATLAB

To run your generated standalone executable application in Shell, you need to set your environment to the following:

Platform	Command

Mac	`setenv DYLD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/maci64 (csh/tcsh)`
	
	`export DYLD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/maci64 (Bash)`

Linux	`setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/glnxa64 (csh/tcsh)`
	
	`export LD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/glnxa64 (Bash)`

Windows	`set PATH = $MATLABROOT\bin\win64;%PATH%`

Algorithms

This object implements the algorithm, inputs, and outputs described on the `To Audio Device` block reference page. The object properties correspond to the block parameters.

See Also

dsp.AudioRecorder | dsp.AudioFileReader

Topics

Set the Audio Hardware API on page 2-1730

Introduced in R2012a

step

System object: dsp.AudioPlayer
Package: dsp

Write audio to audio output device

Note: The `dsp.AudioPlayer` object will be removed in a future release. Existing instances of the object continue to run. For new code, use the audioDeviceWriter object instead.

Syntax

```
step(H,AUDIO)
Underrun = step(H,AUDIO)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`step(H,AUDIO)` writes one frame of `AUDIO` samples to the audio output device.

`Underrun = step(H,AUDIO)` writes one frame of `AUDIO` samples to the audio output device. The output `Underrun` indicates the number of zero samples inserted due to queue underrun since the last call to the `step` method. This syntax applies when you set the `OutputNumUnderrunSamples` on page 4-178 property to `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.AudioRecorder System object

Package: dsp

Record audio data using computer's audio device

Note: The `dsp.AudioRecorder` object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `audioDeviceReader` object from Audio System Toolbox instead.

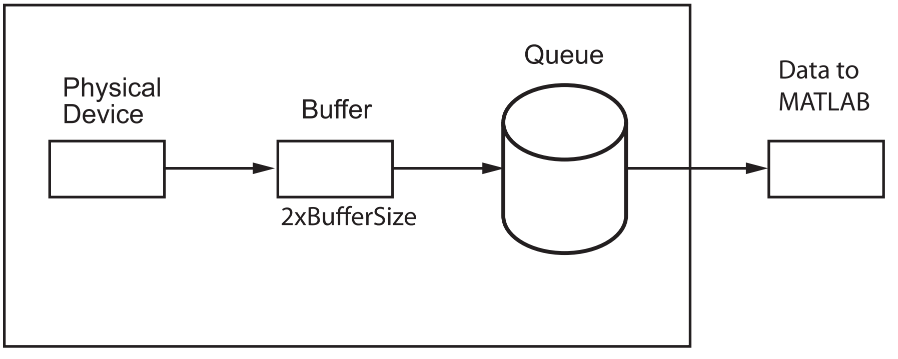
Description

The `AudioRecorder` object records audio data using the computer's audio device.

To record audio data using the computer’s audio device:

1. Define and set up your audio recorder object. See “Construction” on page 4-185.
2. Call step to record audio data according to the properties of `dsp.AudioRecorder`. The behavior of `step` is specific to each object in the toolbox.

This System object buffers the data from a frame of data using the process illustrated by the following figure.



Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = dsp.AudioRecorder` returns an audio recorder object, `H`, that records audio samples using an audio input device in real-time.

`H = dsp.AudioRecorder('PropertyName',PropertyValue, ...)` returns an audio recorder object, `H`, with each property set to the specified value.

Properties

`DeviceName`

Device from which to acquire audio data

Specify the device from which to acquire audio data. The default is `Default`, which is the computer's standard input device. You can use tab completion to query valid `DeviceName` assignments for your computer by typing `H.DeviceName = '` and then pressing the tab key. The tab completion functionality shows all valid audio device names for your computer.

`SampleRate`

Number of samples per second read from audio device

Specify the number of samples per second in the signal as an integer. The default is 44100. This property is tunable.

`NumChannels`

Number of audio channels

Specify the number of audio channels as an integer. The default is 2.

DeviceDataType

Data type used by device

Specify the data type used by the device to acquire audio data as `Determine from output data type` , `8-bit integer`, `16-bit integer`, `24-bit integer`, or `32-bit float`. The default is `Determine from output data type`.

BufferSizeSource

Source of Buffer Size

Specify how to determine the buffer size as `Auto` or `Property`. The default is `Auto`.

BufferSize

Buffer size

Specify as an integer the size of the buffer that the audio recorder object uses to communicate with the audio device. This property applies when you set the `BufferSizeSource` property to `Property`. The default is `4096`. This property is tunable. Tuning this property involves a balance between device latency and the possibility of dropping data (buffer underrun). The instant you receive the buffer from a device, you have samples that are relatively new and samples that are at least as old as the size of the buffer. Therefore, the buffer size introduces *latency*, which is the time required for the device to fill the queue and the buffer. `BufferSize` is half the size of the sound card buffer. The size of the buffer processed in each interrupt from the audio device affects the performance of the system. A frame of data cannot pass through the queue until the buffer is filled by the device, thus introducing latency. `BufferSize` has to be smaller than the effective queue duration. To set the `BufferSize` to a value other than the default, first change the `BufferSizeSource` to 'Property'. You can select `BufferSize` from the list of properties.

QueueDuration

Size of queue in seconds

Specify the length of the audio queue, in seconds. The default is `1.0`. This property is tunable. The purpose of the queue is to control the trade-off between latency and data dropout. Latency is calculated by the following equation:

$$latency = \frac{QueueDuration \times SampleRate + 2 \times BufferSize}{SampleRate}.$$

To minimize latency, lower the `QueueDuration` or set it to 0. However, be aware that data dropouts or loss of system robustness may result. The `QueueDuration` property specifies the duration of the signal, in seconds, that can be buffered during the simulation. This value is the maximum length of time that the System object's data supply can lag the device's data demand. If the MATLAB data throughput rate is lower than the device throughput rate, a buffer overrun occurs. You can use `OutputNumOverrunSamples` to monitor overrun. To correct the overrun, make the queue duration larger than the buffer. If the MATLAB data throughput rate is higher than the device throughput rate, a buffer underrun occurs, causing the System object to wait for new samples to become available. To minimize the chance of dropouts, the System object checks to verify the queue duration is at least as large as the maximum of the buffer size and the frame size. If it is not, the queue duration is automatically set to this maximum value. At the start of the simulation, the queue is filled with silence. At each time step, the audio device sends a buffer of samples to the top of the queue.

SamplesPerFrame

Number of samples in the output signal

Specify the number of samples in the audio recorder's output as an integer. The default is 1024.

OutputDataType

Data type of the output

Select the output data type as `uint8`, `int16`, `int32` , `single` , or `double`. The default is `double`.

OutputNumOverrunSamples

Enable output of overrun count

Set to `true` to output the number of samples dropped due to queue overrun since the last call to the `step` method. The default is `false`.

ChannelMappingSource

Source of device channel mapping

Specify whether to determine the channel mapping as `'Auto'` or as `'Property'`. If you set the value of `ChannelMappingSource` to `'Auto'`, the `ChannelMapping` field is rendered inactive. If you set this property to `'Property'`, the vector specified in the ChannelMapping field is used to route the input. Additionally, the `NumChannels` field is rendered inactive, because the channel map contains information about the number of data channels that the user is attempting to read.

ChannelMapping

Device-to-data channel mapping

Vector of valid channel indices to represent the mapping between device input channels and the data. The term *Channel Mapping* refers to a 1-to-1 mapping that associates channels on the selected audio device to channels of the data. When you record audio, channel mapping allows you to specify which channel of the audio data directs input to a specific channel of audio. By default, the `ChannelMapping` field is [1:MAXNUMINPUTCHANNELS], where MAXNUMINPUTCHANNELS depends upon the selected device.

Methods

step	Record audio from recording device

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Troubleshooting

Running an Executable Outside MATLAB

To run your generated standalone executable application in Shell, you need to set your environment to the following:

Platform	Command

Mac	`setenv DYLD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/maci64 (csh/tcsh)`
	
	`export DYLD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/maci64 (Bash)`

Linux	`setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/glnxa64 (csh/tcsh)`
	
	`export LD_LIBRARY_PATH $LD_LIBRARY_PATH: $MATLABROOT/bin/glnxa64 (Bash)`

Windows	`set PATH = $MATLABROOT\bin\win64;%PATH%`

Algorithms

This object implements the algorithm, inputs, and outputs described on the `From Audio Device` block reference page. The object properties correspond to the block parameters.

See Also

dsp.AudioPlayer | dsp.AudioFileReader

Topics

Set the Audio Hardware API on page 2-823

Introduced in R2012a

step

System object: dsp.AudioRecorder
Package: dsp

Record audio from recording device

Note: The `dsp.AudioRecorder` object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `audioDeviceReader` object from Audio System Toolbox instead.

Syntax

```
AUDIO = step(H)
[AUDIO,Overrun] = step(H)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`AUDIO = step(H)` reads one frame of audio samples from the selected audio input device.

`[AUDIO,Overrun] = step(H)` reads one frame of audio samples from the selected audio input device. The output `Overrun` indicates the number of samples dropped due to queue overrun since the last call to the `step` method. This syntax applies when you set the `OutputNumOverrunSamples` on page 4-187 property to `true`.

dsp.Autocorrelator System object

Package: dsp

Autocorrelation sequence

Description

The `Autocorrelator` object returns the autocorrelation sequence for a discrete-time, deterministic input, or the autocorrelation sequence estimate for a discrete-time, wide-sense stationary (WSS) random process at positive lags.

To obtain the autocorrelation sequence:

1. Define and set up your autocorrelator. See "Construction" on page 4-191.
2. Call step to compute the autocorrelation sequence according to the properties of `dsp.Autocorrelator`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ac = dsp.Autocorrelator` returns an autocorrelator, `ac`, that computes the autocorrelation along the first dimension of an N-D array. By default, the autocorrelator computes the autocorrelation at lags from zero to $N-1$, where N is the length of the input vector or the row dimension of the input matrix. Inputting a row vector results in a row of zero-lag autocorrelation sequence values, one for each column of the row vector. The default autocorrelator returns the unscaled autocorrelation and performs the computation in the time domain.

`ac = dsp.Autocorrelator('PropertyName',PropertyValue, ...)` returns an autocorrelator, `ac`, with each property set to the specified value.

Properties

`MaximumLagSource`

Source of maximum lag

Specify how to determine the range of lags for the autocorrelation as `Auto` or `Property`. If the value of `MaximumLagSource` is `Auto`, the autocorrelator computes the autocorrelation over all nonnegative lags in the interval `[0, N-1]`, where N is the length of the first dimension of the input. Otherwise, the object computes the autocorrelation using lags in the range `[0,MaximumLag on page 4-192]`. The default is `Auto`.

`MaximumLag`

Maximum positive lag

Specify the maximum lag as a positive integer. This property applies only when the `MaximumLagSource` on page 4-192 property is `Property`. The `MaximumLag` must be less than the length of the input data. The default is `1`.

`Scaling`

Autocorrelation function scaling

Specify the scaling to apply to the output as `None`, `Biased`, `Unbiased`, or `Unity at zero-lag`. Set this property to `None` to generate the autocorrelation function without scaling. This option is appropriate if you are computing the autocorrelation of a nonrandom (deterministic) input.

The `Biased` option scales the autocorrelation by $1/N$, where N is the length of the input data. Scaling by $1/N$ yields a biased, finite-sample approximation to the theoretical autocorrelation of a WSS random process. In spite of the bias, scaling by $1/N$ has the desirable property that the sample autocorrelation matrix is nonnegative definite, a property possessed by the theoretical autocorrelation matrices of all wide-sense stationary random processes. The Fourier transform of the biased autocorrelation estimate is the *periodogram*, a widely used estimate of the power spectral density of a WSS process.

The `Unbiased` option scales the estimate of the autocorrelation by 1/N-1. Scaling by $N-1$ produces an unbiased estimate of the theoretical autocorrelation. However, using the unbiased option, you can obtain an estimate of the autocorrelation function that fails to have the nonnegative definite property.

Use the `Unity at zero-lag` option to normalize the autocorrelation estimate as identically one at lag zero. The default is `None`.

`Method`

Domain for computing autocorrelations

Specify the domain for computing autocorrelations as `Time Domain` or `Frequency Domain`. You must set this property to `Time Domain` for fixed-point signals. The default is `Time Domain`.

Fixed-Point Properties

`FullPrecisionOverride`

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see "Full Precision for Fixed-Point System Objects".

`RoundingMethod`

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. This property applies only when you set the `Method` on page 4-193 property to `Time Domain`. The default is `Floor`. This property applies only if the object is not in full precision mode.

`OverflowAction`

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. This property applies only when you set the `Method` on page 4-193 property to `Time Domain`. The default is `Wrap`. This property applies only if the object is not in full precision mode.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of `Full precision`, `Same as input`, or `Custom`. This property applies only when you set the `Method` on page 4-193 property to `Time Domain`. The default is `Full precision`

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `Method` on page 4-193 property to `Time Domain` and the `ProductDataType` on page 4-194 property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Full precision`, `Same as product`, `Same as input`, or `Custom`. This property applies only when the `Method` on page 4-193 property is `Time Domain`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `Method` on page 4-193 property to `Time Domain` and the `AccumulatorDataType` on page 4-194 property to `Custom`. The default is`numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as `Same as accumulator`, `Same as product`, `Same as input`, or `Custom`. This property applies only when the `Method` on page 4-193 property is `Time Domain`. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `Method` on page 4-193 property to `Time Domain` and the `OutputDataType` on page 4-194 property to `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Autocorrelation sequence

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Autocorrelation Of a Number Sequence

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

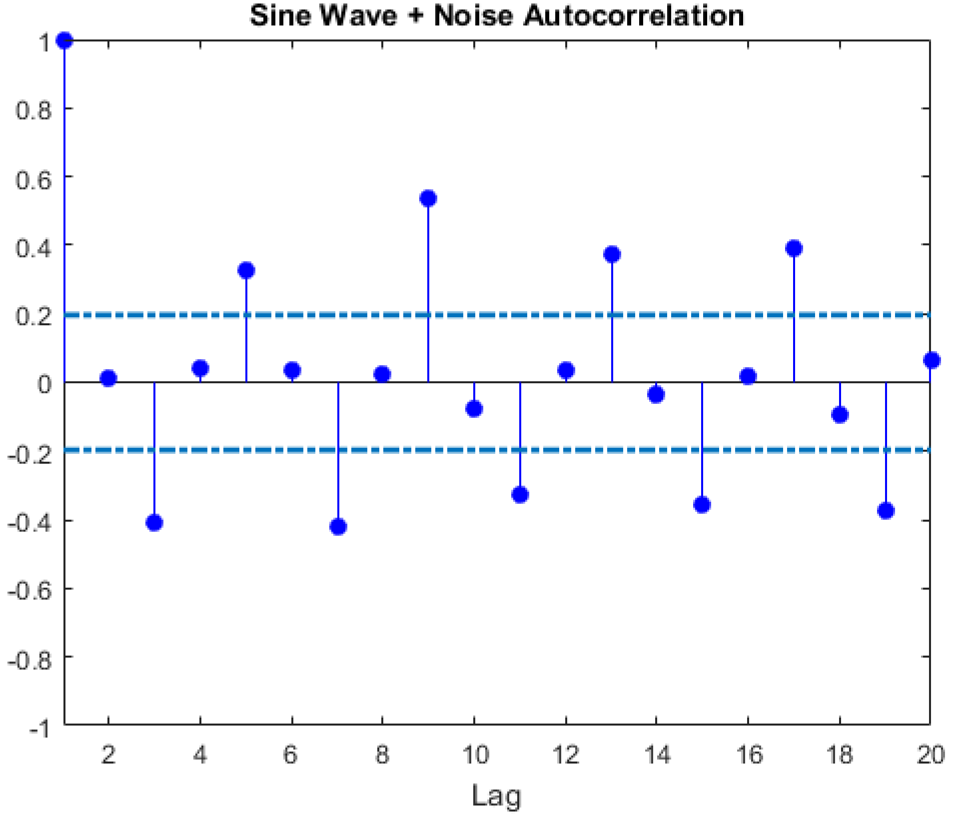
```
ac1 = dsp.Autocorrelator;
% x is a column vector
x = (1:100)';
y = ac1(x);
```

Autocorrelation of a Noisy Sine Wave

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Compute the autocorrelation of a sine wave in white Gaussian noise with approximate 95% upper and lower confidence limits.

```
S = rng('default');
% Sine wave with period N=4
x = 1.4*cos(pi/2*(1:100))'+randn(100,1);
MaxLag = 20;
ac = dsp.Autocorrelator('MaximumLagSource',...
'Property','MaximumLag',MaxLag,'Scaling','Unity at zero-lag');
SigAutocorr = ac(x);
stem(SigAutocorr,'b','markerfacecolor',[0 0 1]);
line(1:MaxLag+1,1.96/sqrt(100)*ones(MaxLag+1,1),...
     'linestyle','-.','linewidth',2);
line(1:MaxLag+1,-1.96/sqrt(100)*ones(MaxLag+1,1),...
     'linestyle','-.','linewidth',2);
axis([1 20 -1 1]);
title('Sine Wave + Noise Autocorrelation'); xlabel('Lag');
```



As this figure shows, the autocorrelation estimate demonstrates the four sample periodic sine wave with excursions outside the 95% white Gaussian noise confidence limits every two samples.

Definitions

Autocorrelation

Autocorrelation is the correlation of a signal with itself at different points in time.

For a deterministic discrete-time sequence, *x(n)*, the autocorrelation is computed using the following relationship:

$$r_x(h) = \sum_{n=0}^{N-h-1} x^*(n)x(n+h) \quad h = 0,1,\ldots,N-1$$

where h is the lag and * denotes the complex conjugate. If the input is a length N realization of a WSS stationary random process, $r_x(h)$ is an estimate of the theoretical autocorrelation:

$$\rho_x(h) = E\{x^*(n)x(n+h)\}$$

where E{ } is the expectation operator. The `Unity at zero-lag` normalization divides each sequence value by the autocorrelation or autocorrelation estimate at zero lag.

$$\frac{\rho_x(h)}{\rho_x(0)} = \frac{E\{x^*(n)x(n+h)\}}{E\{|x(0)|^2\}}$$

The most commonly used estimate of the theoretical autocorrelation of a WSS random process is the biased estimate:

$$\hat{\rho}_x(h) = \frac{1}{N}\sum_{k=0}^{N-h-1} x^*(n)x(n+h)$$

Algorithms

Time domain Computation

When you set the computation domain to time, the algorithm computes the autocorrelation of the input signal in the time domain. The input signal can be a fixed-point signal in this domain.

The autocorrelation sequence, y, is computed using this equation:

$$y_{i,j} = \sum_{k=0}^{M-l-1} u_{k,j}^{*} u_{(k+i),j} \qquad 0 \le i \le l$$

- $y_{0,j}$ is the zero-lag element in the jth column of the input.
- i is the index of the lag.
- j is the index of the input data column.
- * denotes the complex conjugate.
- M is the number of elements in each column.
- l is the maximum positive lag for autocorrelation. When you choose to compute the autocorrelation with all nonnegative lags, l=M–1. Otherwise, l is the maximum nonnegative integer lag value specified.
- u is an M-by-N input matrix.

Frequency Domain Computation

When you set the computation domain to frequency, the algorithm computes the autocorrelation in the frequency domain.

In this domain, the algorithm computes the autocorrelation sequence by taking the Fourier transform of the input signal, multiplying the Fourier transform with its complex conjugate, and taking the inverse Fourier transform of the product. In this domain, depending on the input length, the algorithm can require fewer computations.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.Crosscorrelator

Introduced in R2012a

step

System object: dsp.Autocorrelator
Package: dsp

Autocorrelation sequence

Syntax

```
Y = step(ac,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(ac,X)` computes the autocorrelation sequence `Y` for the columns of the input `X`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.BinaryFileReader System object

Package: dsp

Read data from binary file

Description

The `dsp.BinaryFileReader` System object reads multichannel signal data from a binary file. If the header is not empty, then the header precedes the signal data. The System object specifies the prototype of the header, and the type, size, and complexity of the data. The first time you read the file, the reader reads the header, followed by the data. On subsequent calls, the reader reads the remaining data. Once the end of file is reached, the reader returns zeros of the specified data type, size, and complexity. The reader can read signal data from a binary file that is not created by the dsp.BinaryFileWriter System object.

The object accepts floating-point data or integer data. To read character data and fixed-point data, see the "Write and Read Character Data" on page 4-212 and "Write and Read Fixed-Point Data" on page 4-210 examples. The input data can be real or complex. When the data is complex, the object reads the data as interleaved real and imaginary components. For an example, see "Read Complex Data" on page 4-209. The reader assumes the default endianness of the host machine. To change the endianness, you can use the `swapbytes` function. For an example, see "Change the Endianness of the Data" on page 4-213.

This object supports C and C++ code generation.

To read data from a binary file:

1 Create a `dsp.BinaryFileReader` object and set the properties of the object.
2 Call step to read the binary file.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`reader = dsp.BinaryFileReader` creates a binary file reader object, `reader`, using the default properties.

`reader = dsp.BinaryFileReader(fname)` sets the `Filename` property to `fname`.

`reader = dsp.BinaryFileReader(fname, Name, Value, ...)` with `Filename` set to `fname`, and each property `Name` set to the specified `Value`. Unspecified properties have default values.

Example:

```
reader = dsp.BinaryFileReader('myFilename.bin','SamplesPerFrame',1000,'NumChannels',2);
```

Properties

`Filename` — Name of the file

`'Untitled.bin'` (default) | character vector

Name of the file from which the object reads the data, specified as a character vector. If the file is not on the MATLAB path, then specify the full path for the file.

`HeaderStructure` — Size of the header

`struct('Field1',[])` (default) | structure

Size of the structure, specified as a structure. The structure specifies the prototype of the file header, that is, the size of the header and the data type of the field values. The structure can have an arbitrary number of fields. Each field of the structure must be a real matrix of a built-in type. For example, if `HeaderStructure` is set to `struct('field1',1:10,'field2',single(1))`, the object assumes that the header is formed by 10 real double-precision values followed by 1 single-precision value. If the file contains no header, you can set this property to an empty structure, `struct([])`. To retrieve the file header, call the `readHeader` method on the reader object.

`SamplesPerFrame` — Number of samples per output frame

`1024` (default) | positive integer

Number of samples per output frame, specified as a positive integer. `SamplesPerFrame` specifies the number of rows of the output matrix that the object returns. The size of the data is `SamplesPerFrame`-by-`NumChannels`. Once the end of file is reached, the reader returns zeros of the specified data type, size, and complexity.

`NumChannels` — Number of channels

`1` (default) | positive integer

Number of channels, specified as a positive integer. `NumChannels` specifies the number of columns of the output matrix that the object returns. This property defines the number of consecutive interleaved data samples stored in the file for each time instant. The size of the data is `SamplesPerFrame`-by-`NumChannels`. Once the end of file is reached, if the output matrix is not full, the object fills the matrix with zeros to make it a full-sized matrix.

`DataType` — Storage class of data in file

`'double'` (default) | `'single'` | `'int8'` | `'int16'` | `'int32'` | `'int64'` | `'uint8'` | `'uint16'` | `'uint32'` | `'uint64'`

Storage class of data in file, specified as a character vector. This property defines the data type of the matrix returned by the `step` method.

`IsDataComplex` — Specify data complexity

`false` (default) | `true`

Option to specify data complexity, specified as false or true. When this property is set to `true`, the reader treats the data as being complex. The object reads the data as interleaved real and imaginary components. Consider a reader object configured to read the data as a 2-by-2 matrix. The object reads `[1 5 2 6 3 7 4 8]` as `[1 2; 3 4]+1j*[5 6; 7 8]`. If this property is set to `false`, the same object reads the data as `[1 5; 2 6]`.

Methods

isdone	End-of-file status (logical)
readheader	Read file header
reset	Reset internal states of System object
step	Read data from binary file

Common to All System Objects	
`clone`	Create System object with same property values

Common to All System Objects	
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

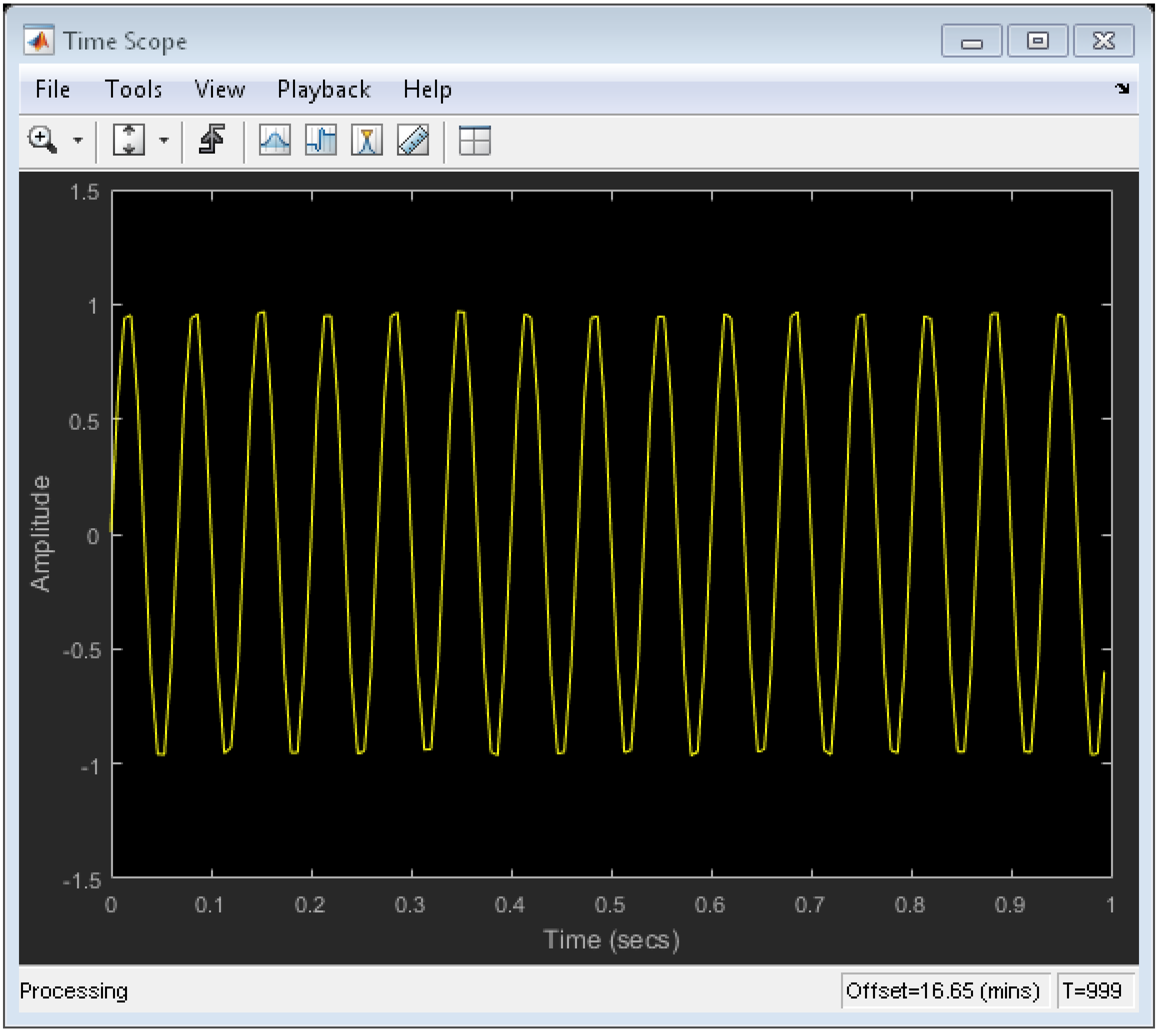
Write and Read Binary Files

Create a binary file with a custom header using the `dsp.BinaryFileWriter` System object™. Write data to this file. Read the header and data using the `dsp.BinaryFileReader` System object.

Write the Data

Specify the file header and create a `dsp.BinaryFileWriter` object. The object writes the header first, followed by the data, to the `ex_file.bin` file. The data is a noisy sine wave signal. View the data in a time scope.

```
header = struct('A',[1 2 3 4],'B','x7');
writer = dsp.BinaryFileWriter('ex_file.bin','HeaderStructure',header);
L = 150;
sine = dsp.SineWave('SamplesPerFrame',L);
scopewriter = dsp.TimeScope(1,'YLimits',[-1.5 1.5],'SampleRate',...
    L,'TimeSpan',1);

for i = 1:1000
    data = sine()+0.01*randn(L,1);
    writer(data);
    scopewriter(data);
end
```



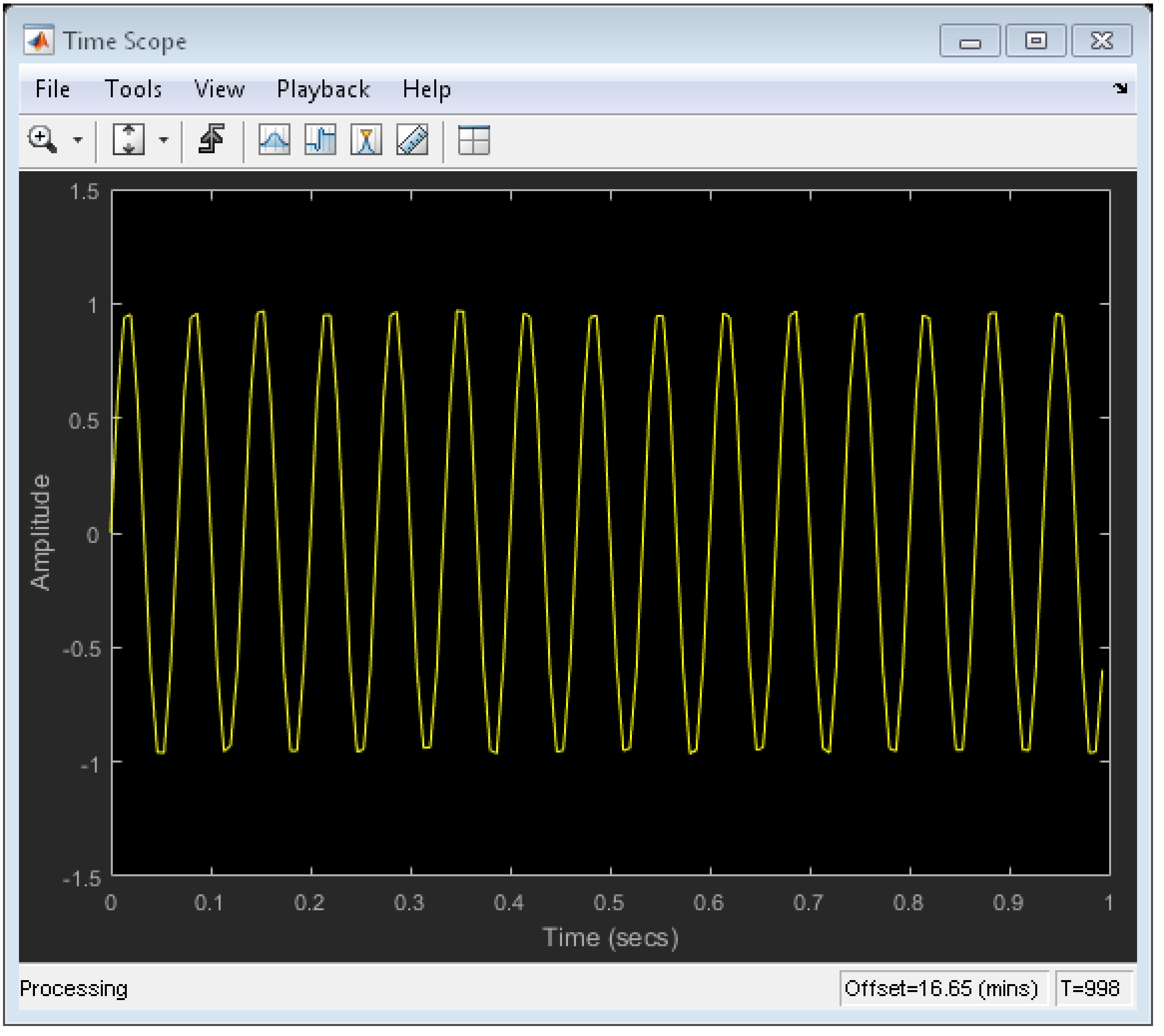
Release the writer so that the reader can access the data from this file.

```
release(writer);
```

Read the Data

Specify the header using the `HeaderStructure` property of the reader object. If the exact header is not known, you must at least specify the prototype of the header, that is, its size and data type. The dsp.BinaryFileReader object reads the binary data from ex_file.bin until the end of file is reached. The data is read into a single channel (column) containing multiple frames, where each frame has 300 samples. View the data in a time scope.

```
headerPrototype = struct('A',[0 0 0 0],'B','-0');

reader = dsp.BinaryFileReader(...
    'ex_file.bin',...
    'HeaderStructure',headerPrototype,...
    'NumChannels',1,'SamplesPerFrame',300);
scopereader = dsp.TimeScope(1,'YLimits',[-1.5 1.5],'SampleRate',...
    L,'TimeSpan',1);
while ~isDone(reader)
    out = reader();
    scopereader(out);
end
release(reader);
```



Even when the reader reads data with a different frame size, the output in both time scopes matches exactly.

Write and Read Matrix Data

Use a `dsp.BinaryFileReader` System object™ to read data from a binary file in a row-major format.

Write the Data

Write the matrix A to the binary file Matdata.bin using a dsp.BinaryFileWriter object. The object writes the specified header followed by the data.

```
A = [1 2 3 8; 4 5 6 10; 7 8 9 11];
header = struct('A',[1 2],'B','x7');
writer = dsp.BinaryFileWriter('Matdata.bin','HeaderStructure',header);
writer(A);
```

Release the writer so that the reader can access the data.

```
release(writer);
```

Read the Data

Specify the header using the HeaderStructure property of the reader object. If the exact header is not known, you must at least specify the prototype of the header, that is, its size and data type. The dsp.BinaryFileReader object reads the binary file Matdata.bin until the end of file is reached. Specify the System object to read the data into 4 channels, with each channel containing 5 samples. Each loop of the iteration reads a channel (or frame) of data.

```
headerPrototype = struct('A',[0 0],'B','-0');
reader = dsp.BinaryFileReader('Matdata.bin','HeaderStructure',header,...
    'NumChannels',4,'SamplesPerFrame',5);
while ~isDone(reader)
    out = reader();
    display(out)
end
```

```
out =

     1     2     3     8
     4     5     6    10
     7     8     9    11
     0     0     0     0
     0     0     0     0
```

Each frame of out contains frames of the matrix A, followed by zeros to complete the frame. The original matrix A contains 4 channels with 3 samples in each channel. The reader is specified to read data into 4 channels, with each channel containing 5 samples.

Because there are not enough samples to complete the frame, the reader object appends zeros at the end of each frame.

Read Header Data

Read the header data from a binary file using the `readHeader` method.

Write a header, followed by the data to a binary file named `myfile.dat`. The header is a 1-by-4 matrix of double precision values, followed by a 5-by-1 vector of single-precision values. The data is a sequence of 1000 double-precision values.

```
fid = fopen('myfile.dat','w');
fwrite(fid,[1 2 3 4],'double');
fwrite(fid,single((1:5).'),'single');
fwrite(fid,(1:1000).','double');
fclose(fid);
```

Read the header using a `dsp.BinaryFileReader` object. Specify the expected header structure. This structure specifies only the format of the expected binary file header and does not contain the exact values.

```
reader = dsp.BinaryFileReader('myfile.dat');
s = struct('A',zeros(1,4),'B',ones(5,1,'single'));
reader.HeaderStructure = s;
```

Read the header using the `readHeader` method of the `dsp.BinaryFileReader` object.

```
H = readHeader(reader);
fprintf('H.A: ')
fprintf('%d ',H.A);
fprintf('\nH.A datatype: %s\n',class(H.A))
fprintf('H.B: ')
fprintf('%d ',H.B);
fprintf('\nH.B datatype: %s\n',class(H.B))
```

```
H.A: 1 2 3 4
H.A datatype: double
H.B: 1 2 3 4 5
H.B datatype: single
```

Read Complex Data

Read complex data from a binary file using the `dsp.BinaryFileReader` object.

Write a sequence of numbers to a binary file named `myfile.dat`. There is no header. The data is a 2-by-4 matrix of double-precision values. `fwrite` writes the data in a column-major format. That is, the 2-by-4 matrix `[1 2 3 4; 9 10 11 12]` is written as `[1 9 2 10 3 11 4 12]` in the binary file.

```
fid = fopen('myfile.dat','w');
fwrite(fid,[1 2 3 4; 9 10 11 12],'double');
fclose(fid);
```

Specify the data to be complex using the `IsDataComplex` property. The object reads the data as interleaved real and imaginary components. The `SamplesPerFrame` and `NumChannel` properties specify the number of rows and columns of the output data. The header structure is specified as empty.

```
reader = dsp.BinaryFileReader('myfile.dat','SamplesPerFrame',2,...
    'NumChannels',2, 'IsDataComplex',true);
s = struct([]);
reader.HeaderStructure = s;
data = reader();
display(data);
release(reader);
```

```
data =

   1.0000 + 9.0000i   2.0000 +10.0000i
   3.0000 +11.0000i   4.0000 +12.0000i
```

Alternatively, if you do not specify the data to be complex, the reader reads the data as a `SamplesPerFrame`-by- `NumChannel` matrix of real values.

```
reader.IsDataComplex = false;
data = reader();
display(data);
release(reader);
```

```
data =

     1     9
     2    10
```

Write and Read Fixed-Point Data

The `dsp.BinaryFileWriter` and `dsp.BinaryFileReader` System objects do not support writing and reading fixed-point data. As a workaround, you can write the stored integer portion of the `fi` data, read the data, and use this value to reconstruct the `fi` data.

Write the Fixed-Point Data

Create a `fi` object to represent 100 signed random numbers with a word length of 14 and a fraction length of 12. Write the stored integer portion of the `fi` object to the data file `myFile.dat`. The built-in data type is `int16`, which can be computed using `class(storeIntData)`.

```
data = randn(100,1);
fiDataWriter = fi(data,1,14,12);
storeIntData = storedInteger(fiDataWriter);

writer = dsp.BinaryFileWriter('myFile.dat');
writer(storeIntData);
```

Release the writer so that the reader can access the data.

```
release(writer);
```

Read the Fixed-Point Data

Specify the reader to read the stored integer data as `int16` data with 100 samples per data frame. The real-world value of the fixed-point number can be represented using $2^{(-fractionLength)*(storedInteger)}$. If you know the signedness, word length, and fraction length of the fixed-point data, you can reconstruct the `fi` data using $fi(realValue, signedness, wordLength, fractionLength)$. In this example, the data is signed with a word length of 14 and a fraction length of 12.

```
reader = dsp.BinaryFileReader('Filename','myFile.dat','SamplesPerFrame',100,...
    'DataType','int16');
data = reader();
fractionLength = 12;
wordLength = 14;
realValue = 2^(-fractionLength)*double(data);

fiDataReader = fi(realValue,1,wordLength,fractionLength);
```

Verify that the writer data is the same as the reader data.

```
isequal(fiDataWriter,fiDataReader)
```

```
ans =

  logical

   1
```

Write and Read Character Data

The `dsp.BinaryFileWriter` and `dsp.BinaryFileReader` System objects do not support writing and reading characters. As a workaround, cast character data to one of the built-in data types and write the integer data. After the reader reads the data, convert the data to a character using the `char` function.

Write the Character Data

Cast a character into `uint8` using the `cast` function. Write the cast data to the data file `myFile.dat`.

```
data = 'binary_file';
castData = cast(data,'uint8');
writer = dsp.BinaryFileWriter('myFile.dat');
writer(castData);
```

Release the writer so that the reader can access the data.

```
release(writer);
```

Read the `uint8` Data

Specify the reader to read the cast data as `uint8` data.

```
reader = dsp.BinaryFileReader('myFile.dat','DataType','uint8','SamplesPerFrame',11);
readerData = reader();
charData = char(readerData);
```

Verify that the writer data is the same as the reader data. By default, the reader returns the data in a column-major format.

```
strcmp(data,charData.')
```

```
ans =

  logical
```

```
    1
```

Change the Endianness of the Data

By default, the `dsp.BinaryFileReader` System object™ uses the endianness of the host machine. To change the endianness, such as when the host machine that writes the data does not have the same endianness as the host machine that reads the data, use the `swapbytes` function.

Write a numeric array into `myfile.dat` in big endian format. Read the data using the `dsp.BinaryFileReader` object. The reader object reads the data in little endian format.

```
fid = fopen('myfile.dat','w','b');
fwrite(fid,[1 2 3 4 5 6 7 8],'double');
fclose(fid);
reader = dsp.BinaryFileReader('myfile.dat','SamplesPerFrame',8);
x = reader();
display(x);
```

```
x =

  1.0e-318 *

    0.3039
    0.0003
    0.0104
    0.0206
    0.0256
    0.0307
    0.0357
    0.0408
```

`x` does not match the original data. Change the endianness of `x` using the `swapbytes` function.

```
y = swapbytes(x);
display(y);
```

```
y =
```

y matches the original data.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

System Objects
dsp.BinaryFileWriter

Blocks
`Binary File Reader` | `Binary File Writer`

Introduced in R2016b

isdone

System object: dsp.BinaryFileReader
Package: dsp

End-of-file status (logical)

Syntax

```
status = isDone(reader)
```

Description

`status = isDone(reader)` returns true if the source System object, `obj`, has reached the end of the file.

Introduced in R2016b

readheader

System object: dsp.BinaryFileReader
Package: dsp

Read file header

Syntax

```
header = readHeader(reader)
```

Description

`header = readHeader(reader)` returns the header structure, `header`, from the file specified by the binary file reader, `reader`.

Introduced in R2016b

reset

System object: dsp.BinaryFileReader
Package: dsp

Reset internal states of System object

Syntax

```
reset(reader)
```

Description

`reset(reader)` resets to the beginning of the file and skips the header bytes.

Introduced in R2016b

step

System object: dsp.BinaryFileReader
Package: dsp

Read data from binary file

Syntax

`data = step(reader)`

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`data = step(reader)` reads data from the binary file in row-major format. The data type, size, and complexity of the data are determined by the properties of the `reader` object. Once end of file is reached, the `step` method returns zeros of the specified data type, size, and complexity.

Introduced in R2016b

dsp.BinaryFileWriter System object

Package: dsp

Write data to binary files

Description

The `dsp.BinaryFileWriter` System object writes multichannel signal data to a binary file. If the header is not empty, then the header precedes the signal data. The object specifies the file name and the structure of the header. The first time you write to the file, the object writes the header, followed by the data. On subsequent calls, the object writes the remaining data. If the header is empty, then no header is written.

The object can write floating-point data and integer data. To write character data and fixed-point data, see "Write and Read Character Data" on page 4-228 and "Write and Read Fixed-Point Data" on page 4-226. The input data can be real or complex. When the data is complex, the object writes the data as interleaved real and imaginary components. For an example, see "Write and Read Fixed-Point Data" on page 4-226. The writer assumes the default endianness of the host machine. To change the endianness, you can use the `swapbytes` function. For an example, see "Change the Endianness of the Data Before Writing the Data" on page 4-229 .

The object writes the data in row-major format. For example if the input array is [`1 2 4 5; 8 7 9 2`], the object writes the data as [`1 2 4 5 8 7 9 2`].

The object supports C and C++ code generation.

To write data to a binary file:

1. Create a `dsp.BinaryFileWriter` object and set the properties of the object.
2. Call step to write to the binary file. In the first call to `step`, the object writes the header before the data. In subsequent calls, the object writes the data to the end of the file.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`writer = dsp.BinaryFileWriter` creates a binary file writer object, `writer`, using the default properties.

`writer = dsp.BinaryFileWriter(fname)` sets the `Filename` property to `fname`.

`writer = dsp.BinaryFileWriter(fname,Name,Value, ...)` with `Filename` set to `fname`, and each property `Name` set to the specified `Value`. Unspecified properties have default values.

Example:

```
writer = dsp.BinaryFileWriter('myFilename.bin','HeaderStructure',struct('field1',1:10,
```

Properties

`Filename` — Name of the file

`'untitled.bin'` (default) | character vector

Name of the file to which the object writes the data, specified as a character vector. You must specify the full path for the file.

`HeaderStructure` — Header to write at the beginning of the file

`struct([])` (default) | structure

Header to write at the beginning of the file, specified as a structure. The structure can have an arbitrary number of fields. Each field of the structure must be a real matrix of a built-in type. For example, if `HeaderStructure` is set to `struct('field1',1:10,'field2',single(1))`, the object writes a header formed by 10 double-precision values, `(1:10)`, followed by one single precision value, `single(1)`. If you do not specify a header, the object sets this property to an empty structure, `struct([])`.

Methods

reset	Reset internal states of System object

step	Write data to the binary file

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

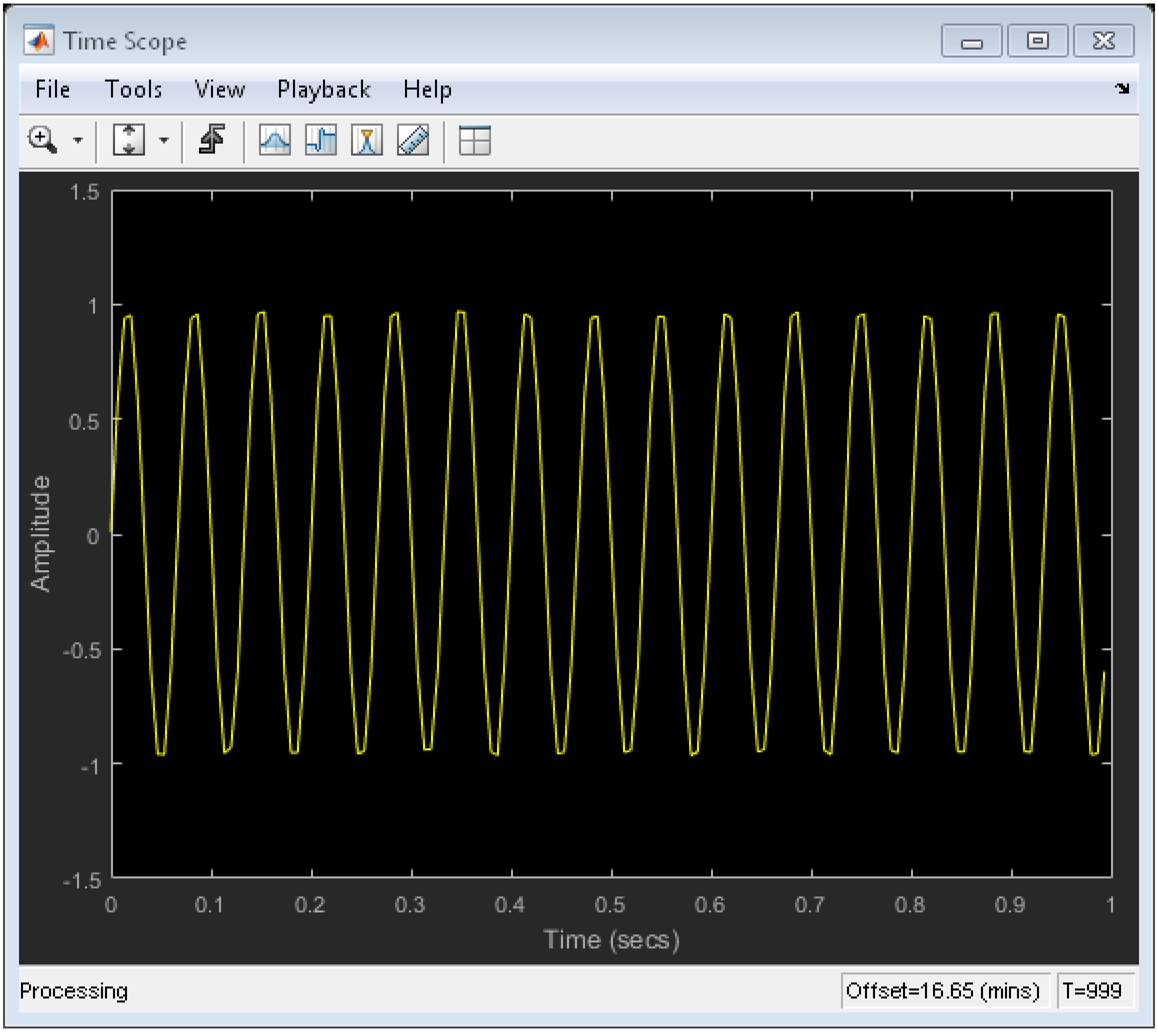
Write and Read Binary Files

Create a binary file with a custom header using the `dsp.BinaryFileWriter` System object™. Write data to this file. Read the header and data using the `dsp.BinaryFileReader` System object.

Write the Data

Specify the file header and create a `dsp.BinaryFileWriter` object. The object writes the header first, followed by the data, to the `ex_file.bin` file. The data is a noisy sine wave signal. View the data in a time scope.

```
header = struct('A',[1 2 3 4],'B','x7');
writer = dsp.BinaryFileWriter('ex_file.bin','HeaderStructure',header);
L = 150;
sine = dsp.SineWave('SamplesPerFrame',L);
scopewriter = dsp.TimeScope(1,'YLimits',[-1.5 1.5],'SampleRate',...
    L,'TimeSpan',1);

for i = 1:1000
    data = sine()+0.01*randn(L,1);
    writer(data);
    scopewriter(data);
end
```



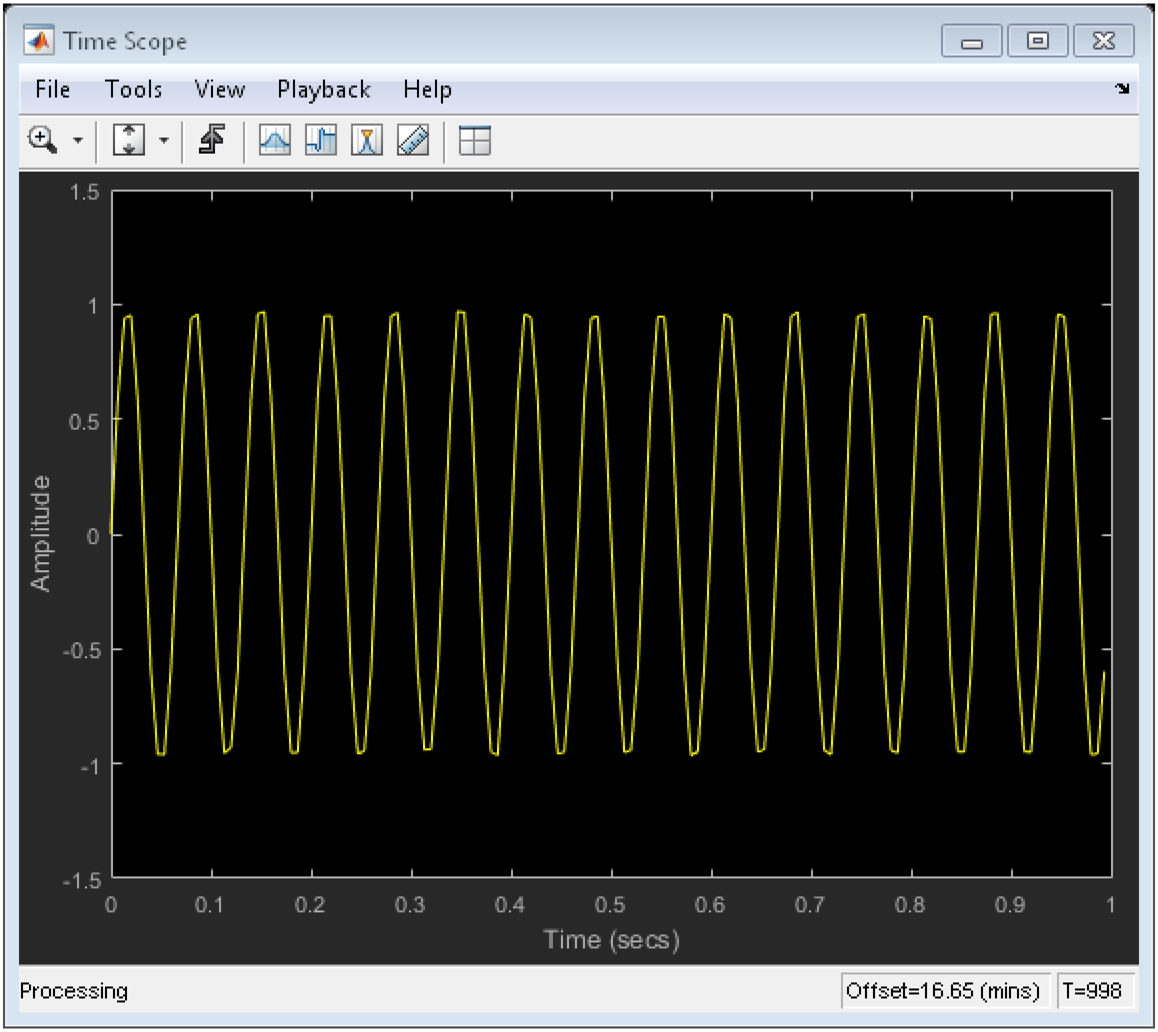
Release the writer so that the reader can access the data from this file.

```
release(writer);
```

Read the Data

Specify the header using the `HeaderStructure` property of the reader object. If the exact header is not known, you must at least specify the prototype of the header, that is, its size and data type. The `dsp.BinaryFileReader` object reads the binary data from `ex_file.bin` until the end of file is reached. The data is read into a single channel (column) containing multiple frames, where each frame has 300 samples. View the data in a time scope.

```
headerPrototype = struct('A',[0 0 0 0],'B','-0');

reader = dsp.BinaryFileReader(...
    'ex_file.bin',...
    'HeaderStructure',headerPrototype,...
    'NumChannels',1,'SamplesPerFrame',300);
scopereader = dsp.TimeScope(1,'YLimits',[-1.5 1.5],'SampleRate',...
    L,'TimeSpan',1);
while ~isDone(reader)
    out = reader();
    scopereader(out);
end
release(reader);
```



Even when the reader reads data with a different frame size, the output in both time scopes matches exactly.

Write and Read Matrix Data

Use a `dsp.BinaryFileReader` System object™ to read data from a binary file in a row-major format.

Write the Data

Write the matrix A to the binary file Matdata.bin using a dsp.BinaryFileWriter object. The object writes the specified header followed by the data.

```
A = [1 2 3 8; 4 5 6 10; 7 8 9 11];
header = struct('A',[1 2],'B','x7');
writer = dsp.BinaryFileWriter('Matdata.bin','HeaderStructure',header);
writer(A);
```

Release the writer so that the reader can access the data.

```
release(writer);
```

Read the Data

Specify the header using the HeaderStructure property of the reader object. If the exact header is not known, you must at least specify the prototype of the header, that is, its size and data type. The dsp.BinaryFileReader object reads the binary file Matdata.bin until the end of file is reached. Specify the System object to read the data into 4 channels, with each channel containing 5 samples. Each loop of the iteration reads a channel (or frame) of data.

```
headerPrototype = struct('A',[0 0],'B','-0');
reader = dsp.BinaryFileReader('Matdata.bin','HeaderStructure',header,...
    'NumChannels',4,'SamplesPerFrame',5);
while ~isDone(reader)
    out = reader();
    display(out)
end
```

```
out =

     1     2     3     8
     4     5     6    10
     7     8     9    11
     0     0     0     0
     0     0     0     0
```

Each frame of out contains frames of the matrix A, followed by zeros to complete the frame. The original matrix A contains 4 channels with 3 samples in each channel. The reader is specified to read data into 4 channels, with each channel containing 5 samples.

Because there are not enough samples to complete the frame, the reader object appends zeros at the end of each frame.

Write Complex Data

This example shows how the `dsp.BinaryFileWriter` object writes complex data.

Create a `dsp.BinaryFileWriter` object which writes to a file named 'myfile.dat'. There is no header. The data is complex.

```
writer = dsp.BinaryFileWriter('myfile.dat');
data = [1 2 3 4]+1i*[5 6 7 8];
writer(data);
release(writer);
```

Read the data using the `dsp.BinaryFileReader` System object™. To view data in the format it is written to the file, set the `IsDataComplex` property to `false`. The reader object reads the data as a sequence of numbers in a row major format. Set `SamplesPerFrame` to 1 and `NumChannels` to 8.

```
reader = dsp.BinaryFileReader('myfile.dat','SamplesPerFrame',1,...
    'NumChannels',8);
s = struct([]);
reader.HeaderStructure = s;
dataRead = reader();
```

You can see that the real and imaginary components of the original data are sample interleaved.

```
display(dataRead);
```

```
dataRead =

     1     5     2     6     3     7     4     8
```

Write and Read Fixed-Point Data

The `dsp.BinaryFileWriter` and `dsp.BinaryFileReader` System objects do not support writing and reading fixed-point data. As a workaround, you can write the stored integer portion of the `fi` data, read the data, and use this value to reconstruct the `fi` data.

Write the Fixed-Point Data

Create a `fi` object to represent 100 signed random numbers with a word length of 14 and a fraction length of 12. Write the stored integer portion of the `fi` object to the data file `myFile.dat`. The built-in data type is `int16`, which can be computed using `class(storeIntData)`.

```
data = randn(100,1);
fiDataWriter = fi(data,1,14,12);
storeIntData = storedInteger(fiDataWriter);

writer = dsp.BinaryFileWriter('myFile.dat');
writer(storeIntData);
```

Release the writer so that the reader can access the data.

```
release(writer);
```

Read the Fixed-Point Data

Specify the reader to read the stored integer data as `int16` data with 100 samples per data frame. The real-world value of the fixed-point number can be represented using $2^{(-fractionLength)*(storedInteger)}$. If you know the signedness, word length, and fraction length of the fixed-point data, you can reconstruct the `fi` data using $fi(realValue, signedness, wordLength, fractionLength)$. In this example, the data is signed with a word length of 14 and a fraction length of 12.

```
reader = dsp.BinaryFileReader('Filename','myFile.dat','SamplesPerFrame',100,...
    'DataType','int16');
data = reader();
fractionLength = 12;
wordLength = 14;
realValue = 2^(-fractionLength)*double(data);

fiDataReader = fi(realValue,1,wordLength,fractionLength);
```

Verify that the writer data is the same as the reader data.

```
isequal(fiDataWriter,fiDataReader)
```

```
ans =

  logical
```

```
     1
```

Write and Read Character Data

The `dsp.BinaryFileWriter` and `dsp.BinaryFileReader` System objects do not support writing and reading characters. As a workaround, cast character data to one of the built-in data types and write the integer data. After the reader reads the data, convert the data to a character using the `char` function.

Write the Character Data

Cast a character into `uint8` using the `cast` function. Write the cast data to the data file `myFile.dat`.

```
data = 'binary_file';
castData = cast(data,'uint8');
writer = dsp.BinaryFileWriter('myFile.dat');
writer(castData);
```

Release the writer so that the reader can access the data.

```
release(writer);
```

Read the `uint8` Data

Specify the reader to read the cast data as `uint8` data.

```
reader = dsp.BinaryFileReader('myFile.dat','DataType','uint8','SamplesPerFrame',11);
readerData = reader();
charData = char(readerData);
```

Verify that the writer data is the same as the reader data. By default, the reader returns the data in a column-major format.

```
strcmp(data,charData.')
```

```
ans =

  logical

   1
```

Change the Endianness of the Data Before Writing the Data

By default, the `dsp.BinaryFileWriter` System object™ uses the endianness of the host machine. To change the endianness, use the `swapbytes` function.

Write a numeric array into `myfile.dat` using the `dsp.BinaryFileWriter` object. Before writing the data, change the endianness of the data using the `swapbytes` function.

```
data = [1 2 3 4 2 2];
swapData = swapbytes(data);
writer = dsp.BinaryFileWriter('myfile.dat');
writer(swapData);
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

System Objects

dsp.BinaryFileReader

Blocks

`Binary File Reader` | `Binary File Writer`

Introduced in R2016b

reset

System object: dsp.BinaryFileWriter
Package: dsp

Reset internal states of System object

Syntax

`reset(writer)`

Description

`reset(writer)` resets the position in the file at which the writer object writes the data. Upon calling reset, the writer object starts writing the data where the header data ends.

Introduced in R2016b

step

System object: dsp.BinaryFileWriter
Package: dsp

Write data to the binary file

Syntax

`step(writer,data)`

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`step(writer,data)` writes data to the binary file in row-major format. Each call to `step` writes the elements of `data` at the end of the file. At the first call to `step`, the object writes the header first, followed by the data. If the header is empty, then no header is written.

The input data can be real or complex. For complex data, real and imaginary parts are interleaved. For example, if `data = [1 2; 3 4]+1j*[5 6; 7 8]`, then the object writes the elements as `1 5 2 6 3 7 4 8`.

Introduced in R2016b

dsp.BiquadFilter System object

Package: dsp

IIR filter using biquadratic structures

Description

The `BiquadFilter` object implements an IIR filter structure using biquadratic or second–order sections (SOS).

To implement an IIR filter structure using biquadratic or SOS:

1. Define and set up your biquadratic IIR filter. See “Construction” on page 4-232.
2. Call step to implement the IIR filter according to the properties of `dsp.BiquadFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`biquad = dsp.BiquadFilter` returns a default biquadratic IIR filter, `biquad`, which independently filters each channel of the input over successive calls to the `step` method using the SOS section`[1 0.3 0.4 1 0.1 0.2]` with a direct-form II transposed structure.

`biquad = dsp.BiquadFilter('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a biquadratic filter object, `biquad`, with each property set to the specified value.

Properties

`Structure`

Filter structure

Specify the filter structure as one of | Direct form I | Direct form I transposed | Direct form II | Direct form II transposed |. The default is Direct form II transposed.

SOSMatrixSource

SOS matrix source

Specify the source of the SOS matrix as one of | Property | Input port|. The default is Property.

SOSMatrix

SOS matrix

Specify the second-order section (SOS) matrix as an N-by-6 matrix, where N is the number of sections in the filter. The default is [1 0.3 0.4 1 0.1 0.2]. Each row of the SOS matrix contains the numerator and denominator coefficients of the corresponding section of the filter. The system function, *H(z)*, of a biquad filter is:

$$H(z) = \frac{\sum_{k=0}^{2} b_k z^{-k}}{1 - \sum_{l=1}^{2} a_l z^{-l}}$$

The coefficients are ordered in the rows of the SOS matrix as (b_0, b_1,b_2, 1, $-a_1$, $-a_2$). You can use coefficients of real or complex values. This property applies only when you set the SOSMatrixSource on page 4-233 property to Property. The leading denominator coefficient of the biquad filter, a_0, equals 1 for each filter section, regardless of the specified value.

ScaleValues

Scale values for each biquad section

Specify the scale values to apply before and after each section of a biquad filter. ScaleValues must be either a scalar or a vector of length N+1, where N is the number of sections. If you set this property to a scalar, the scalar value is used as the gain value only before the first filter section. The remaining gain values are set to 1. If you set this property to a vector of N+1values, each value is used for a separate section of the filter.

This property applies only when you set the SOSMatrixSource on page 4-233 property to Property. The default is 1.

InitialConditions

Initial conditions for direct form II structures

Specify the initial conditions of the filter states when the Structure on page 4-232 property is one of | Direct form II | Direct form II transposed |. The number of states or delay elements (zeros and poles) in a direct-form II biquad filter equals twice the number of filter sections. You can specify the initial conditions as a scalar, vector, or matrix.

When you specify a scalar value, the biquad filter initializes all delay elements in the filter to that value. When you specify a vector of length equal to the number of delay elements in the filter, each vector element specifies a unique initial condition for the corresponding delay element.

The biquad filter applies the same vector of initial conditions to each channel of the input signal. When you specify a vector of length equal to the product of the number of input channels and the number of delay elements in the filter, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel. When you specify a matrix with the same number of rows as the number of delay elements in the filter, and one column for each channel of the input signal, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel. The default is the scalar 0.

NumeratorInitialConditions

Initial conditions on zeros side

Specify the initial conditions of the filter states on the side of the filter structure with the zeros. This property applies only when you set the Structure on page 4-232 property to one of | Direct form I | Direct form I transposed |. The number of states or delay elements in the numerator of a direct-form I biquad filter equals twice the number of filter sections. You can specify the initial conditions as a scalar, vector, or matrix. When you specify a scalar, the biquad filter initializes all delay elements on the zeros side in the filter to that value. When you specify a vector of length equal to the number of delay elements on the zeros side in the filter, each vector element specifies a unique initial condition for the corresponding delay element on the zeros side.

The biquad filter applies the same vector of initial conditions to each channel of the input signal. When you specify a vector of length equal to the product of the number of input channels and the number of delay elements on the zeros side in the filter, each element specifies a unique initial condition for the corresponding delay element on the zeros side in the corresponding channel. When you specify a matrix with the same number of rows as the number of delay elements on the zeros side in the filter, and one column for each channel of the input signal, each element specifies a unique initial condition for the corresponding delay element on the zeros side in the corresponding channel. The default is the scalar `0`.

`DenominatorInitialConditions`

Initial conditions on poles side

Specify the initial conditions of the filter states on the side of the filter structure with the poles. This property only applies when you set the `Structure` on page 4-232 property to one of `| Direct form I | Direct form I transposed |`. The number of denominator states, or delay elements, in a direct-form I (noncanonic) biquad filter equals twice the number of filter sections. You can specify the initial conditions as a scalar, vector, or matrix. When you specify a scalar, the biquad filter initializes all delay elements on the poles side of the filter to that value. When you specify a vector of length equal to the number of delay elements on the poles side in the filter, each vector element specifies a unique initial condition for the corresponding delay element on the poles side.

The object applies the same vector of initial conditions to each channel of the input signal. When you specify a vector of length equal to the product of the number of input channels and the number of delay elements on the poles side in the filter, each element specifies a unique initial condition for the corresponding delay element on the poles side in the corresponding channel. When you specify a matrix with the same number of rows as the number of delay elements on the poles side in the filter, and one column for each channel of the input signal, each element specifies a unique initial condition for the corresponding delay element on the poles side in the corresponding channel. The default is the scalar `0`.

`OptimizeUnityScaleValues`

Optimize unity scale values

When this Boolean property is set to `true`, the biquad filter removes all unity scale gain computations. This reduces the number of computations and increases the fixed-point accuracy. This property applies only when you set the `SOSMatrixSource` on page 4-233 property to `Property`. The default is `true`.

`ScaleValuesInputPort`

How to specify scale values

Select how to specify scale values. This property applies only when the `SOSMatrixSource` on page 4-233 property is `Input port`. By default, this property is `true`, and the scale values are specified via the input port. When this property is `false`, all scale values are 1.

Fixed-Point Properties

`RoundingMethod`

Rounding method for fixed-point operations

Specify the rounding method as one of `| Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |`. The default is `Floor`.

`OverflowAction`

Overflow action for fixed-point operations

Specify the overflow action as one of `| Wrap | Saturate |`. The default is `Wrap`.

`MultiplicandDataType`

Multiplicand word and fraction lengths

Specify the multiplicand fixed-point data type as one of `| Same as output | Custom |`. This property applies only when you set the `Structure` on page 4-232 property to `Direct form I transposed`. The default is `Same as output`.

`CustomMultiplicandDataType`

Custom multiplicand word and fraction lengths

Specify the multiplicand fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the

`MultiplicandDataType` on page 4-236 property to `Custom`. The default is `numerictype([],32,30)`.

SectionInputDataType

Section input word and fraction lengths

Specify the section input fixed-point data type as one of | `Same as input` | `Custom` |. The default is `Same as input`.

CustomSectionInputDataType

Custom section input word and fraction lengths

Specify the section input fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `SectionInputDataType` on page 4-237 property to `Custom`. The default is `numerictype([],16,15)`.

SectionOutputDataType

Section output word and fraction lengths

Specify the section output fixed-point data type as one of | `Same as section input` | `Custom` |. The default is `Same as section input`.

CustomSectionOutputDataType

Custom section output word and fraction lengths

Specify the section output fixed-point type as a signed, scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `SectionOutputDataType` on page 4-237 property to `Custom`. The default is `numerictype([],16,15)`.

NumeratorCoefficientsDataType

Numerator coefficients word and fraction lengths

Specify the numerator coefficients fixed-point data type as one of | `Same word length as input` | `Custom` |. This property applies only when you set the `SOSMatrixSource` on page 4-233 property to `Property`. Setting this property also sets the `DenominatorCoefficientsDataType` on page 4-238 and `ScaleValuesDataType` on page 4-238 properties to the same value. The default is `Same word length as input`.

CustomNumeratorCoefficientsDataType

Custom numerator coefficients word and fraction lengths

Specify the numerator coefficients fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `SOSMatrixSource` on page 4-233 property to `Property` and the `NumeratorCoefficientsDataType` on page 4-237 property to `Custom`. The word length of the `CustomNumeratorCoefficientsDataType` on page 4-238, `CustomDenominatorCoefficientsDataType` on page 4-238, and `CustomScaleValuesDataType` on page 4-239 properties must be the same. The default is `numerictype([],16,15)`.

DenominatorCoefficientsDataType

Denominator coefficients word and fraction lengths

Specify the denominator coefficients fixed-point data type as one of `| Same word length as input | Custom |`. This property applies only when you set the `SOSMatrixSource` on page 4-233 property to `Property`. Setting this property also sets the `NumeratorCoefficientsDataType` on page 4-237 and `ScaleValuesDataType` on page 4-238 properties to the same value. The default is `Same word length as input`.

CustomDenominatorCoefficientsDataType

Custom denominator coefficients word and fraction lengths

Specify the denominator coefficients fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `SOSMatrixSource` on page 4-233 property to `Property` and the `DenominatorCoefficientsDataType` on page 4-238 property to `Custom`. The `CustomNumeratorCoefficientsDataType` on page 4-238, `CustomDenominatorCoefficientsDataType` on page 4-238, and `CustomScaleValuesDataType` on page 4-239 properties must have the same word lengths. The default is `numerictype([],16,15)`.

ScaleValuesDataType

Scale values word and fraction lengths

Specify the scale values fixed-point data type as one of | Same word length as input | Custom |. This property applies only when you set the SOSMatrixSource on page 4-233 property to Property. Setting this property also sets the NumeratorCoefficientsDataType on page 4-237 and DenominatorCoefficientsDataType on page 4-238 properties to the same value. The default is Same word length as input.

CustomScaleValuesDataType

Custom scale values word and fraction lengths

Specify the scale values fixed-point type as a numerictype object with a Signedness of Auto. This property applies only when you set the SOSMatrixSource on page 4-233 property to Property and the ScaleValuesDataType on page 4-238 property to Custom. The CustomNumeratorCoefficientsDataType on page 4-238, CustomDenominatorCoefficientsDataType on page 4-238, and CustomScaleValuesDataType on page 4-239 properties must have the same word lengths. The default is numerictype([],16,15).

NumeratorProductDataType

Numerator product word and fraction lengths

Specify the mode to determine the numerator product fixed-point data type as:

- Same as input (default) — The numerator product word and fraction lengths are same as that of the input.
- Custom — Enables the CustomNumeratorProductDataType property, which you can use to specify the custom numerator product data type. Specify the data type as a numerictype object.
- Full precision — Use full-precision rules to specify the data type. These rules provide the most accurate fixed-point numerics. The rules prevent quantization from occurring within the object. Bits are added, as needed, so that no roundoff or overflow occurs. For more information, see “Full Precision for Fixed-Point System Objects”.

Setting this property also sets the DenominatorProductDataType on page 4-240 property to the same value.

CustomNumeratorProductDataType

Custom numerator product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `NumeratorProductDataType` on page 4-239 property to `Custom`. The `CustomNumeratorProductDataType` on page 4-239 and `CustomDenominatorProductDataType` on page 4-240 properties must have the same word lengths. The default is `numerictype([],32,30)`.

DenominatorProductDataType

Denominator product word and fraction lengths

Specify the mode to determine the denominator product fixed-point data type as:

- `Same as input` (default) — The denominator product word and fraction lengths are same as that of the input.
- `Custom` — Enables the `CustomDenominatorProductDataType` property, which you can use to specify the custom denominator product data type. Specify the data type as a `numerictype` object.
- `Full precision` — Use full-precision rules to specify the data type. These rules provide the most accurate fixed-point numerics. The rules prevent quantization from occurring within the object. Bits are added, as needed, so that no roundoff or overflow occurs. For more information, see "Full Precision for Fixed-Point System Objects".

Setting this property also sets the `NumeratorProductDataType` on page 4-239 property to the same value.

CustomDenominatorProductDataType

Custom denominator product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `DenominatorProductDataType` on page 4-240 to `Custom`. The `CustomNumeratorProductDataType` on page 4-239 and `CustomDenominatorProductDataType` on page 4-239 properties must have the same word lengths. The default is `numerictype([],32,30)`.

NumeratorAccumulatorDataType

Numerator accumulator word and fraction lengths

Specify the numerator accumulator fixed-point data type as one of `| Same as input | Same as product | Custom |`. Setting this property also sets the

`DenominatorAccumulatorDataType` on page 4-241 property to the same value. The default is `Same as product`.

CustomNumeratorAccumulatorDataType

Custom numerator accumulator word and fraction lengths

Specify the numerator accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `NumeratorAccumulatorDataType` on page 4-240 property to `Custom`. The `CustomNumeratorAccumulatorDataType` on page 4-241 and `CustomDenominatorAccumulatorDataType` on page 4-241 properties must have the same word lengths. The default is `numerictype([],32,30)`.

DenominatorAccumulatorDataType

Denominator accumulator word and fraction lengths

Specify the denominator accumulator fixed-point data type as one of `| Same as input | Same as product | Custom|`. Setting this property also sets the `NumeratorAccumulatorDataType` on page 4-240 property to the same value. The default is `Same as product`.

CustomDenominatorAccumulatorDataType

Custom denominator accumulator word and fraction lengths

Specify the denominator accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `DenominatorAccumulatorDataType` on page 4-241 property to `Custom`. The `CustomNumeratorAccumulatorDataType` on page 4-241 and `CustomDenominatorAccumulatorDataType` on page 4-241 properties must have the same word lengths. The default is `numerictype([],32,30)`.

StateDataType

State word and fraction lengths

Specify the state fixed-point data type as one of `| Same as input | Same as accumulator | Custom|`. This property applies when you set the `Structure` on page 4-232 property to `Direct form II` or `Direct form II transposed`. The default is `Same as accumulator`.

CustomStateDataType

Custom state word and fraction lengths

Specify the state fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `StateDataType` on page 4-241 property to `Custom`. The default is `numerictype([],16,15)`.

NumeratorStateDataType

Numerator state word and fraction lengths

Specify the numerator state fixed-point data type as one of `| Same as input | Same as accumulator | Custom |`. Setting this property also sets the `DenominatorStateDataType` on page 4-242 property to the same value. This property applies only when you set the `Structure` on page 4-232 property to `Direct form I transposed`. The default is `Same as accumulator`.

CustomNumeratorStateDataType

Custom numerator state word and fraction lengths

Specify the numerator state fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `StateDataType` on page 4-241 property to `Custom`. The `CustomNumeratorProductDataType` on page 4-239 and `CustomDenominatorProductDataType` on page 4-240 properties must have the same word lengths. The default is `numerictype([],16,15)`.

DenominatorStateDataType

Denominator state word and fraction lengths

Specify the denominator state fixed-point data type as one of `| Same as input | Same as accumulator | Custom |`. Setting this property also sets the `NumeratorStateDataType` on page 4-242 property to the same value. This property applies only when you set the `Structure` on page 4-232 property to `Direct form I transposed`. The default is `Same as accumulator`.

CustomDenominatorStateDataType

Custom denominator state word and fraction lengths

Specify the denominator state fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `StateDataType` on page 4-241 property to `Custom`. The `CustomNumeratorStateDataType` on page 4-242 and `CustomDenominatorStateDataType` on page 4-242 properties must have the same word lengths. The default is `numerictype([],16,15)`.

`OutputDataType`

Output word and fraction lengths

Specify the output fixed-point data type as one of `| Same as input | Same as accumulator | Custom |`. The default is `Same as accumulator`.

`CustomOutputDataType`

Custom output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` on page 4-243 property to `Custom`. The default is `numerictype([],16,15)`.

Methods

freqz	Frequency response
fvtool	Open filter visualization tool
impz	Impulse response
phasez	Unwrapped phase response
reset	Reset states of biquad filter object
step	Filter input with biquad filter object

More "Analysis Methods for Filter System Objects" on page 3-2.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)

Common to All System Objects	
`release`	Allow System object property value changes

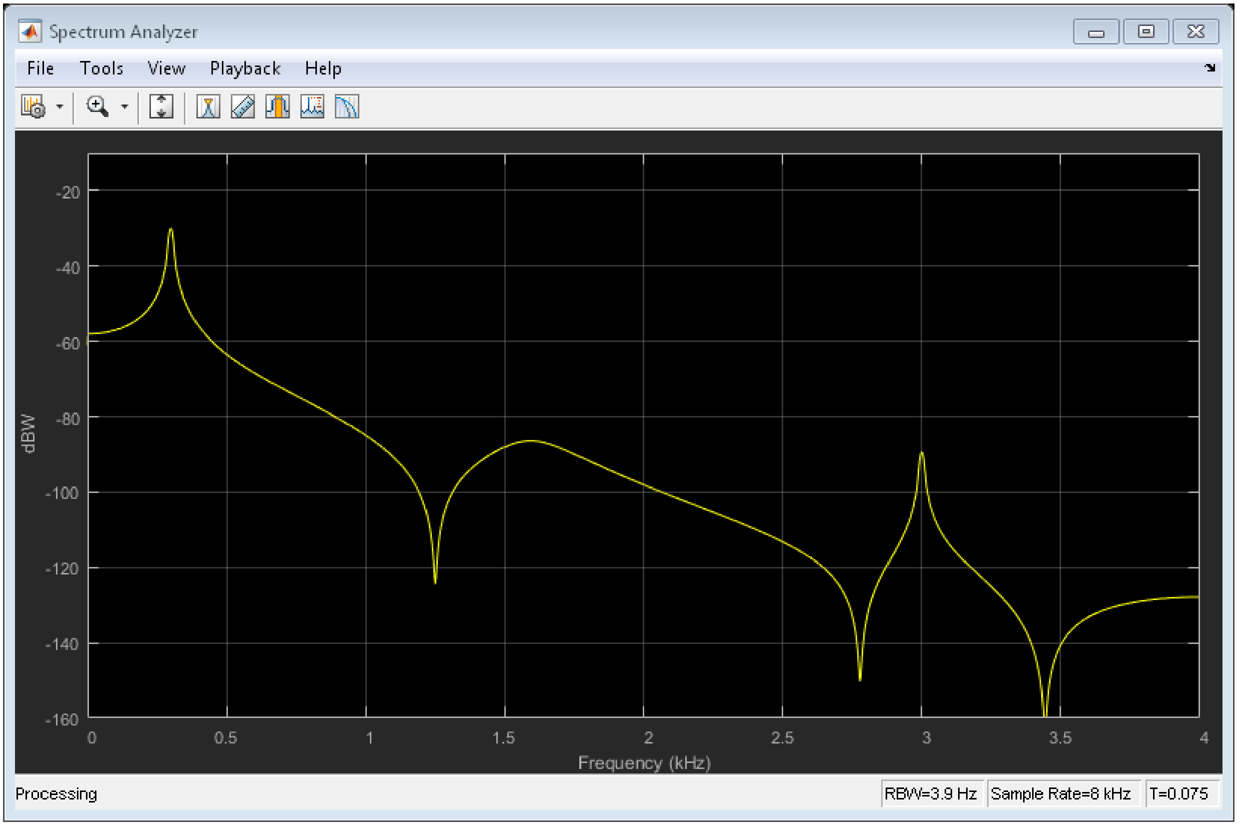
Examples

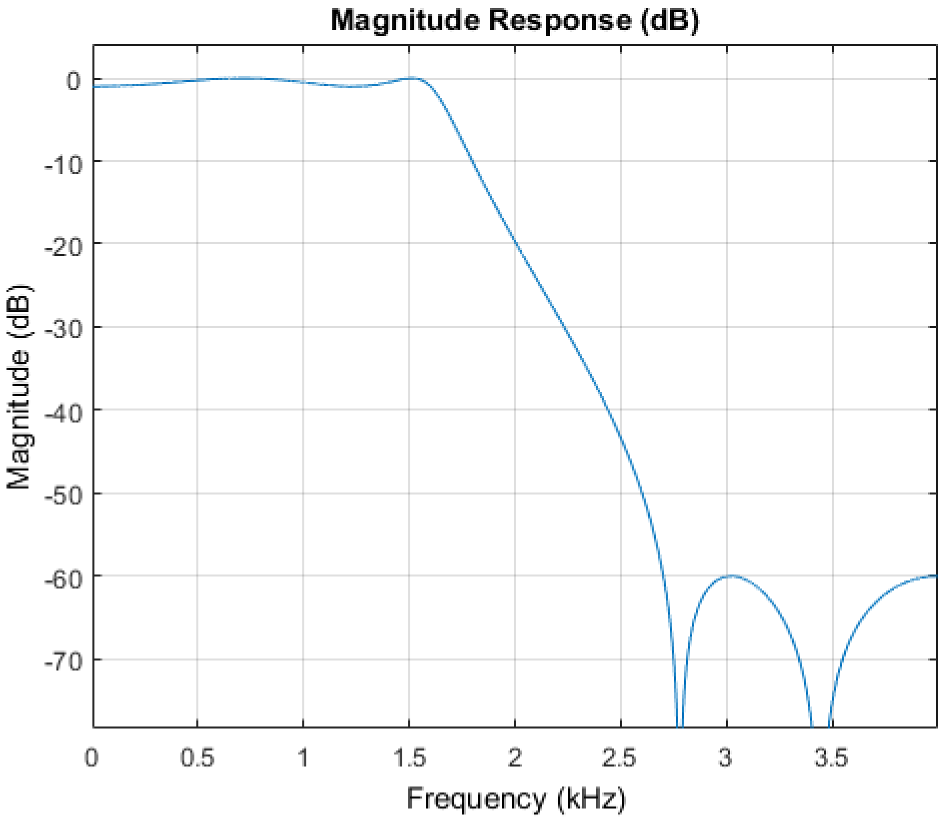
Filter a Signal Using Biquadratic Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Use a fourth order, lowpass biquadratic filter object with a normalized cutoff frequency of 0.4 to filter high frequencies from an input signal. Display the result as a power spectrum using the Spectrum Analyzer.

```
t = (0:1000)'/8e3;
xin = sin(2*pi*0.3e3*t)+sin(2*pi*3e3*t); % Input is 0.3 &amp;
                                        % 3kHz sinusoids
src = dsp.SignalSource(xin, 100);
sink = dsp.SignalSink;

[z,p,k] = ellip(4,1,60,.4);    % Set up the filter
[s,g] = zp2sos(z,p,k);
biquad = dsp.BiquadFilter('Structure','Direct form I', ...
   'SOSMatrix',s,'ScaleValues',g);

sa = dsp.SpectrumAnalyzer('SampleRate',8e3,...
    'PlotAsTwoSidedSpectrum',false,...
    'OverlapPercent', 80,'PowerUnits','dBW',...
    'YLimits', [-160 -10]);

while ~isDone(src)
     input = src();
     filteredOutput = biquad(input);
     sink(filteredOutput);
     sa(filteredOutput)
end

filteredResult = sink.Buffer;
fvtool(biquad,'Fs',8000)
```


Spectrum Analyzer
File Tools View Playback Help
-20
-40
-60
-80
-100
-120
-140
-160
dBW
0 0.5 1 1.5 2 2.5 3 3.5 4
Frequency (kHz)
Processing
RBW=3.9 Hz
Sample Rate=8 kHz
T=0.075

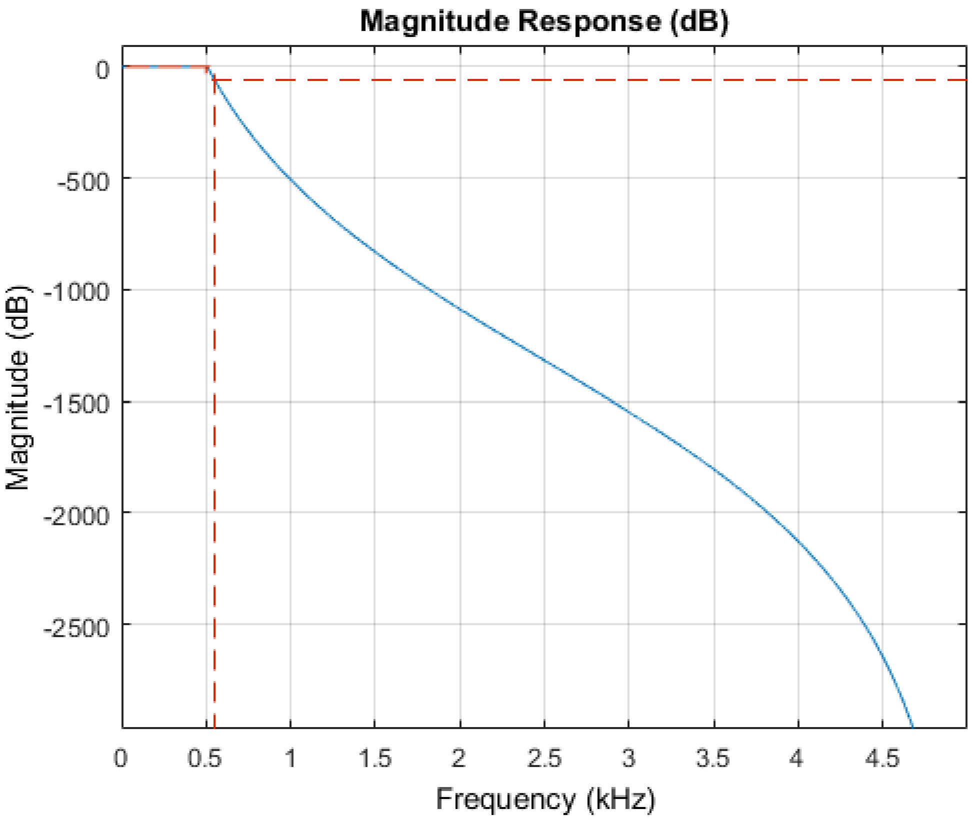


Design and apply a lowpass biquad filter System object™ using the `design` function.

```
lpSpec = fdesign.lowpass('Fp,Fst,Ap,Ast',500,550,0.5,60,10000);
lpfilter = design(lpSpec,'butter','systemobject',true)
fvtool(lpfilter);
```

```
lpfilter =

  dsp.BiquadFilter with properties:

                 Structure: 'Direct form II'
           SOSMatrixSource: 'Property'
                 SOSMatrix: [42×6 double]
```

```
              ScaleValues: [43×1 double]
        InitialConditions: 0
 OptimizeUnityScaleValues: true

Use get to show all properties
```



Algorithm

This object implements the algorithm, inputs, and outputs described on the `Biquad Filter` block reference page. The object properties correspond to the block parameters, except:

- **Coefficient source**
- **Action when the a0 values of the SOS matrix are not one** – the biquad filter object assumes the zero-th-order denominator coefficient equals 1 regardless of the specified value. The biquad filter object does not support the `Error` or `Warn` options found in the corresponding block.

Both this object and its corresponding block support variable-size input. This means that the step method can handle an input which is changing in size.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

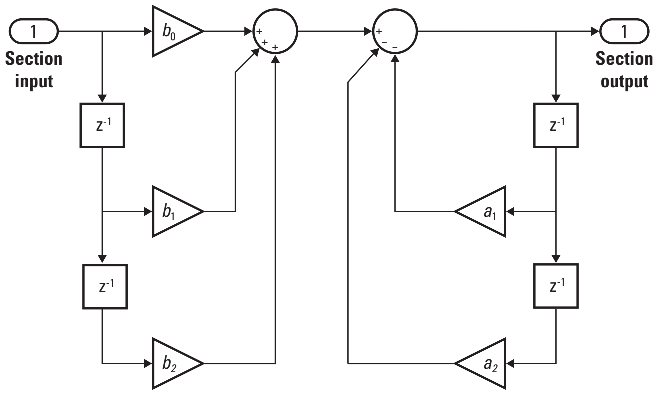
This object supports HDL code generation if you have the HDL Coder installed. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

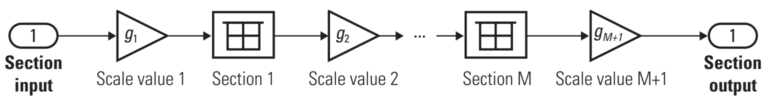
Fixed-Point Conversion

Design and simulate fixed-point algorithms using Fixed-Point Designer™.

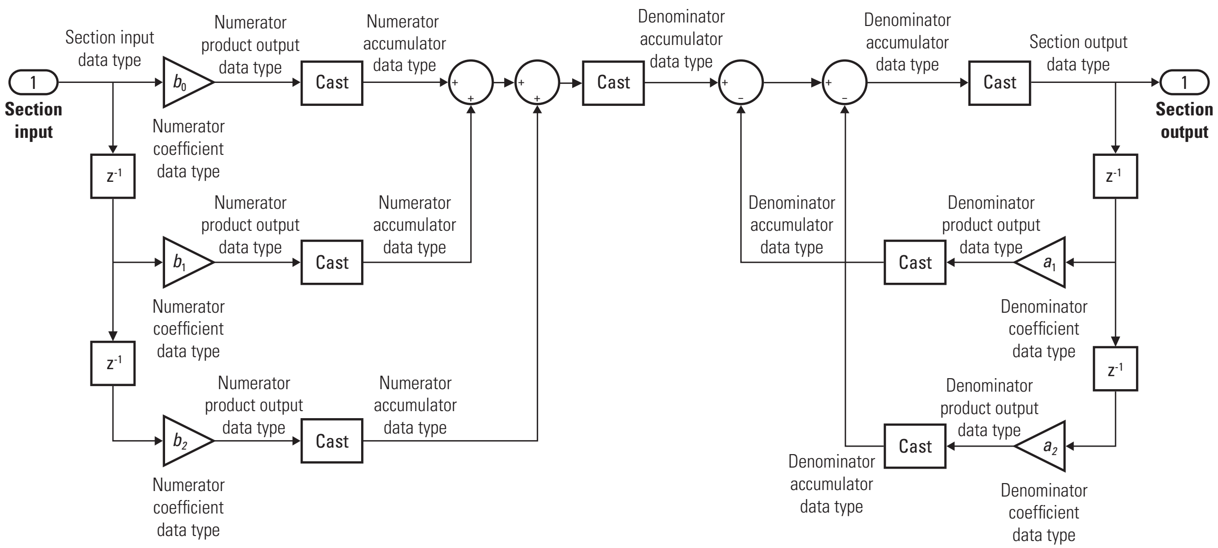
The following diagrams show the data types used in the `dsp.BiquadFilter` object when the input is fixed-point. For each filter structure the object supports, the data types shown in the diagrams can be set through the respective fixed-point properties of the object.

Direct Form I



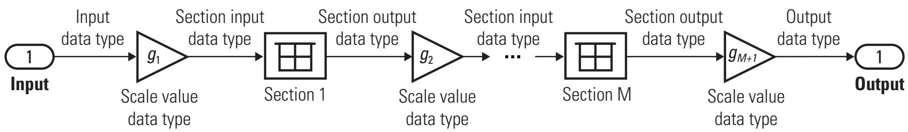


The following diagram shows the data types for one section of the filter for fixed-point signals.

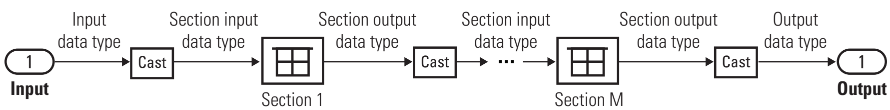


The following diagrams show the fixed-point data types between filter sections.

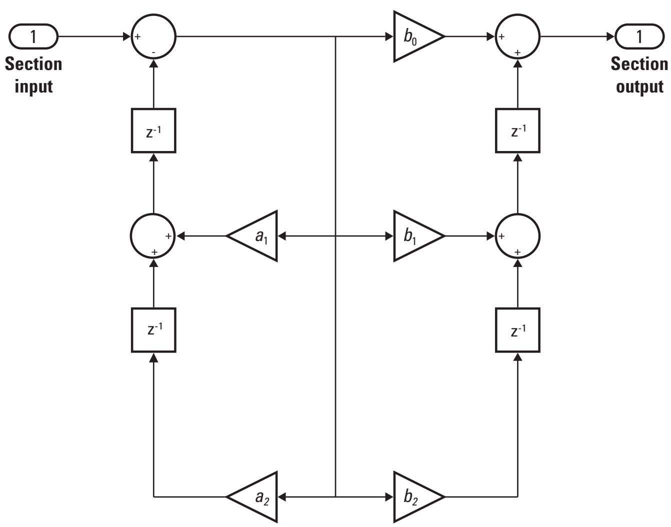
When the data is not optimized:

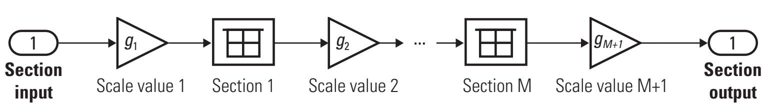


When you specify `OptimizeUnityScaleValues` to `true`, and scale values to 1:

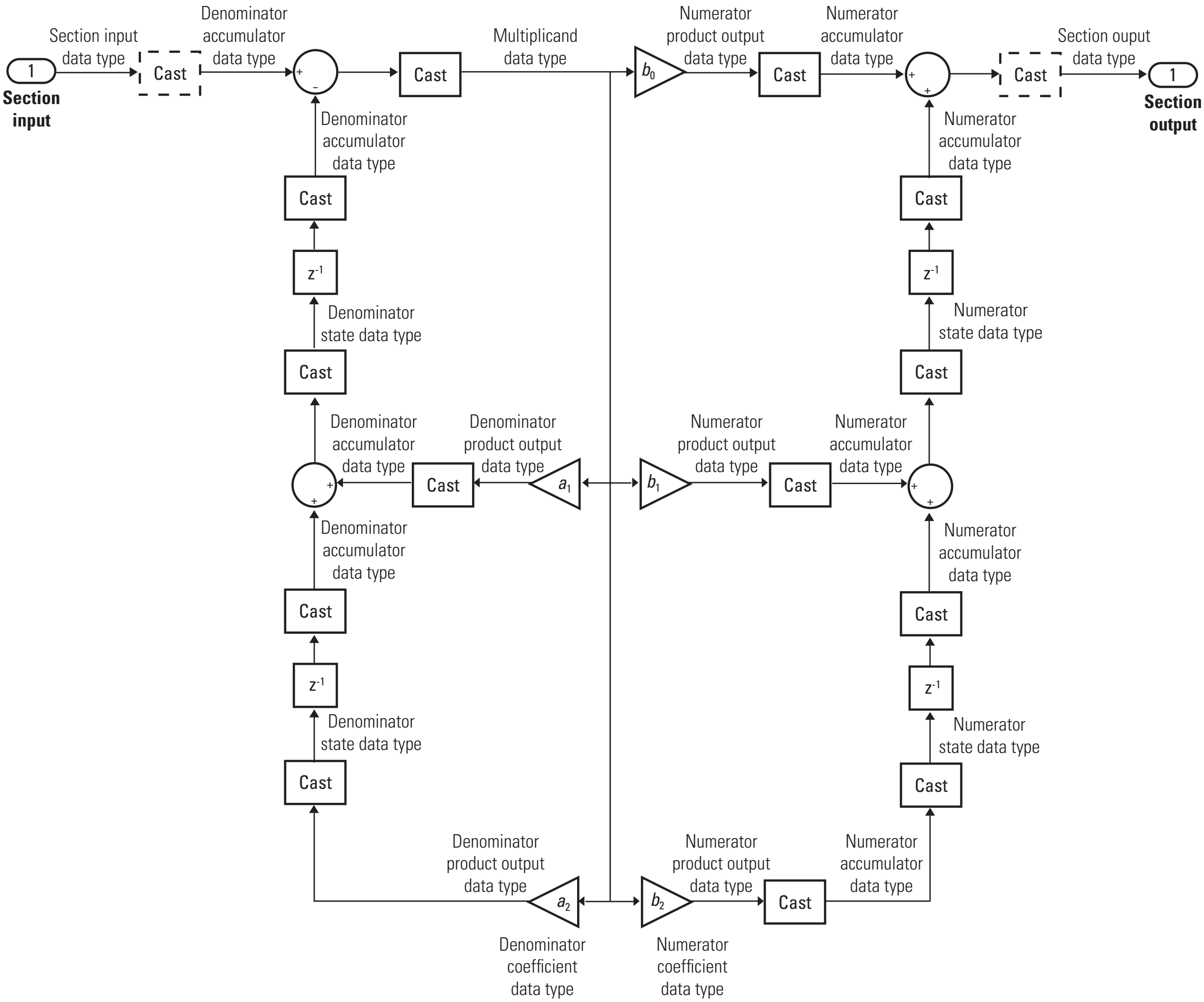


Direct Form I Transposed





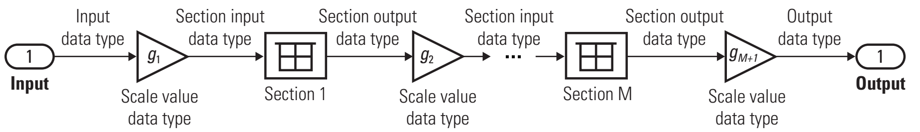
The following diagram shows the data types for one section of the filter for fixed-point signals.



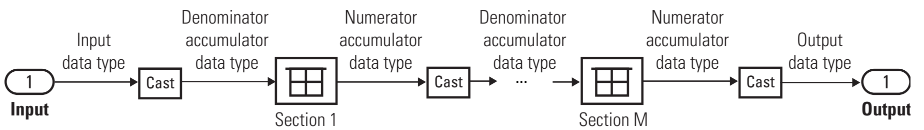
The dashed casts are omitted when you specify `OptimizeUnityScaleValues` to `true`, and scale values to 1.

The following diagrams show the fixed-point data types between filter sections.

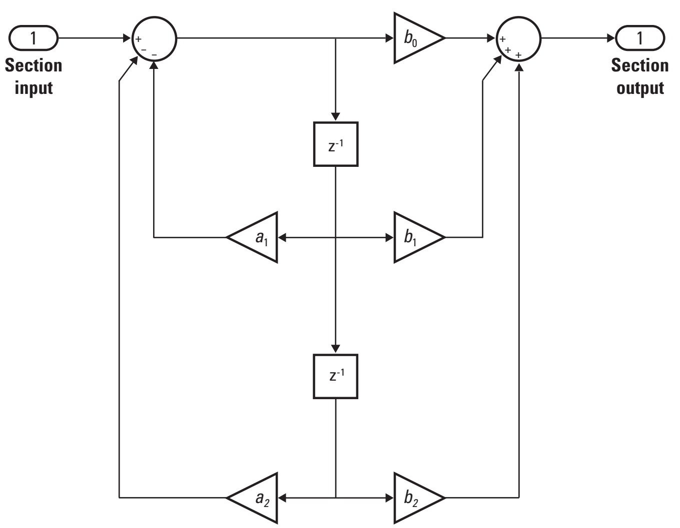
When the data is not optimized:

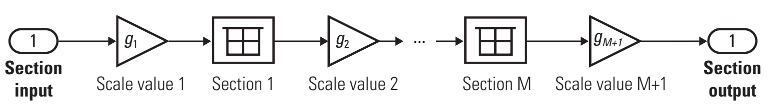


When you specify `OptimizeUnityScaleValues` to `true`, and scale values to 1:

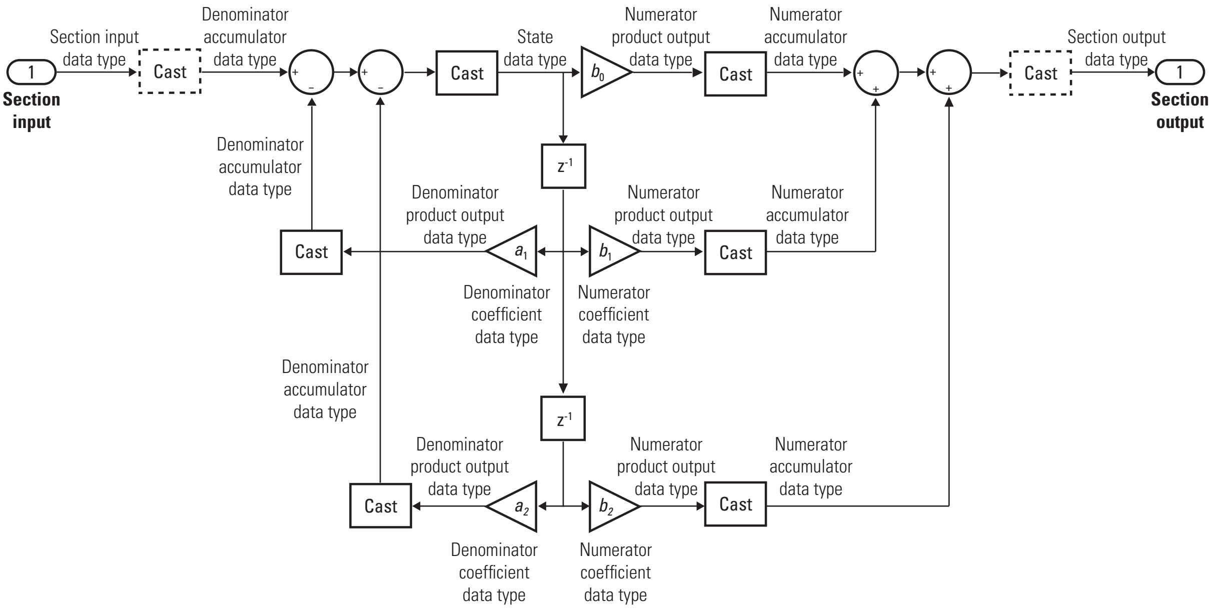


Direct Form II





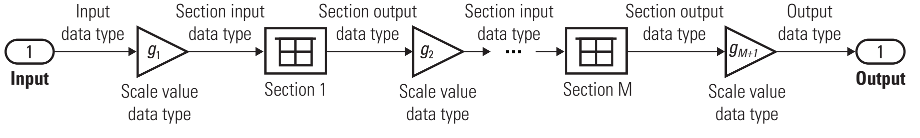
The following diagram shows the data types for one section of the filter for fixed-point signals.



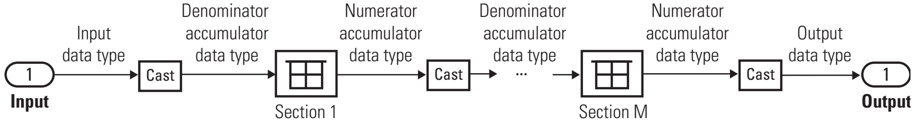
The dashed casts are omitted when you specify `OptimizeUnityScaleValues` to `true`, and scale values to 1.

The following diagrams show the fixed-point data types between filter sections.

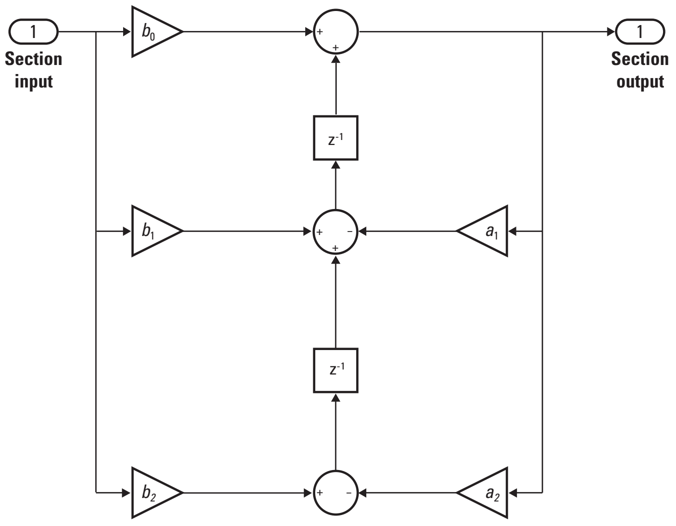
When the data is not optimized:

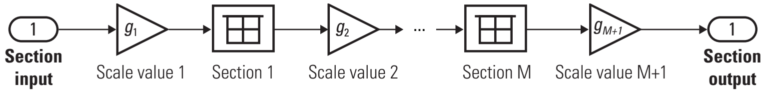


When you specify `OptimizeUnityScaleValues` to `true`, and scale values to 1:

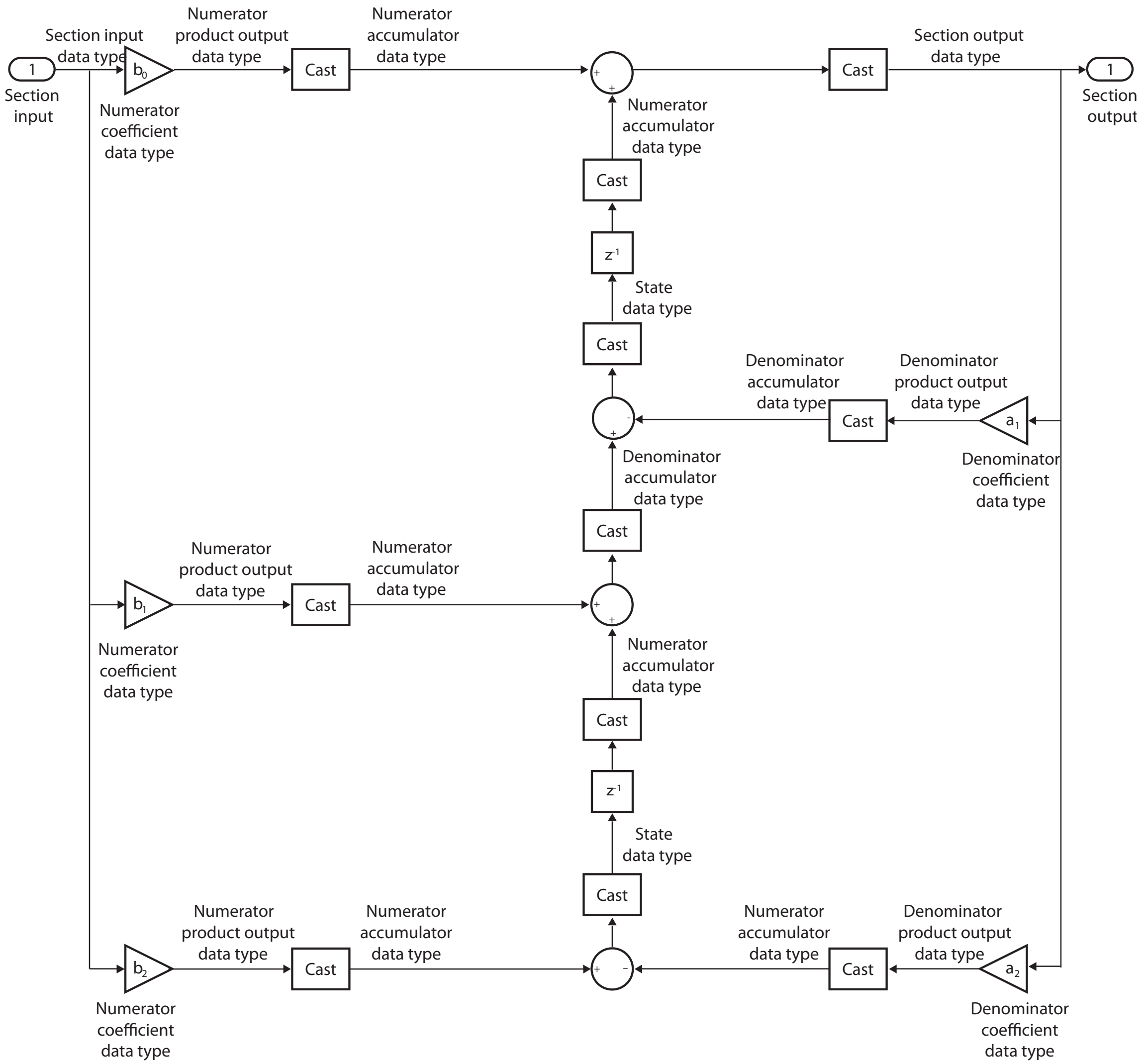


Direct Form II Transposed



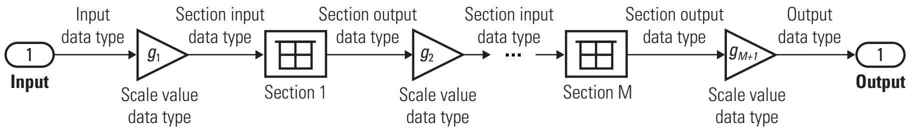


The following diagram shows the data types for one section of the filter for fixed-point signals.

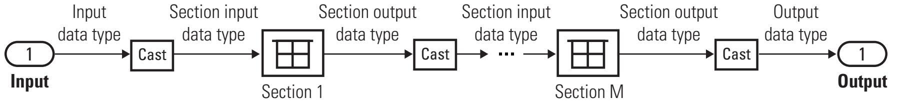


The following diagrams show the fixed-point data types between filter sections.

When the data is not optimized:



When you specify `OptimizeUnityScaleValues` to `true`, and scale values to 1:



See Also

See Also

System Objects

dsp.FIRFilter | dsp.IIRFilter

Blocks

`Biquad Filter`

Introduced in R2012a

freqz

System object: dsp.BiquadFilter
Package: dsp

Frequency response

Syntax

```
[h,w] = freqz(biquad)
[h,w] = freqz(biquad,n)
[h,w] = freqz(biquad,Name,Value)
freqz(biquad)
```

Description

`[h,w] = freqz(biquad)` returns the complex, 8192–element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample.

`[h,w] = freqz(biquad,n)` returns the complex, n-element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[h,w] = freqz(biquad,Name,Value)` returns the frequency response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`freqz(biquad)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the filter System object `biquad`.

fvtool

System object: dsp.BiquadFilter
Package: dsp

Open filter visualization tool

Syntax

```
fvtool(biquad)
fvtool(biquad,'Arithmetic',ARITH,...)
```

Description

`fvtool(biquad)` performs an analysis and computes the magnitude response of the filter System object `biquad`.

`fvtool(biquad,'Arithmetic',ARITH,...)` analyzes the filter System object `biquad`, based on the arithmetic specified in the ARITH input. ARITH can be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The 'Arithmetic' input is only relevant for the analysis of filter System objects.

impz

System object: dsp.BiquadFilter
Package: dsp

Impulse response

Syntax

```
[h,t] = impz(biquad)
[h,t] = impz(biquad,Name,Value)
impz(biquad)
```

Description

`[h,t] = impz(biquad)` returns the impulse response `h`, and the corresponding time points `t` at which the impulse response of`biquad` is computed.

`[h,t] = impz(biquad,Name,Value)` returns the impulse response `h`, and the corresponding time points`t` , with additional options specified by one or more Name, Value pair arguments.

`impz(biquad)` uses FVTool to plot the impulse response of the filter System object `biquad`.

Note: You can use `impz` for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

---

phasez

System object: dsp.BiquadFilter
Package: dsp

Unwrapped phase response

Syntax

```
[phi,w] = phasez(biquad)
[phi,w] = phasez(biquad,n)
[phi,w] = phasez(biquad,Name,Value)
phasez(biquad)
```

Description

`[phi,w] = phasez(biquad)` returns the 8192–element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample.

`[phi,w] = phasez(biquad,n)` returns the n-element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[phi,w] = phasez(biquad,Name,Value)` returns the phase response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`phasez(biquad)` uses FVTool to plot the phase response of the filter System object `biquad`.

reset

System object: dsp.BiquadFilter
Package: dsp

Reset states of biquad filter object

Syntax

```
reset(biquad)
```

Description

`reset(biquad)` resets the filter states of the biquad filter object,`biquad`, to the specified initial conditions. After the `step` method applies the biquad filter object to nonzero input data, the states may change. Invoking the `step` method again without first invoking the `reset` method may produce different outputs for an identical input.

Reset a Biquad Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
n =0:20;
x = cos(pi/4*n)+sin(pi/2*n);
biquad = dsp.BiquadFilter;
y = biquad(x');
```

Call the biquad filter without invoking reset.

```
y1 = biquad(x');
isequal(y,y1)
```

```
ans =

  logical
```

```
   0
```

Reset filter states to zero.

```
reset(biquad);
```

Call the biquad filter again.

```
y2 = biquad(x');
isequal(y,y2)
```

```
ans =

  logical

   1
```

step

System object: dsp.BiquadFilter
Package: dsp

Filter input with biquad filter object

Syntax

```
Y = step(biquad,X)
Y = step(biquad,X,NUM,DEN)
Y = step(biquad,X,NUM,DEN,G)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(biquad,X)` filters the real or complex input signal `X` , and outputs the filtered values, `Y`. The biquad filter object filters each channel of the input signal over successive calls to the `step` method.

`Y = step(biquad,X,NUM,DEN)` filters the input using `NUM` as the numerator coefficients, and `DEN` as the denominator coefficients of the biquad filter. `NUM` must be a 3-by-N numeric matrix and `DEN` must be a 2-by-N numeric matrix, where N is the number of biquad filter sections. The object assumes that the first denominator coefficient of each section is 1. This configuration applies when the `SOSMatrixSource` property is `Input port` and the `ScaleValuesInputPort` property is `false`.

`Y = step(biquad,X,NUM,DEN,G)` specifies the scale values, `G`, of the biquad filter. `G` must be a 1-by-(N+1) numeric vector, where `N` is the number of biquad filter sections. This configuration applies when the `SOSMatrixSource` property is `Input Port` and the `ScaleValuesInputPort` property is true.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.BlockLMSFilter System object

Package: dsp

Compute output, error, and weights using Block LMS adaptive algorithm

Description

The `BlockLMSFilter` object computes output, error, and weights using the Block LMS adaptive algorithm.

To compute the output, error, and weights:

1. Define and set up your adaptive FIR filter. See "Construction" on page 4-265.
2. Call step to compute the output, error, and weights according to the properties of `dsp.BlockLMSFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`blms = dsp.BlockLMSFilter` returns an adaptive FIR filter, `blms`, that filters the input signal and computes filter weights based on the Block Least Mean Squares (LMS) algorithm.

`blsm = dsp.BlockLMSFilter('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an adaptive FIR filter, `blms`, with each specified property set to the specified value.

`blms = dsp.BlockLMSFilter(length,blocksize,'`*`PropertyName`*`',...`
`PropertyValue``,...)` returns an adaptive FIR filter, `blms`, with the `Length` property set to `length`, the `BlockSize` property set to `blocksize`, and other specified properties set to the specified values.

Properties

Length

Length of FIR filter weights vector

Specify the length of the FIR filter weights vector as a positive integer scalar. The default is 32.

BlockSize

Number of samples acquired before weight adaptation

Specify the number of samples of the input signal to acquire before the object updates the filter weights. The input frame length must be an integer multiple of the block size. The default is 32.

StepSizeSource

Source of adaptation step size

Choose to specify the adaptation step size factor as Property or Input port. The default is Property.

StepSize

Adaptation step size

Specify the adaptation step size factor as a scalar, nonnegative numeric value. The default is 0.1. This property applies only when you set the StepSizeSource on page 4-266 property to 'Property'. This property is tunable.

LeakageFactor

Leakage factor used in Leaky LMS algorithm

Specify the leakage factor used in Leaky LMS algorithm as a scalar numeric value between 0 and 1, both inclusive. When the value is less than 1, the System object implements a leaky LMS algorithm. The default is 1, providing no leakage in the adapting algorithm. This property is tunable.

InitialWeights

Initial values of filter weights

Specify the initial values of the filter weights as a scalar or a vector of length equal to the `Length` on page 4-266 property value. The default is `0`.

AdaptInputPort

Additional input to enable adaptation of filter weights.

Specify when the object should adapt the filter weights. By default, the value of this property is `false`, and the filter continuously updates the filter weights. When this property is set to `true`, an adaptation control input is provided to the `step` method. If the value of this input is nonzero, the filter continuously updates the filter weights. If the input is zero, the filter weights remain at their current value.

WeightsResetInputPort

Additional input to enable weights reset

Specify whether the FIR filter can reset the filter weights. By default, the value of this property is `false`, and the object does not reset the weights. When this property is set to `true`, a reset control input is provided to the `step` method, and the `WeightsResetCondition` on page 4-267 property applies. The object resets the filter weights based on the values of the `WeightsResetCondition` on page 4-267 property and the reset input to the `step` method.

WeightsResetCondition

Condition that triggers the resetting of filter weights

Specify the event to reset the filter weights as one of `Rising edge`, `Falling edge`, `Either edge`, or `Non-zero`. The object resets the filter weights based on the values of this property and the reset input to the `step` method. This property applies only when you set the `WeightsResetInputPort` on page 4-267 property to `true`. The default is `Non-zero`.

WeightsOutputPort

Output filter weights

Set this property to `true` to output the adapted filter weights. The default is `true`.

Methods

msesim	Mean-square error for Block LMS filter
reset	Reset internal states of adaptive FIR filter object
step	Filter inputs using Block LMS algorithm

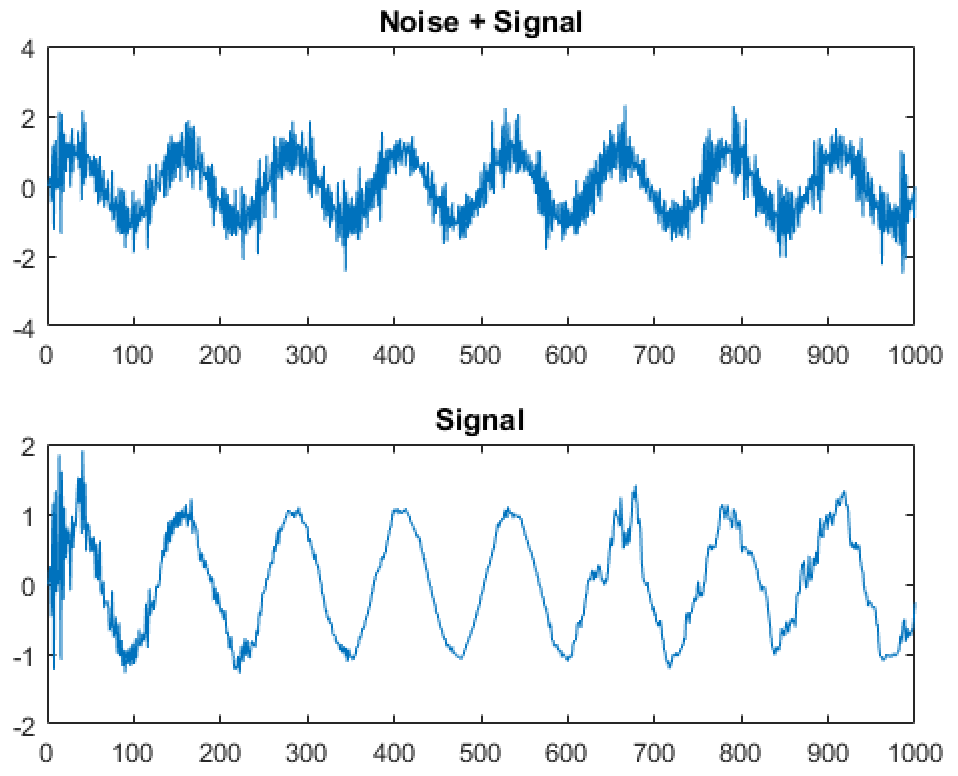
Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Remove Noise Using Block LMS Adaptive Algorithm

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
blms = dsp.BlockLMSFilter(10,5);
blms.StepSize = 0.01;
blms.WeightsOutputPort = false;
filt = dsp.FIRFilter;
filt.Numerator = fir1(10,[.5, .75]);
x = randn(1000,1); % Noise
d = filt(x) + sin(0:.05:49.95)'; % Noise + Signal
[y, err] = blms(x, d);
subplot(2,1,1);
plot(d);
title('Noise + Signal');
subplot(2,1,2);
plot(err);
title('Signal');
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Block LMS Filter` block reference page. The object properties correspond to the block parameters.

See Also

dsp.FIRFilter | dsp.LMSFilter

Introduced in R2012a

msesim

System object: dsp.BlockLMSFilter
Package: dsp

Mean-square error for Block LMS filter

Syntax

```
MSE = msesim(sysObj,X,D)
[MSE,MEANW,W,TRACEK] = msesim(sysObj,X,D)
[...] = msesim(sysObj,X,D,M)
```

Description

`MSE = msesim(sysObj,X,D)` returns a sequence of mean-square errors. This column vector contains estimates of the mean-square error of the adaptive filter at each time instant. The length of `MSE` is equal to `SIZE(X,1)`. The columns of the matrix `X` contain individual input signal sequences, and the columns of the matrix `D` contain corresponding desired response signal sequences.

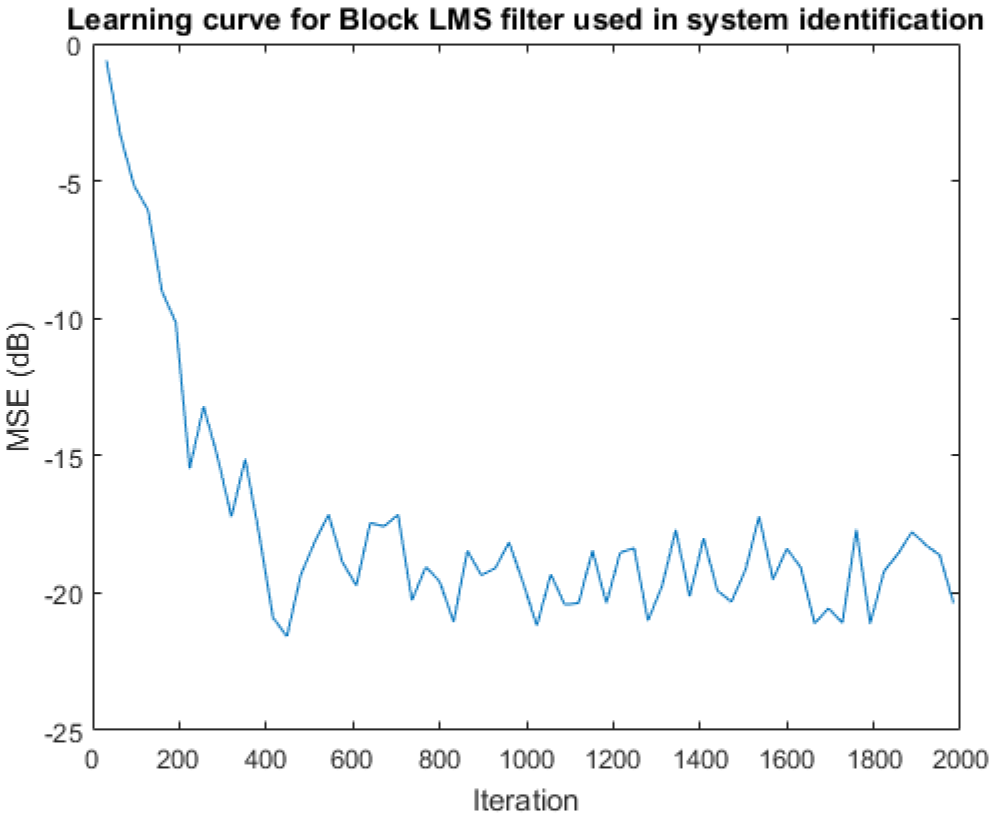
`[MSE,MEANW,W,TRACEK] = msesim(sysObj,X,D)` calculates three parameters corresponding to the simulated behavior of the adaptive filter defined by `sysObj`. `MEANW` is a sequence of coefficient vector means. The columns of this matrix contain estimates of the mean values of the adaptive filter coefficients at each time instant. The dimensions of `MEANW` are `(SIZE(X,1))` by `(sysObj.length)`. `W` is an estimate of the final values of the adaptive filter coefficients for the algorithm corresponding to `sysObj`. `TRACEK` is a sequence of total coefficient error powers. This column vector contains estimates of the total coefficient error power of the adaptive filter at each time instant. The length of `TRACEK` is equal to `SIZE(X,1)`.

`[...] = msesim(sysObj,X,D,M)` specifies an optional decimation factor for computing `MSE`, `MEANW`, and `TRACEK`. If `M > 1`, every M^{th} predicted value of each of these sequences is saved. If omitted, the value of `M` is the default, which is 1.

System Identification of an FIR Filter Using Block LMS Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
fir = fir1(31,0.5);
firFilter = dsp.FIRFilter('Numerator',fir); % FIR system to be identified
iirFilter = dsp.IIRFilter('Numerator',sqrt(0.75),...
        'Denominator',[1 -0.5]);
x = iirFilter(sign(randn(2000,25)));
n = 0.1*randn(size(x));            % Observation noise signal
d = firFilter(x)+n;                  % Desired signal
l = 32;                            % Filter length
mu = 0.008;                        % Block LMS Step size.
m  = 32;                           % Decimation factor for analysis
                                   % and simulation results
fir = dsp.BlockLMSFilter(l,'StepSize',mu);
[simmse,meanWsim,Wsim,traceKsim] = msesim(fir,x,d,m);
plot(m*(1:length(simmse)),10*log10(simmse));
xlabel('Iteration'); ylabel('MSE (dB)');
title('Learning curve for Block LMS filter used in system identification')
```


Learning curve for Block LMS filter used in system identification
MSE (dB)
0
-5
-10
-15
-20
-25
0
200
400
600
800
1000
1200
1400
1600
1800
2000
Iteration

reset

System object: dsp.BlockLMSFilter
Package: dsp

Reset internal states of adaptive FIR filter object

Syntax

```
reset(blms)
```

Description

`reset(blms)` sets the internal states of the `BlockLMSFilter` object `blms` to their initial values.

step

System object: dsp.BlockLMSFilter
Package: dsp

Filter inputs using Block LMS algorithm

Syntax

```
[Y,ERR,WTS] = step(blms,X,D)
[Y,ERR] = step(blms,X,D)
[...] = step(blms,X,D,MU)
[...] = step(blms,X,D,A)
[...] = step(blms,X,D,R)
[Y,ERR,WTS] = step(blms,X,D,MU,A,R)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[Y,ERR,WTS] = step(blms,X,D)` filters the input `X`, using `D` as the desired signal, and returns the filtered output in `Y`. The filter error is in `ERR`, and the estimated filter weights is in `WTS`. The filter weights update once for every block of data that the object processes.

`[Y,ERR] = step(blms,X,D)` returns only the filtered output in `Y` and the filter error in `ERR`, when the `WeightsOutputPort` property is `false`.

`[...] = step(blms,X,D,MU)` uses `MU` as the step size, when you set the `StepSizeSource` property to `Input port`.

`[...] = step(blms,X,D,A)` uses `A` as the adaptation control, when you set the `AdaptInputPort` property to `true`. When `A` is nonzero, the filter continuously updates the filter weights. When `A` is zero, the filter weights remain constant.

`[...] = step(blms,X,D,R)` uses `R` as a reset signal, when you set the `WeightsResetInputPort` property to `true`. Use the `WeightsResetCondition` property to set the reset trigger condition. If a reset event occurs, the filter resets the filter weights to their initial values.

`[Y,ERR,WTS] = step(blms,X,D,MU,A,R)` filters input `X`, using `D` as the desired signal, `MU` as the step size, `A` as the adaptation control, and `R` as the reset signal. The objects returns the filtered output in `Y`, the filter error in `ERR`, and the adapted filter weights in `WTS`. Set the properties appropriately to provide all possible inputs.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.Buffer System object

Package: dsp

Buffer input signal

Description

The `Buffer` object buffers an input signal. The number of samples per channel in the input must equal the difference between the output buffer size and buffer overlap (i.e., `Length` - `OverlapLength`).

To buffer an input signal:

1. Define and set up your buffer System object. See "Construction" on page 4-276.
2. Call step to buffer the input according to the properties of `dsp.Buffer`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`buff = dsp.Buffer` returns a buffer System object, `buff`, used to buffer input signals with overlap.

`buff = dsp.Buffer('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a buffer object, `buff`, with each specified property set to the specified value.

`buff = dsp.Buffer(LEN,OVRLAP,ICS,'`*`PropertyName`*`',...`
`PropertyValue``,...)` returns a buffer object, `buff`, with `Length` on page 4-277 property set to `LEN`, `OverlapLength` on page 4- property set to `OVRLAP`, `InitialConditions` on page 4-277 property set to `ICS` and other specified properties set to the specified values.

Properties

`Length`

Number of samples to buffer

Specify the number of consecutive samples from each input channel to buffer. You can set this property to any scalar integer greater than `0` of MATLAB built-in numeric data type. The default is `64`.

`OverlapLength`

Amount of overlap between outputs

Specify the number of samples by which consecutive output frames overlap. You can set this property to any scalar integer greater than or equal to `0` of MATLAB built-in numeric data type. The default is `0`.

`InitialConditions`

Initial output

Specify the value of the object's initial output for cases of nonzero latency as a scalar, vector, or matrix. The default is `0`.

Methods

reset	Reset the internal states of a System object
step	Buffer input signal based on past values

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Create a Buffer

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create a buffer of 256 samples with 128 sample overlap.

```
reader = dsp.SignalSource(randn(1024,1),128);
buff = dsp.Buffer(256,128);

for i = 1:8
    y = buff(reader());
end
```

y is of length 256 with 128 samples from previous input.

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Buffer` block reference page. The object properties correspond to the block properties, except as noted.

See Also

dsp.Delay | dsp.DelayLine

Introduced in R2012a

reset

System object: dsp.Buffer
Package: dsp

Reset the internal states of a System object

Syntax

```
reset(buff)
```

Description

`reset(buff)` resets the internal states of buffer to their initial values.

For many System objects, this method is a no-op. Objects that have internal states will describe in their help what the reset method does for that object.

The reset method is always a no-op for unlocked System objects, as the states may not be allocated when the object is not locked.

step

System object: dsp.Buffer
Package: dsp

Buffer input signal based on past values

Syntax

`Y = step(buff,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(buff,X)` creates output `Y` based on current input and stored past values of `X` . Output length equals the `Length` property.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.BurgAREstimator System object

Package: dsp

Estimate of autoregressive (AR) model parameters using Burg method

Description

The `BurgAREstimator` object computes the estimate of the autoregressive (AR) model parameters using the Burg method.

To compute the estimate of the AR model parameters:

1. Define and set up your System object. See "Construction" on page 4-281.
2. Call step to compute the estimate according to the properties of `dsp.BurgAREstimator`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`burgarest = dsp.BurgAREstimator` returns a Burg `BurgAREstimator` System object, `burgarest`, that performs parametric AR estimation using the Burg maximum entropy method.

`burgarest = dsp.BurgAREstimator('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a Burg AR estimator object, `burgarest`, with each specified property set to the specified value.

Properties

`AOutputPort`

Enable output of polynomial coefficients

Set this property to `true` to output the polynomial coefficients, `A`, of the AR model the object computes. The default is `true`. Either the `AOutputPort` property, the `KOutputPort on page 4-282` property, or both must be `true`.

`KOutputPort`

Enable output of reflection coefficients

Set this property to `true` to output the reflection coefficients, `K`, for the AR model that the object computes. The default is `false`. Either the `AOutputPort on page 4-282` property, the `KOutputPort` property, or both must be `true`.

`EstimationOrderSource`

Source of estimation order

Specify how to determine estimator order as `Auto` or `Property`. When you set this property to `Auto`, the object assumes the estimation order is one less than the length of the input vector. When you set this property to `Property`, the value in EstimationOrder on page 4-282 is used. The default is `Auto`.

`EstimationOrder`

Order of AR model

Set the AR model estimation order to a real positive integer. This property applies when you set the `EstimationOrderSource on page 4-282` to `Property`. The default is `4`.

Methods

step	Normalized estimate of AR model parameter

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

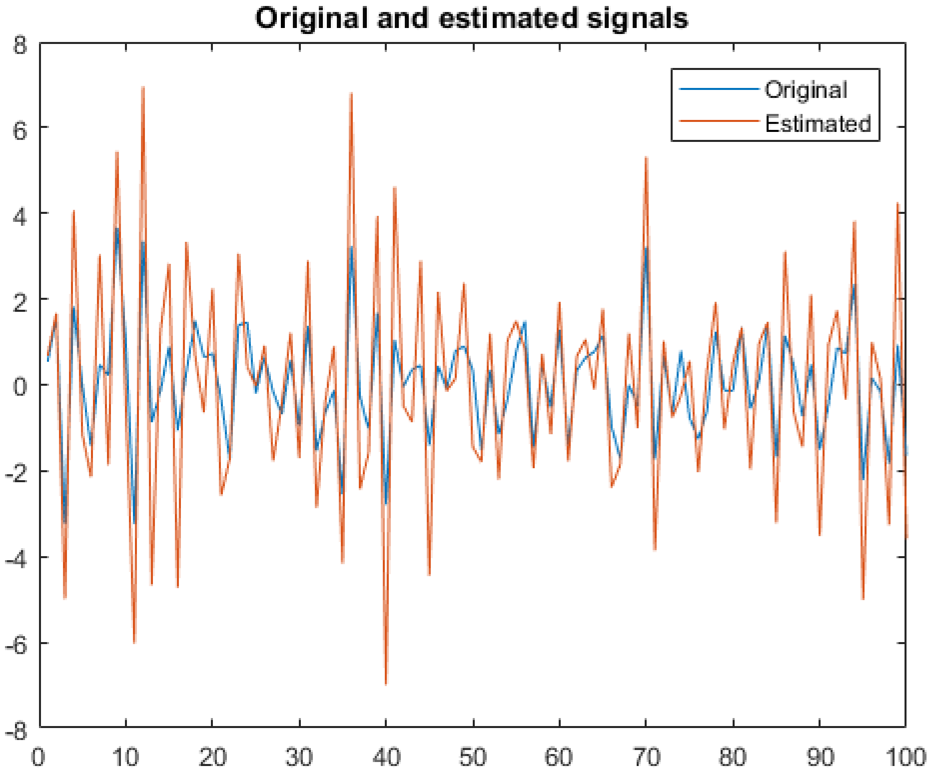
Examples

Estimate the parameters of an AR model

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Use the `dsp.BurgAREstimator` System object™ to estimate the parameters of an AR model.

```
rng default; % Use default random number generator and seed
noise = randn(100,1); % Normalized white Gaussian noise
x = filter(1,[1 1/2 1/3 1/4 1/5],noise);
burgarest = dsp.BurgAREstimator(...
    'EstimationOrderSource', 'Property', ...
    'EstimationOrder', 4);
[a, g] = burgarest(x);
x_est = filter(g, a, x);
plot(1:100,[x x_est]);
title('Original and estimated signals');
legend('Original', 'Estimated');
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Burg AR Estimator` block reference page. The object properties correspond to the block parameters, except:

Output(s) block parameter corresponds to the `AOutputPort` and the `KOutputPort` object properties.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.LevinsonSolver

Introduced in R2012a

step

System object: dsp.BurgAREstimator
Package: dsp

Normalized estimate of AR model parameter

Syntax

```
[A,G] = step(burgarest,X)
[K,G] = step(burgarest,X)
[A,K,G] = step(burgarest,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[A,G] = step(burgarest,X)` computes the normalized estimate of the `AR` model parameters to fit the input, `X`, in the least square sense. The input `X` must be a column vector. Output `A` is a column vector that contains the normalized estimate of the `AR` model polynomial coefficients in descending powers of z. The scalar `G` is the `AR` model gain.

`[K,G] = step(burgarest,X)` returns `K`, a column vector containing the `AR` model reflection coefficients when you set the `KOutputPort` property to `true` and the `AOutputPort` property to `false`.

`[A,K,G] = step(burgarest,X)` returns the `AR` model polynomial coefficients `A`, reflection coefficients `K`, and the scalar gain `G` when the `AOutputPort` and `KOutputPort` properties are both `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.BurgSpectrumEstimator System object

Package: dsp

Parametric spectral estimate using Burg method

Description

The `BurgSpectrumEstimator` object computes a parametric spectral estimate of the input using the Burg method. The object fits an autoregressive (AR) model to the signal by minimizing the forward and backward prediction errors (via least-squares). The AR parameters are constrained to satisfy the Levinson-Durbin recursion.

To compute the parametric spectral estimate of the input:

1. Define and set up your System object. See “Construction” on page 4-288.
2. Call step to compute the estimate according to the properties of `dsp.BurgSpectrumEstimator`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`burgspecest = dsp.BurgSpectrumEstimator` returns an object, `burgspecest`, that estimates the power spectral density (PSD) of the input frame using the Burg method.

`burgspecest = dsp.BurgSpectrumEstimator('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a spectrum estimator, `burgspecest`, with each specified property set to the specified value.

Properties

`EstimationOrderSource`

Source of estimation order

Specify the source of the estimation order as `Auto` or `Property`. If you set this property to `Auto`, the object assumes the estimation order is one less than the length of the input vector. The default value is `Property`.

`EstimationOrder`

Order of AR model

Specify the order of AR model as a real positive integer. This property applies only when you set the `EstimationOrderSource` property to `Property`. The default value is `6`.

`FFTLengthSource`

Source of FFT length

Specify the source of the FFT length as `Auto` or `Property`. When you set this property to `Auto`, the objects assumes the FFT length is one more than the estimation order. When you set this property to `Property`, the `FFTLength` on page 4-289 property value must be an integer power of two.

`FFTLength`

FFT length as power-of-two integer value

Specify the FFT length as a power-of-two numeric scalar. This property applies when you set the `FFTLengthSource` on page 4-289 property to `Property`. The default value is `256`.

`SampleRate`

Sample rate of input time series

Specify the sampling rate of the original input time series as a positive numeric scalar in hertz. The default value is `1`.

Methods

step	Estimate of power spectral density

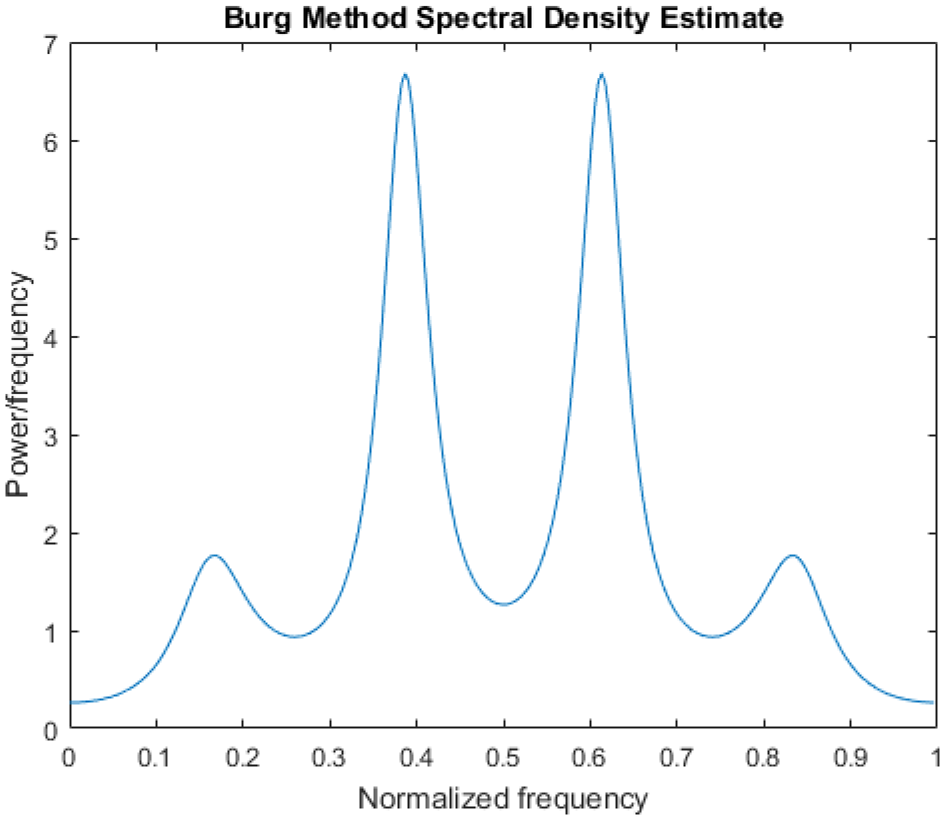
Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Estimate PSD Using the Burg Method

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
x = randn(100,1);
burgspecest = dsp.BurgSpectrumEstimator('EstimationOrder', 4);
y = filter(1,[1 1/2 1/3 1/4 1/5],x); % Fourth order AR filter
p = burgspecest(y); % Uses default FFT length of 256

plot((0:255)/256, p);
title('Burg Method Spectral Density Estimate');
xlabel('Normalized frequency'); ylabel('Power/frequency');
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Burg Method` block reference page. The object properties correspond to the block properties.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- When the FFT length is not a power of two, the executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- When the FFT length is a power of two, you can generate standalone C and C++ code from this System object.

See Also

See Also

System Objects

dsp.BurgAREstimator | dsp.LevinsonSolver

Introduced in R2012a

step

System object: dsp.BurgSpectrumEstimator
Package: dsp

Estimate of power spectral density

Syntax

`Y = step(burgspecest,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(burgspecest,X)` outputs `Y`, a spectral estimate of input `X`, using the Burg method.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.CepstralToLPC System object

Package: dsp

Convert cepstral coefficients to linear prediction coefficients

Description

The `CepstralToLPC` object converts cepstral coefficients to linear prediction coefficients (LPC).

To convert cepstral coefficients to LPC:

1. Define and set up your System object. See "Construction" on page 4-294.
2. Call step to convert the coefficients according to the properties of `dsp.CepstralToLPC`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`cc2lpc = dsp.CepstralToLPC` returns a System object, `cc2lpc`, that converts the cepstral coefficients(CCs) to linear prediction coefficients (LPCs).

`cc2lpc = dsp.CepstralToLPC('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a `Cepstral to LPC` object, `cc2lpc`, with each specified property set to the specified value.

Properties

`PredictionErrorOutputPort`

Enable prediction error power output

Set this property to `true` to output the prediction error power. The prediction error power is the power of the error output of an FIR analysis filter represented by the LPCs for a given input signal. The default is `false`.

Methods

step	LPC coefficients from column of cepstral coefficients

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Convert Cepstral To LPC Coefficients

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Convert cepstral coefficients to linear prediction coefficients.

```
cc = [0 0.9920 0.4919 0.3252 0.2418 , ...
    0.1917 0.1583 0.1344 0.1165 0.0956]';
cc2lpc = dsp.CepstralToLPC;
a = cc2lpc(cc)
```

```
a =

    1.0000
   -0.9920
    0.0001
```

```
0.0001
0.0001
0.0001
0.0001
0.0001
0.0001
0.0070
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `LPC to/from Cepstral Coefficients` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.LSFToLPC | dsp.LPCToCepstral | dsp.RCToLPC

Introduced in R2012a

step

System object: dsp.CepstralToLPC
Package: dsp

LPC coefficients from column of cepstral coefficients

Syntax

```
A = step(cc2lpc,CC)
[A,P]=step(cc2lpc,CC)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`A = step(cc2lpc,CC)` computes the linear prediction coefficients (LPC) coefficients, `A` , from the columns of cepstral coefficients, `CC`.

`[A,P]=step(cc2lpc,CC)` converts the columns of the cepstral coefficients `CC` to the LPCs and returns the prediction error power `P` when the `PredictionErrorOutputPort` property is `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.Channelizer System object

Package: dsp

Polyphase FFT analysis filter bank

Description

The `dsp.Channelizer` System object separates a broadband input signal into multiple narrow subbands using an FFT-based analysis filter bank. The filter bank uses a prototype lowpass filter and is implemented using a polyphase structure. You can specify the filter coefficients directly or through design parameters.

The number of rows in the input signal, `x`, must be a multiple of the number of frequency bands of the filter bank. Each column of `x` corresponds to a separate channel. If M is the number of frequency bands, and `x` is an L-by-1 matrix, then the output signal, `y`, has dimensions L/M-by-M. Each narrowband signal forms a column in the `y`. If `x` has more than one channel, that is, it has dimensions L-by-N, then `y` has dimensions L/M-by-M-by-N. The input `x` can be complex and supports single and double data types.

This object also accepts variable-size inputs. That is, once the object is locked, you can change the size of each input channel. The number of channels cannot change. This object supports C and C++ code generation.

To separate a broadband signal into multiple narrow subbands:

1 Create a `dsp.Channelizer` object and set the properties of the object.
2 Call step to do the signal analysis.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`channelizer = dsp.Channelizer` creates a channelizer object, using the default properties.

`channelizer = dsp.Channelizer(num)` sets the `NumFrequencyBands` property to `num`.

`channelizer = dsp.Channelizer(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

Example:

```
channelizer = dsp.Channelizer('NumTapsPerBand',20,'StopbandAttenuation',140);
```

Properties

`NumFrequencyBands` — Number of frequency bands

`8` (default) | positive integer greater than 1

Number of frequency bands into which the object separates the input broadband signal, specified as a positive integer greater than 1. This property indicates the FFT length and the decimation factor used by the algorithm.

`Specification` — Filter design parameters or coefficients

`'Number of taps per band and stopband attenuation'` (default) | `'Coefficients'`

Filter design parameters or filter coefficients, specified as one of these options:

- `'Number of taps per band and stopband attenuation'` — Specify the filter design parameters through the `NumTapsPerBand` and `StopbandAttenuation` properties.
- `'Coefficients'` — Specify the filter coefficients directly using `LowpassCoefficients`.

`NumTapsPerBand` — Number of filter coefficients per frequency band

`12` (default) | positive integer

Number of filter coefficients each polyphase branch uses, specified as a positive integer. The number of polyphase branches matches the number of frequency bands. The total number of filter coefficients for the prototype lowpass filter is given by `NumFrequencyBands×NumTapsPerBand`. For a given stopband attenuation, increasing the number of taps per band narrows the transition width of the filter. As a result, there is more usable bandwidth for each frequency band at the expense of increased computation.

This property applies when you set `Specification` to `'Number of taps per band and stopband attenuation'`.

`StopbandAttenuation` — Stopband attenuation
`80` (default) | positive real scalar

Stopband attenuation of the lowpass filter, specified as a positive real scalar in dB. This value controls the maximum amount of aliasing from one frequency band to the next. Larger is the stopband attenuation, smaller is the passband ripple.

This property applies when you set `Specification` to `'Number of taps per band and stopband attenuation'`.

`LowpassCoefficients` — Coefficients of the prototype lowpass filter
`[1×49 double]` (default) | row vector

Coefficients of the prototype lowpass filter, specified as a row vector. There must be at least one coefficient per frequency band. If the length of the lowpass filter is less than the number of frequency bands, the object zero-pads the coefficients.

This property applies when you set `Specification` to `'Coefficients'`.

This property is tunable. You can change its value even after the object is locked.

Methods

coeffs	Coefficients of prototype lowpass filter
polyphase	Polyphase matrix
reset	Reset internal states of System object
step	Separate broadband signal into narrow bands
tf	Transfer function

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object

Common to All System Objects	
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Quadrature Mirror Filter Bank

The quadrature mirror filter bank (QMF) contains an analysis filter bank section and a synthesis filter bank section. `dsp.Channelizer` implements the analysis filter bank. `dsp.ChannelSynthesizer` implements the synthesis filter bank using the efficient polyphase implementation that is based on a prototype lowpass filter.

Initialization

Initialize the `dsp.Channelizer` and `dsp.ChannelSynthesizer` System objects. Each object is set up with 8 frequency bands, 8 polyphase branches in each filter, 12 coefficients per polyphase branch, and a stopband attenuation of 140 dB. Use a sine wave with multiple frequencies as the input signal. View the input spectrum and the output spectrum using a spectrum analyzer.

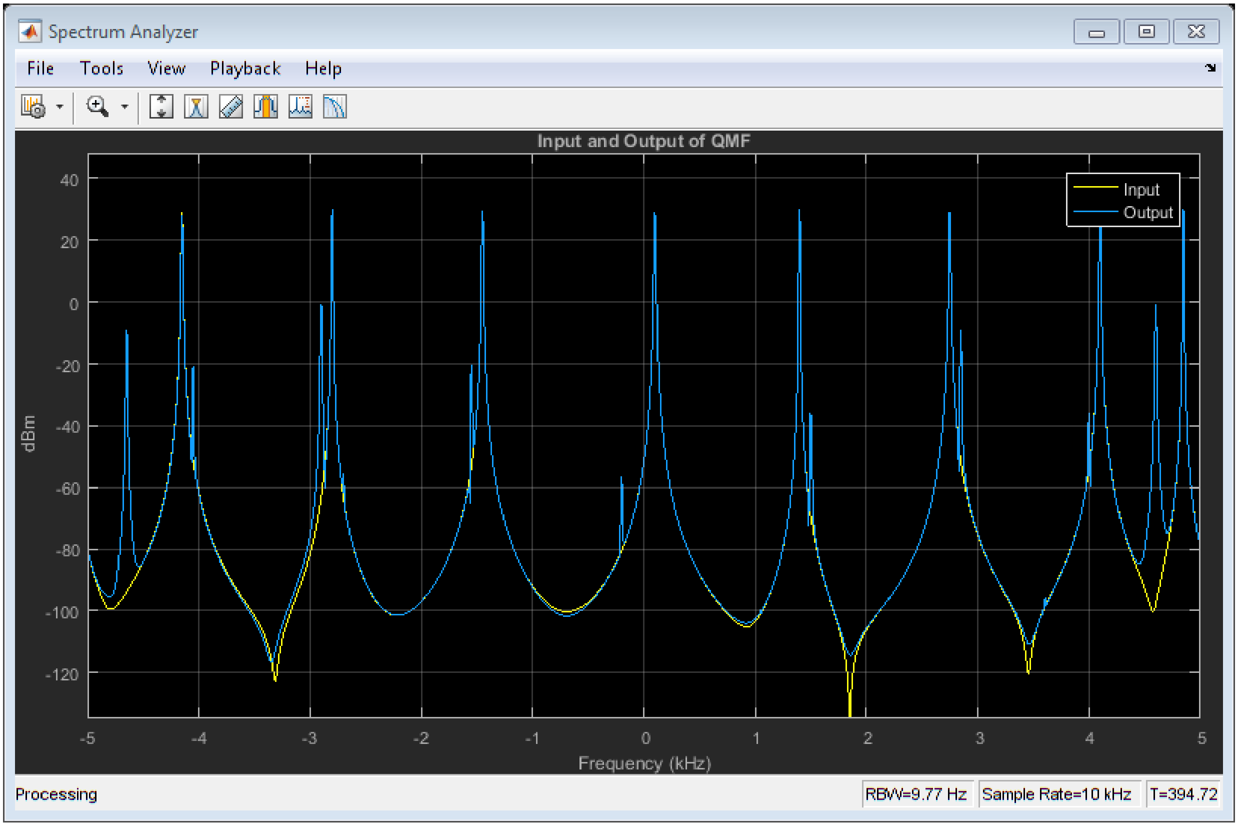
```
offsets = [-40,-30,-20,10,15,25,35,-15];
sinewave = dsp.SineWave('ComplexOutput',true,'Frequency',...
    offsets+(-375:125:500),'SamplesPerFrame',800);

channelizer = dsp.Channelizer('StopbandAttenuation',140);
synthesizer = dsp.ChannelSynthesizer('StopbandAttenuation',140);
spectrumAnalyzer =  dsp.SpectrumAnalyzer('ShowLegend',true,'NumInputPorts',...
    2,'ChannelNames',{'Input','Output'},'Title','Input and Output of QMF');
```

Streaming

Use the channelizer to split the broadband input signal into multiple narrow bands. Then pass the multiple narrowband signals into the synthesizer, which merges these signals to form the broadband signal. Compare the spectra of the input and output signals.

```
for i = 1:5000
    x = sum(sinewave(),2);
    y = channelizer(x);
    v = synthesizer(y);
```

```
    spectrumAnalyzer(x,v);
end
```

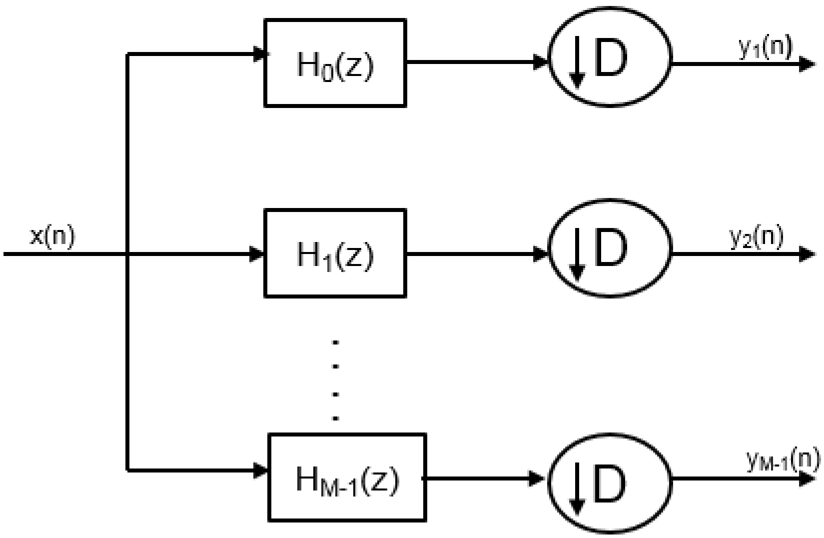


The input and output spectra match very closely.

Definitions

Analysis Filter Bank

The analysis filter bank consists of a series of parallel bandpass filters that split an input broadband signal, x(n), into a series of narrow subbands. Each bandpass filter retains a different portion of the input signal. After the bandwidth is reduced by one of the bandpass filters, the signal is downsampled to a lower sampling rate commensurate with the new bandwidth.



Prototype Lowpass Filter

To implement the analysis filter bank efficiently, the channelizer uses a prototype lowpass filter. This filter has an impulse response of h[n], a normalized two-sided bandwidth of $2\pi/M$, and a cutoff frequency of π/M. M is the number of frequency bands, that is, the branches of the analysis filter bank. This value corresponds to the FFT length that the filter bank uses. M can be high, in the order of 2048 or more. The stopband attenuation determines the minimum level of interference (aliasing) from one frequency band to another. The passband ripple must be small so that the input signal is not distorted in the passband.

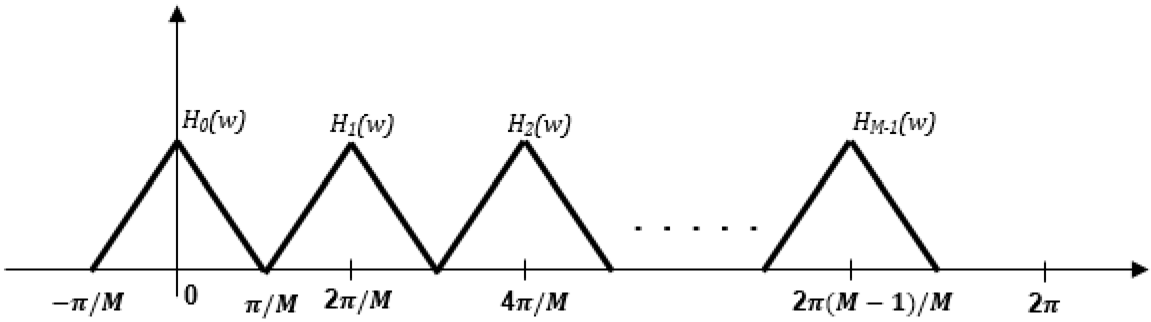
The prototype lowpass filter models the first branch of the filter bank. The other $M-1$ branches are modeled by filters that are modulated versions of the prototype filter. The modulation factor is given by $e^{-jw_k n}, \quad w_k = 2\pi k / M, \quad k = 0, 1, \ldots, M-1$.

Using Prototype Lowpass Filter

The transfer function of the modulated kth bandpass filter is given by:

$$H_k(z) = H_0(ze^{-jw_k}), \quad w_k = 2\pi k / M, \quad k = 1,2,\ldots,M-1$$

This figure shows the frequency response of *M* filters.

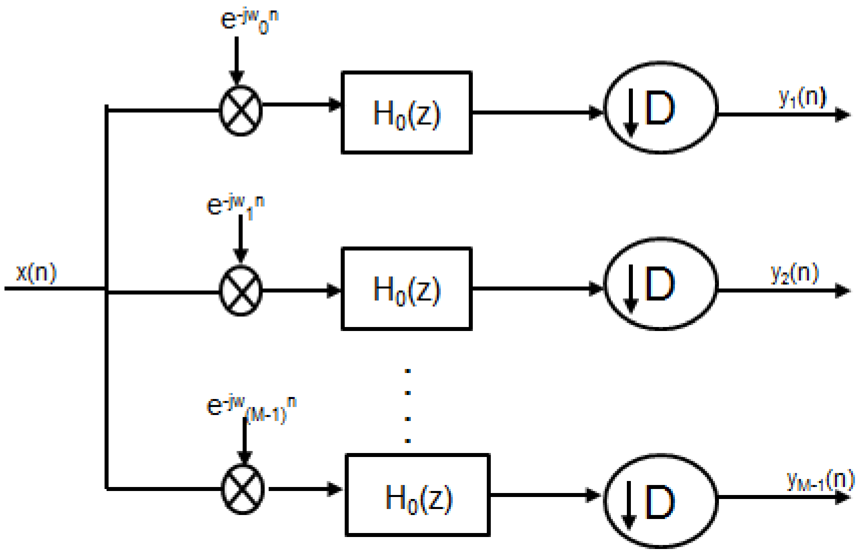


To obtain the frequency response characteristics of the filter $H_k(z)$, where $k = 1,\ldots,M-1$, uniformly shift the frequency response of the prototype filter, $H_0(z)$, by multiples of $2\pi/M$. Each subband filter, H_k(z), $\{k = 1,\ldots,M-1\}$, is derived from the prototype filter.

Shift Narrow Subbands to Baseband

The frequency components in the input signal, *x(n)*, are translated in frequency to baseband by multiplying *x(n)* with the complex exponentials, $e^{-jw_k n}, \quad where \quad w_k = 2\pi k / M, and\ k = 1,2,..,M-1$, where, $w_k = 2\pi k / M$, and $k = 1,2,\ldots,M-1$. The resulting product signals are passed through the lowpass filters, $H_0(z)$. The output of the lowpass filter is relatively narrow in bandwidth. Downsample the signal commensurate with the new bandwidth. Choose a decimation factor, $D \le M$, the number of branches of the analysis filter bank.

The figure shows an analysis filter bank that uses the prototype lowpass filter.



$y_1(n)$, $y_2(n)$, ..., $y_{M-1}(n)$ are narrow subband signals translated into baseband.

Algorithm

Polyphase Implementation

The analysis filter bank can be implemented efficiently using the polyphase structure.

To derive the polyphase structure, start with the transfer function of the prototype lowpass filter.

$$H_0(z) = b_0 + b_1 z^{-1} + \ldots + b_N z^{-N}$$

N+1 is the length of the prototype filter.

You can rearrange this equation as follows:

$$H_0(z) = \begin{array}{l} \left(b_0 + b_M z^{-M} + b_{2M} z^{-2M} + .. + b_{N-M+1} z^{-(N-M+1)}\right) + \\ z^{-1}\left(b_1 + b_{M+1} z^{-M} + b_{2M+1} z^{-2M} + .. + b_{N-M+2} z^{-(N-M+1)}\right) + \\ \vdots \\ z^{-(M-1)}\left(b_{M-1} + b_{2M-1} z^{-M} + b_{3M-1} z^{-2M} + .. + b_N z^{-(N-M+1)}\right) \end{array}$$

M is the number of polyphase components.

You can write this equation as:

$$H_0(z) = E_0(z^M) + z^{-1} E_1(z^M) + ... + z^{-(M-1)} E_{M-1}(z^M)$$

$E_0(z^M)$, $E_1(z^M)$, ..., $E_{M-1}(z^M)$ are polyphase components of the prototype lowpass filter, H_0(z).

The other filters in the filter bank, H_k(z), where k = 1, ..., M-1, are modulated versions of this prototype filter.

You can write the transfer function of the kth modulated bandpass filter as $H_k(z) = H_0(ze^{-jw_k})$. Replacing z with ze^{-jw_k},

$$H_k(z) = h_0 + h_1 e^{-jwk} z^{-1} + h_2 e^{-j2wk} z^{-2} ... + h_N e^{-jNwk} z^{-N}$$

N+1 is the length of the kth filter.

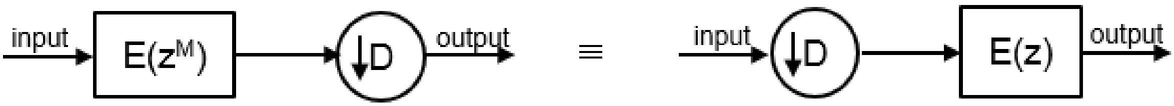
In polyphase form, the equation is as follows:

$$H_k(z) = \begin{bmatrix} 1 & e^{-jw_k} & e^{-j2w_k} & ... & e^{-j(M-1)w_k} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1} E_1(z^M) \\ \vdots \\ z^{-(M-1)} E_{M-1}(z^M) \end{bmatrix}$$

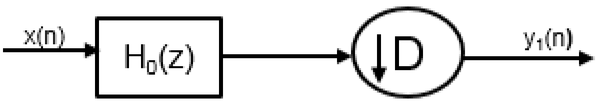
For all M channels in the filter bank, the MIMO transfer function, H(z), is given by:

$$H(z)=\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & e^{-jw_1} & e^{-j2w_1} & \cdots & e^{-j(M-1)w_1} \\ & & \vdots & & \\ 1 & e^{-jw_{M-1}} & e^{-j2w_{M-1}} & \cdots & e^{-j(M-1)w_{M-1}} \end{bmatrix}\begin{bmatrix} E_0(z^M) \\ z^{-1}E_1(z^M) \\ \vdots \\ z^{-(M-1)}E_{M-1}(z^M) \end{bmatrix}$$

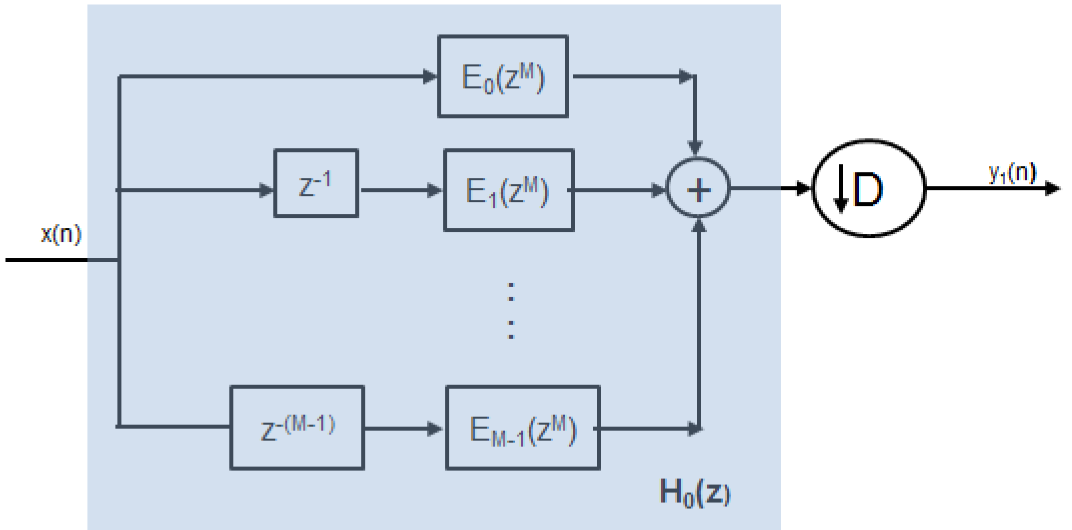
Here is the multirate noble identity for decimation.



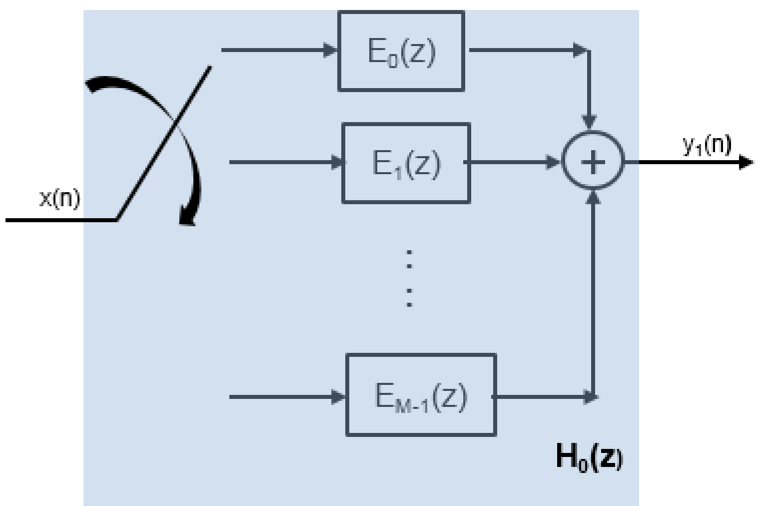
For illustration, consider the first branch of the filter bank that contains the lowpass filter.



Replace H_0(z) with its polyphase representation.



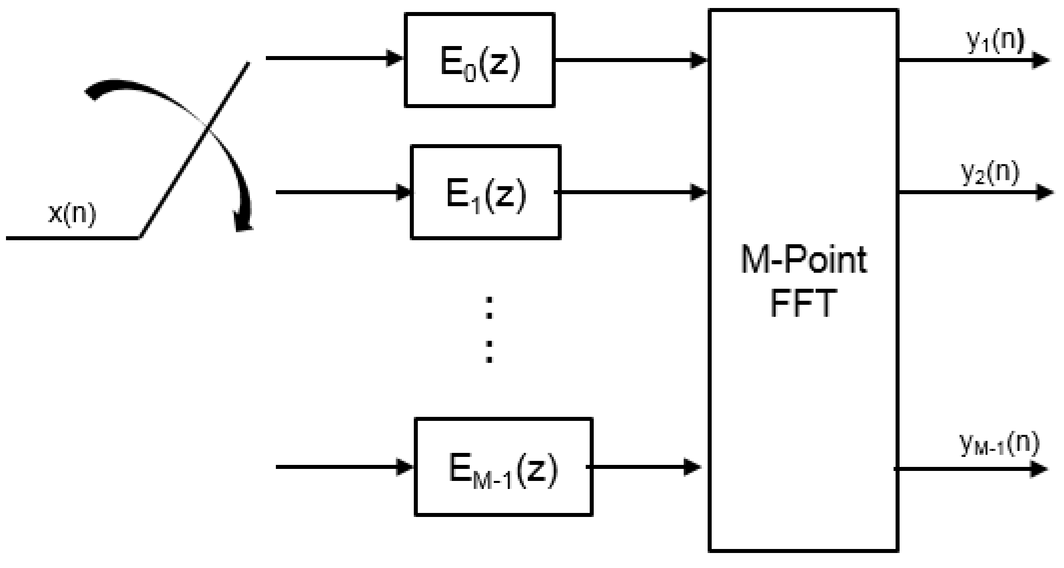
After applying the noble identity for decimation, you can replace the delays and the decimation factor with a commutator switch.



For all *M* channels in the filter bank, the MIMO transfer function, *H*(z), is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{-jw_1} & e^{-j2w_1} & \dots & e^{-j(M-1)w_1} \\ & & \vdots & & \\ 1 & e^{-jw_{M-1}} & e^{-j2w_{M-1}} & \dots & e^{-j(M-1)w_{M-1}} \end{bmatrix} \begin{bmatrix} E_0(z) \\ E_1(z) \\ \vdots \\ E_{M-1}(z) \end{bmatrix}$$

The matrix on the left is a DFT matrix. With the DFT matrix, the efficient implementation of the lowpass prototype based filter bank looks like the following.



References

[1] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.

[2] Harris, F.J., Chris Dick, Michael Rice. "Digital Receivers and Transmitters Using Polyphase Filter Banks for Wireless Communications." IEEE Transactions on microwave theory and techniques. Vol. 51, Number 4, April 2003.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

System Objects

dsp.ChannelSynthesizer | dsp.FIRHalfbandDecimator | dsp.FIRHalfbandInterpolator | dsp.IIRHalfbandDecimator | dsp.DyadicAnalysisFilterBank

Blocks

`Channel Synthesizer` | `Channelizer` | `Dyadic Analysis Filter Bank` | `Two-Channel Analysis Subband Filter`

Functions

`firpr2chfb`

Introduced in R2016b

coeffs

System object: dsp.Channelizer
Package: dsp

Coefficients of prototype lowpass filter

Syntax

```
c = coeffs(channelizer)
```

Description

`c = coeffs(channelizer)` returns a structure that contains the coefficients of the prototype lowpass filter. The value, `c`, equals the number of frequency bands times the number of coefficients per band. These values are given by the `NumFrequencyBands` and `NumTapsPerBand` properties of the channelizer respectively.

Introduced in R2016b

polyphase

System object: dsp.Channelizer
Package: dsp

Polyphase matrix

Syntax

```
p = polyphase(channelizer)
```

Description

`p = polyphase(channelizer)` returns the polyphase matrix used by the filter bank. Each row in the matrix corresponds to a polyphase branch. The number of columns in *p* corresponds to the number of filter taps per polyphase branch.

Introduced in R2016b

reset

System object: dsp.Channelizer
Package: dsp

Reset internal states of System object

Syntax

`reset(channelizer)`

Description

`reset(channelizer)` resets the states of the input object to their initial values. The initial state values correspond to the initial conditions for the difference equation defining the object. After you call the object using nonzero input data, the filter might have nonzero states. If you continue to pass nonzero input data to the object without first calling `reset`, the object might produce different outputs for an identical input.

Introduced in R2016b

step

System object: dsp.Channelizer
Package: dsp

Separate broadband signal into narrow bands

Syntax

```
y = step(channelizer,x)
```

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(channelizer,x)` separates the broadband input signal `x` into narrow band signals. Each signal forms a column in output signal `y`. The number of rows in `x` must be a multiple of the number of frequency bands. Each column of `x` corresponds to a separate channel. If M is the number of frequency bands, and `x` is a L-by-1 matrix, then `y` has dimensions, L/M-by-M. If `x` has more than one channel, that is, it has dimensions L-by-N, then `y` has dimensions L/M-by-M-by-N. The input `x` can be complex and supports single and double data types.

Introduced in R2016b

tf

System object: dsp.Channelizer
Package: dsp

Transfer function

Syntax

```
[b,a] = tf(channelizer)
```

Description

`[b,a] = tf(channelizer)` returns the vector of numerator coefficients, `b`, and the vector of denominator coefficients, `a`, of the prototype lowpass filter.

Introduced in R2016b

dsp.HDLChannelizer System object

Package: dsp

Polyphase filter bank and fast Fourier transform—optimized for HDL code generation

Description

The `HDLChannelizer` System object separates a broadband input signal into multiple narrowband output signals. It provides hardware speed and area optimization for streaming data applications. The object accepts scalar or vector input of real or complex data, provides hardware-friendly control signals, and has optional output frame control signals. You can achieve giga-sample-per-second (GSPS) throughput using vector input. The object implements a polyphase filter, with one subfilter per input vector element. The hardware implementation interleaves the subfilters, which results in sharing each filter multiplier (*FFT Length* / *Input Size*) times. The object implements the same pipelined Radix 2^2 FFT algorithm as the dsp.HDLFFT System object.

To process input data with an HDL-optimized channelizer:

1. Create and configure a `dsp.HDLChannelizer` System object. See “Construction” on page 4-316.
2. Call the step method to channelize the input data according to the properties of the `dsp.HDLChannelizer` object.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`channelize = dsp.HDLChannelizer` returns a System object, `channelize`, that implements a raised-cosine filter and an 8-point FFT.

`channelize = dsp.HDLChannelizer(Name,Value)` returns a System object, `channelize`, with additional options specified by one or more `Name,Value` pair arguments.

Properties

`NumFrequencyBands` — FFT length
`8` (default) | integer power of two

For HDL code generation, the FFT length must be a power of 2 from 2^3 to 2^{16}.

`FilterCoefficients` — Polyphase filter coefficients
`[ -0.032, 0.121, 0.318, 0.482, 0.546, 0.482, 0.318, 0.121, -0.032 ]` (default) | vector of numeric values

If the number of coefficients is not a multiple of `NumFrequencyBands`, the object pads this vector with zeros. The default filter specification is a raised-cosine FIR filter, `rcosdesign(0.25,2,4,'sqrt')`. You can specify a vector of coefficients or a call to a filter design function that returns the coefficient values. Complex coefficients are not supported. By default, the object casts the coefficients to the same data type as the input.

`ComplexMultiplication` — HDL implementation of complex multipliers
`'Use 3 multipliers and 5 adders'` (default) | `'Use 4 multipliers and 2 adders'`

Each multiplication is implemented with either 3 multipliers and 5 adders, or 4 multipliers and 2 adders. Which option is faster or smaller depends on your synthesis tool and target device.

`OutputSize` — Size and order of output data
`'Same as number of frequency bands'` (default) | `'Same as input size'`

- `'Same as number of frequency bands'` — Output data is a 1-by-M vector, where M is the FFT length. The output order is bit natural.
- `'Same as input size'` — Output data is an M-by-1 vector, where M is the input vector size. The output order is bit reversed.

`Normalize` — FFT scaling
`true` (default) | `false`

When you set this property to `true`, the FFT implements an overall 1/N scale factor by scaling the result of each pipeline stage by 2. This adjustment keeps the output of the FFT in the same amplitude range as its input. If scaling is disabled, the FFT avoids overflow by increasing the word length by 1 bit at each stage.

`RoundingMethod` — Rounding method used for internal fixed-point calculations
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

See “Rounding Modes”. The object uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input is `single` or `double`. Each FFT stage rounds after the twiddle factor multiplication but before the butterflies. Rounding can also occur when casting the coefficients and the output of the polyphase filter to the data types you specify.

`OverflowAction` — Overflow handling for internal fixed-point calculations
`'Wrap'` (default) | `'Saturate'`

See “Overflow Handling”. The object uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input is `single` or `double`. This option applies to casting the coefficients and the output of the polyphase filter to the data types you specify.

The FFT algorithm avoids overflow by either scaling the output of each stage (`Normalize` enabled), or by increasing the word length by 1 bit at each stage (`Normalize` disabled).

`CoefficientsDataType` — Data type of the filter coefficients
`'Same word length as input'` (default) | `numerictype` object

The object casts the polyphase filter coefficients to this data type, using the rounding and overflow settings you specify. When you select `Inherit: Same word length as input` (default), the object selects the binary point using `fi()` best-precision rules.

`FilterOutputDataType` — Data type of the output of the polyphase filter
`'Same word length as input'` (default) | `numerictype` object

The object casts the output of the polyphase filter (the input to the FFT) to this data type, using the rounding and overflow settings you specify. When you select `Inherit: Same word length as input` (default), the object selects a best-precision binary point by considering the values of your filter coefficients and the range of your input data type.

By default, the FFT logic does not change the data type. When you disable `Normalize`, the FFT increases the word length by 1 bit at each stage to avoid overflow.

`ResetInputPort` — Optional reset signal
`false` (default) | `true`

When you enable this property, the object expects an extra argument, `reset`. When the `reset` input is `true`, the object stops calculation and clears all internal state.

`StartOutputPort` — Optional control signal indicating start of data

`false` (default) | `true`

When you enable this property, the object returns an extra value, `startOut`. The `startOut` signal is true for the first cycle of output data in a frame.

`EndOutputPort` — Optional control signal indicating end of data

`false` (default) | `true`

When you enable this property, the object returns an extra value, `endOut`. The `endOut` signal is true for the last cycle of output data in a frame.

Methods

getLatency	Latency of channelizer operation
step	Filter and compute FFT

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Create Channelizer for HDL Generation

Create a function that contains the channelizer object and supports HDL code generation.

Create the specifications and input signal. The signal has 8 frequency channels.

```
N = 8;
loopCount = 1024;
```

```
offsets = [-40 -30 -20 10 15 25 35 -15];
sinewave = dsp.SineWave('ComplexOutput',true,'Frequency', ...
     offsets+(-375:125:500),'SamplesPerFrame',loopCount);
spectrumAnalyzer = dsp.SpectrumAnalyzer('ShowLegend',true, ...
     'SampleRate',sinewave.SampleRate/N);
```

Write a function that creates and calls the channelizer System object™. You can generate HDL from this function.

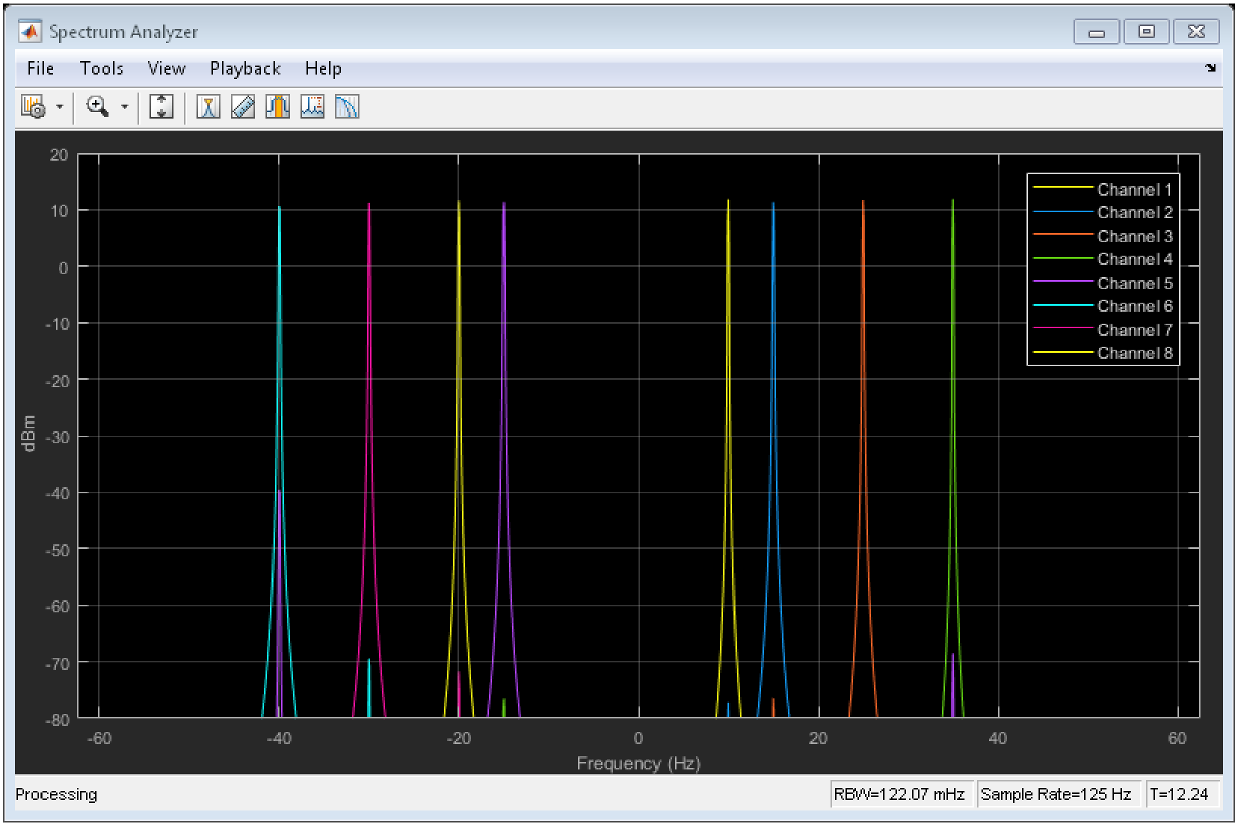
```
function [yOut,validOut] = HDLChannelizer8(yIn,validIn)
%HDLChannelizer8
% Process one sample of data using the dsp.HDLChannelizer System object
% yIn is a fixed-point scalar or column vector.
% validIn is a logical scalar value.
% You can generate HDL code from this function.

  persistent channelize8;
  if isempty(channelize8)
    % Use filter coeffs from non-HDL channelizer, or supply your own.
    channelizer = dsp.Channelizer('NumFrequencyBands',8);
    channelize8 = dsp.HDLChannelizer('NumFrequencyBands',8,'FilterCoefficients',tf(char
  end
  [yOut,validOut] = channelize8(yIn,validIn);
end
```

Channelize the input data by calling the object for each data sample.

```
y = zeros(loopCount/N,N);
validOut = false(loopCount/N,1);
yValid = zeros(loopCount/(N*N),N);
for loop2=1:20
   x = sum(sinewave(),2);
   for loop=1:length(x)
      [y(loop,:),validOut(loop)]= HDLChannelizer8(complex(x(loop)),true);
   end
   yValid = y(validOut == 1,:);
   spectrumAnalyzer(yValid);
end
```

```
channelize8 =
```

```
dsp.HDLChannelizer with properties:

      NumFrequencyBands: 8
     FilterCoefficients: [1×96 double]
  ComplexMultiplication: 'Use 4 multipliers and 2 adders'
             OutputSize: 'Same as number of frequency bands'
              Normalize: true

Use get to show all properties
```



Explore Latency of the HDL Channelizer Object

The latency of the object varies with the FFT length and the vector size. Use the `getLatency` method to find the latency of a particular configuration. The latency is measured as the number of cycles between the first valid input and the first valid output, assuming the input is contiguous. The number of filter coefficients does not affect the latency. Setting the output size equal to the input size reduces the latency because the samples are not saved and reordered.

Create a new `dsp.HDLChannelizer` object and request the latency.

```
channelize = dsp.HDLChannelizer('NumFrequencyBands',512);
L512 = getLatency(channelize)
```

```
L512 =

        1118
```

Request hypothetical latency information about a similar object with a different number of frequency bands (FFT length). The properties of the original object do not change.

```
L256 = getLatency(channelize,256)
N = channelize.NumFrequencyBands
```

```
L256 =

   592


N =

   512
```

Request hypothetical latency information of a similar object that accepts eight-sample vector input.

```
L256v8 = getLatency(channelize,256,8)
```

```
L256v8 =

   132
```

Set `Normalize` to `true`. The latency does not change.

```
channelize.Normalize = true;
L512n = getLatency(channelize)
```

```
L512n =

        1118
```

Change the `OutputSize` to match the input size. The latency decreases because the object does not need to store and reorder the data before output. The default input size is scalar.

```
channelize.OutputSize = 'Same as input size';
L512r = getLatency(channelize)
```

```
L512r =

   605
```

Check the latency of a vector input implementation with the output size the same as the input size. Specify the current value of the FFT length, and a vector size of 8 samples. The latency decreases because the object computes results in parallel when the input is a vector.

```
L256rv8 = getLatency(channelize,channelize.NumFrequencyBands,8)
```

```
L256rv8 =

   145
```

Algorithm

This object implements the algorithm described on the `Channelizer HDL Optimized` block reference page.

Latency

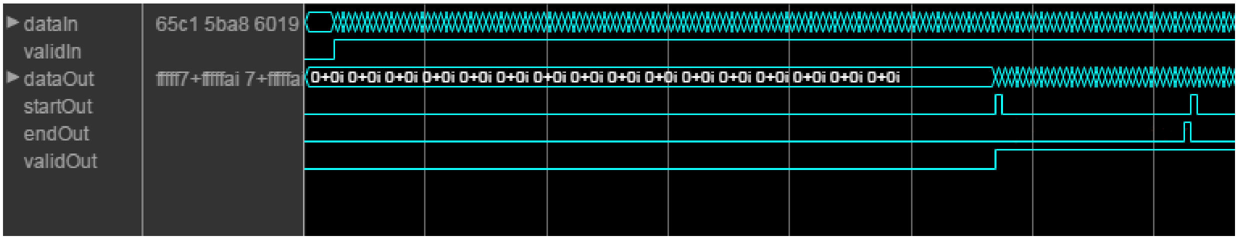
The latency varies with the FFT length and the vector size. Use the getLatency method to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming that the input is contiguous. The filter coefficients do not affect the latency. Setting the output size equal to the input size reduces the latency, because the samples are not saved and reordered.

Control Signals

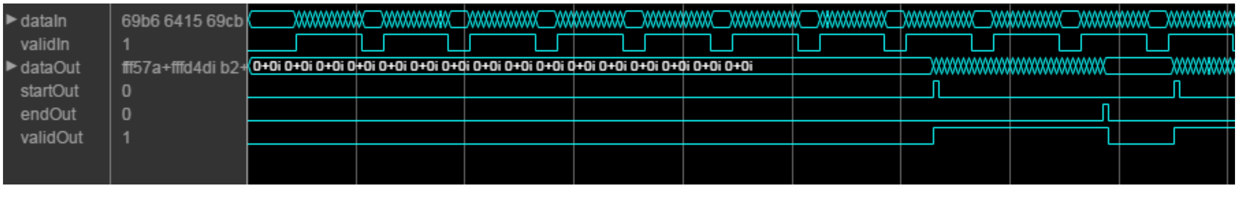
This diagram shows `validIn` and `validOut` signals for contiguous input data with a vector size of 16 and an FFT length of 512.

The diagram also shows the optional `startOut` and `endOut` signals that indicate frame boundaries. If you enable `startOut`, it pulses for one cycle with the first `validOut` of the frame. If you enable `endOut`, it pulses for one cycle with the last `validOut` of the frame.

If you apply continuous input frames, the output will also be continuous, after the initial latency.



The `validIn` signal can be noncontiguous. Data accompanied by a `validIn` signal is stored until a frame is filled, and output in a contiguous frame of *N* (FFT length) cycles. This diagram shows noncontiguous input and contiguous output for an FFT length of 512 and a vector size of 16 samples.



Performance

These resource and performance data are the place-and-route results from the generated HDL targeted to a Xilinx Virtex 6 (XC6VLX240-1ff784) FPGA. The three examples in the tables use this configuration:

- FFT length — 8
- Filter length — 96 coefficients
- 16-bit complex input data
- Coefficient and filter output data types same word length as input data (default)
- Complex multiplication (default) — 4 multipliers, 2 adders
- Output scaling — Enabled
- Minimize clock enables (HDL Coder parameter)

Performance of the synthesized HDL code varies with your target and synthesis options.

For scalar input, the design achieves a clock frequency of 346 MHz. The latency is 53 cycles. The subfilters share each multiplier eight (*N*) times. The design uses these resources.

Resource	Uses
LUT	1591
FFS	2681
Xilinx LogiCORE DSP48	16

For four-sample vector input, the design achieves a clock frequency of 333 MHz. The latency is 31 cycles. The subfilters share each multiplier twice (*N*/*M*). The design uses these resources.

Resource	Uses
LUT	1912
FFS	3986
Xilinx LogiCORE DSP48	56

For eight-sample vector input, the design achieves a clock frequency of 292 MHz. The latency is 20 cycles. When the input size is the same as the FFT length, the subfilters do not share any multipliers. The design uses these resources.

Resource	Uses
LUT	1388
FFS	2302
Xilinx LogiCORE DSP48	110

See Also

See Also

Blocks
`Channelizer HDL Optimized`

System Objects
dsp.Channelizer

Introduced in R2017a

getLatency

System object: dsp.HDLChannelizer
Package: dsp

Latency of channelizer operation

Syntax

```
Y = getLatency(channelize)
Y = getLatency(channelize,N)
Y = getLatency(channelize,N,V)
```

Description

`Y = getLatency(channelize)` returns the number of cycles, `Y`, that the object takes to calculate the first channel analysis results from an input sample. The latency depends on the input vector size and the FFT length. The filter coefficients do not affect the latency.

`Y = getLatency(channelize,N)` returns the latency, `Y`, that an object with FFT Length `N` would take to channelize scalar input data. This method does not change the properties of `channelize`.

`Y = getLatency(channelize,N,V)` returns the latency, `Y`, that an object with FFT Length `N` would take to channelize vector input data of size `V`. This method does not change the properties of `channelize`.

Examples

Explore Latency of the HDL Channelizer Object

The latency of the object varies with the FFT length and the vector size. Use the `getLatency` method to find the latency of a particular configuration. The latency is measured as the number of cycles between the first valid input and the first valid output, assuming the input is contiguous. The number of filter coefficients does not affect the latency. Setting the output size equal to the input size reduces the latency because the samples are not saved and reordered.

Create a new `dsp.HDLChannelizer` object and request the latency.

```
channelize = dsp.HDLChannelizer('NumFrequencyBands',512);
L512 = getLatency(channelize)
```

```
L512 =

        1118
```

Request hypothetical latency information about a similar object with a different number of frequency bands (FFT length). The properties of the original object do not change.

```
L256 = getLatency(channelize,256)
N = channelize.NumFrequencyBands
```

```
L256 =

   592


N =

   512
```

Request hypothetical latency information of a similar object that accepts eight-sample vector input.

```
L256v8 = getLatency(channelize,256,8)
```

```
L256v8 =

   132
```

Set `Normalize` to `true`. The latency does not change.

```
channelize.Normalize = true;
```

```
L512n = getLatency(channelize)
```

```
L512n =

        1118
```

Change the `OutputSize` to match the input size. The latency decreases because the object does not need to store and reorder the data before output. The default input size is scalar.

```
channelize.OutputSize = 'Same as input size';
L512r = getLatency(channelize)
```

```
L512r =

   605
```

Check the latency of a vector input implementation with the output size the same as the input size. Specify the current value of the FFT length, and a vector size of 8 samples. The latency decreases because the object computes results in parallel when the input is a vector.

```
L256rv8 = getLatency(channelize,channelize.NumFrequencyBands,8)
```

```
L256rv8 =

   145
```

Input Arguments

`channelize` — Channelizer

`dsp.HDLChannelizer` System object

Specify a `dsp.HDLChannelizer` System object that you created and configured.

`N` — FFT length

integer power of two

Use this argument to request the latency of a similar object with FFT length `N`.

V — vector size
integer

Use this argument to request the latency of a similar object with `V`-sample vector input. When you do not specify this argument, the function assumes scalar input.

Output Arguments

Y — Cycles of latency
integer

Cycles of latency that the object takes to channelize the input data. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous. Each call to the object simulates one cycle.

Introduced in R2017a

step

System object: dsp.HDLChannelizer
Package: dsp

Filter and compute FFT

Syntax

```
[dataOut,validOut] = step(obj,dataIn,validIn)
[dataOut,validOut] = step(obj,dataIn,validIn,reset)
[dataOut,startOut,endOut,validOut] = step(obj,___)
```

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[dataOut,validOut] = step(obj,dataIn,validIn)` filters and computes a fast Fourier transform, and returns the frequency channels, `dataOut`, detected in the input signal, `dataIn`, when `validIn` is `true`. The `validIn` and `validOut` arguments are logical scalars that indicate the validity of the input and output signals, respectively.

`[dataOut,validOut] = step(obj,dataIn,validIn,reset)` returns the frequency channels, `dataOut`, detected in the input signal, `dataIn`, when `validIn` is `true` and `reset` is `false`. When `reset` is `true`, the object stops the current calculation and clears all internal state. To use this syntax, set the `ResetInputPort` property to `true`.

`[dataOut,startOut,endOut,validOut] = step(obj,___)` returns the frequency channels, `dataOut`, computed from the input arguments of any of the previous syntaxes. `startOut` is `true` on the first sample of a frame of output data. `endOut` is `true` for the last sample of a frame of output data. To use this syntax, set the `StartOutputPort` and `EndOutputPort` properties to `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Input Arguments

`obj` — Channelizer

`dsp.HDLChannelizer` System object

Specify a `dsp.HDLChannelizer` System object that you created and configured.

`dataIn` — Input data

scalar or column vector of real or complex values

The vector size must be a power of 2 between 1 and 64, that is not greater than the number of channels (FFT length). `double` and `single` are allowed for simulation but not for HDL code generation.

The object does not accept `uint64` data.

Data Types: `fi` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `single` | `double`
Complex Number Support: Yes

`validIn` — Indicates valid input data

scalar

When `validIn` is `true`, the object captures the value on `dataIn`.

Data Types: `logical`

`reset` — Reset control signal (optional)

scalar

When `reset` is `true`, the object stops the current calculation and clears internal state.

Dependencies

To enable this argument, set `ResetInputPort` to `true`.

Data Types: `logical`

Output Arguments

`dataOut` — Frequency channel output data
row vector

- If you set `OutputSize` to `'Same as number of frequency bands'` (default), the output data is a 1-by-M vector, where M is the FFT length. The output order is bit natural.
- If you set `OutputSize` to `'Same as input size'`, the output data is an M-by-1 vector, where M is the input vector size. The output order is bit reversed.

The data type is a result of the `FilterOutputDataType` and the bit growth in the FFT necessary to avoid overflow.

`validOut` — Indicates valid output data
scalar

The object sets `validOut` to `true` with each valid sample on `dataOut`.

Data Types: `logical`

`startOut` — Indicates the first valid cycle of output data (optional)
scalar

The object sets `startOut` to `true` during the first valid sample on `dataOut`.

Dependencies

To enable this argument, set `StartOutputPort` to `true`.

Data Types: `logical`

`endOut` — Indicates the last valid cycle of output data (optional)
scalar

The object sets `endOut` to `true` during the last valid sample on `dataOut`.

Dependencies

To enable this argument, set `EndOutputPort` to `true`.

Data Types: `logical`

Introduced in R2017a

dsp.ChannelSynthesizer System object

Package: dsp

Polyphase FFT synthesis filter bank

Description

The `dsp.ChannelSynthesizer` System object merges multiple narrowband signals into a broadband signal by using an FFT based synthesis filter bank. The filter bank uses a prototype lowpass filter and is implemented using a polyphase structure. You can specify the filter coefficients directly or through design parameters.

Each narrowband signal is stored as a column in the input signal, `x`. The number of columns in `x` corresponds to the number of frequency bands of the filter bank. If `x` is three-dimensional, each matrix corresponds to a separate channel. If M is the number of frequency bands, and `x` is an L-by-M matrix, then the output signal, `y`, has dimensions $L \times M$-by-1. If `x` has more than one channel, that is, it has dimensions L-by-M-by-N, then `y` has dimensions $L \times M$-by-N. The input `x` can be complex and supports single and double data types.

This object also accepts variable-size inputs. That is, once the object is locked, you can change the size of each input channel. The number of channels cannot change. This object supports C and C++ code generation.

To merge multiple narrowband signals into a broadband signal:

1. Create a `dsp.ChannelSynthesizer` object and set the properties of the object.
2. Call step to synthesize the signal.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`synthesizer = dsp.ChannelSynthesizer` creates a synthesizer object, using the default properties.

`synthesizer = dsp.ChannelSynthesizer(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

Example:

```
synthesizer = dsp.ChannelSynthesizer('NumTapsPerBand',20,'StopbandAttenuation',140);
```

Properties

`Specification` — Filter design parameters or coefficients

`'Number of taps per band and stopband attenuation'` (default) | `'Coefficients'`

Filter design parameters or filter coefficients, specified as one of these options:

- `'Number of taps per band and stopband attenuation'` — Specify the filter design parameters through the `NumTapsPerBand` and `StopbandAttenuation` properties.
- `'Coefficients'` — Specify the filter coefficients directly using `LowpassCoefficients`.

`NumTapsPerBand` — Number of filter coefficients per frequency band

`12` (default) | positive integer

Number of filter coefficients each polyphase branch uses, specified as a positive integer. The number of polyphase branches matches the number of frequency bands. The total number of filter coefficients for the prototype lowpass filter is given by product of the number of frequency bands and `NumTapsPerBand`. For a given stopband attenuation, increasing the number of taps per band narrows the transition width of the filter. As a result, there is more usable bandwidth for each frequency band at the expense of increased computation.

This property applies when you set `Specification` to `'Number of taps per band and stopband attenuation'`.

`StopbandAttenuation` — Stopband attenuation

`80` (default) | positive real scalar

Stopband attenuation of the lowpass filter, specified as a positive real scalar in dB. This value controls the maximum amount of aliasing from one frequency band to the next. Larger is the stopband attenuation, smaller is the passband ripple.

This property applies when you set `Specification` to `'Number of taps per band and stopband attenuation'`.

`LowpassCoefficients` — Coefficients of the prototype lowpass filter
`[1×49 double]` (default) | row vector

Coefficients of the prototype lowpass filter, specified as a row vector. There must be at least one coefficient per frequency band. If the length of the lowpass filter is less than the number of frequency bands, the object zero-pads the coefficients.

This property applies when you set `Specification` to `'Coefficients'`.

This property is tunable. You can change its value even after the object is locked.

Methods

coeffs	Coefficients of prototype lowpass filter
polyphase	Polyphase matrix
reset	Reset internal states of System object
step	Merge narrowband signals into a broadband signal
tf	Transfer function

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Quadrature Mirror Filter Bank

The quadrature mirror filter bank (QMF) contains an analysis filter bank section and a synthesis filter bank section. `dsp.Channelizer` implements the analysis filter bank.

`dsp.ChannelSynthesizer` implements the synthesis filter bank using the efficient polyphase implementation that is based on a prototype lowpass filter.

Initialization

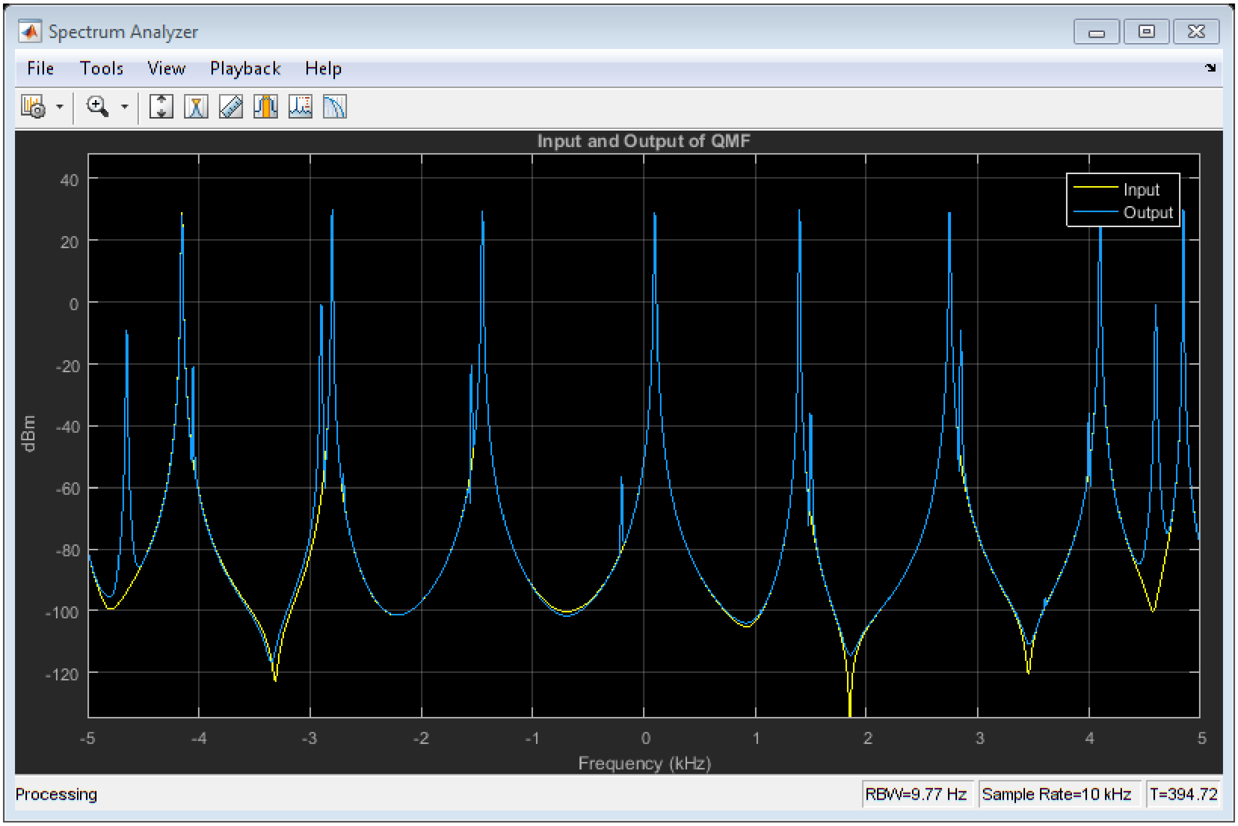
Initialize the `dsp.Channelizer` and `dsp.ChannelSynthesizer` System objects. Each object is set up with 8 frequency bands, 8 polyphase branches in each filter, 12 coefficients per polyphase branch, and a stopband attenuation of 140 dB. Use a sine wave with multiple frequencies as the input signal. View the input spectrum and the output spectrum using a spectrum analyzer.

```
offsets = [-40,-30,-20,10,15,25,35,-15];
sinewave = dsp.SineWave('ComplexOutput',true,'Frequency',...
    offsets+(-375:125:500),'SamplesPerFrame',800);

channelizer = dsp.Channelizer('StopbandAttenuation',140);
synthesizer = dsp.ChannelSynthesizer('StopbandAttenuation',140);
spectrumAnalyzer =  dsp.SpectrumAnalyzer('ShowLegend',true,'NumInputPorts',...
    2,'ChannelNames',{'Input','Output'},'Title','Input and Output of QMF');
```

Streaming

Use the channelizer to split the broadband input signal into multiple narrow bands. Then pass the multiple narrowband signals into the synthesizer, which merges these signals to form the broadband signal. Compare the spectra of the input and output signals.

```
for i = 1:5000
    x = sum(sinewave(),2);
    y = channelizer(x);
    v = synthesizer(y);
    spectrumAnalyzer(x,v);
end
```

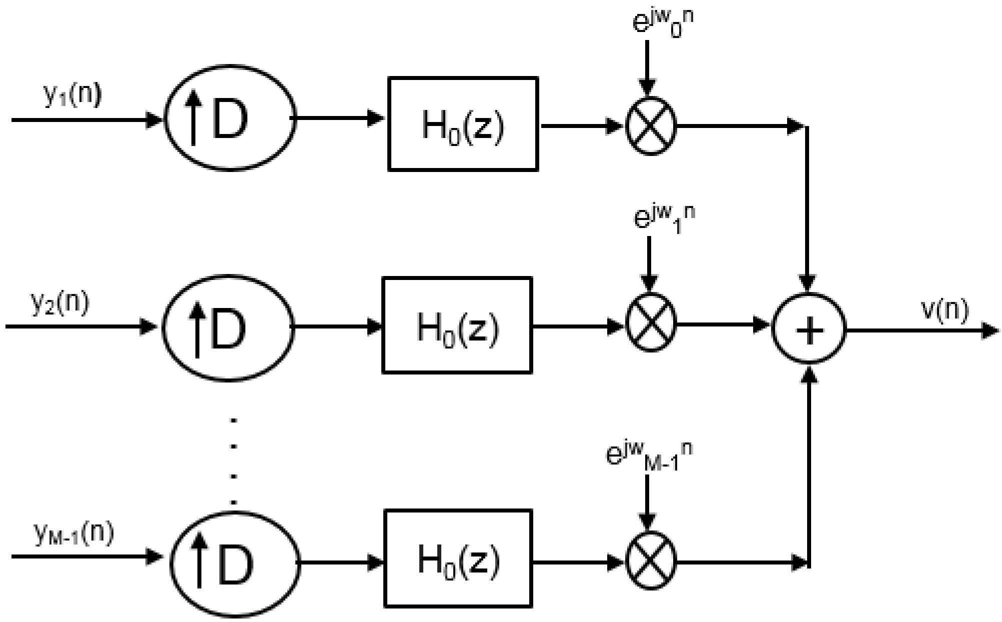


The input and output spectra match very closely.

Definitions

Synthesis Filter Bank

The synthesis filter bank consists of a set of parallel bandpass filters that merge multiple input narrowband signals, y_1(n), y_2(n),..., y_{M-1}(n) into a single broadband signal, *v(n)*. The input narrowband signals are in the baseband. Each narrowband signal is interpolated to a higher sampling rate by using the upsampler, and then filtered by the lowpass filter. A complex exponential that follows the lowpass filter centers the baseband signal around w_k.



Prototype Lowpass Filter

To implement the synthesis filter bank efficiently, the synthesizer uses a prototype lowpass filter. This filter has an impulse response of h[n], a normalized two-sided bandwidth of $2\pi/M$, and a cutoff frequency of π/M. M is the number of frequency bands, that is, the branches of the synthesis filter bank. This value corresponds to the FFT length that the filter bank uses. M can be high, in the order of 2048 or more. The stopband attenuation determines the minimum level of interference (aliasing) from one frequency band to another. The passband ripple must be small so that the input signal is not distorted in the passband.

The prototype lowpass filter models the first branch of the filter bank. The other $M - 1$ branches are modeled by filters that are modulated versions of the prototype filter. The modulation factor is given by $e^{jw_k n}, \quad w_k = 2\pi k / M, \quad k = 0,1,\ldots,M-1$.

The output of each bandpass filter forms a specific portion of the broadband signal. The output of all the branches are added to form the broadband signal, *v(n)*.

Algorithm

Polyphase Implementation

The synthesis filter bank can be implemented efficiently using the polyphase structure.

To derive the polyphase structure, start with the transfer function of the prototype lowpass filter.

$$H_0(z) = b_0 + b_1 z^{-1} + \ldots + b_N z^{-N}$$

N+1 is the length of the prototype filter.

You can rearrange this equation as follows:

$$\begin{aligned} H_0(z) = \; & \left(b_0 + b_M z^{-M} + b_{2M} z^{-2M} + .. + b_{N-M+1} z^{-(N-M+1)}\right) + \\ & z^{-1}\left(b_1 + b_{M+1} z^{-M} + b_{2M+1} z^{-2M} + .. + b_{N-M+2} z^{-(N-M+1)}\right) + \\ & \vdots \\ & z^{-(M-1)}\left(b_{M-1} + b_{2M-1} z^{-M} + b_{3M-1} z^{-2M} + .. + b_N z^{-(N-M+1)}\right) \end{aligned}$$

M is the number of polyphase components.

You can write this equation as:

$$H_0(z) = E_0(z^M) + z^{-1} E_1(z^M) + \ldots + z^{-(M-1)} E_{M-1}(z^M)$$

$E_0(z^M)$, $E_1(z^M)$,..., $E_{M-1}(z^M)$ are polyphase components of the prototype lowpass filter, H_0(z).

The other filters in the filter bank, H_k(z), where k = 1, ..., M-1, are modulated versions of this prototype filter.

You can write the transfer function of the kth modulated bandpass filter as $H_k(z) = H_0(ze^{jw_k})$. Replacing z with ze^{jw_k},

$$H_k(z) = h_0 + h_1 e^{jwk} z^{-1} + h_2 e^{j2wk} z^{-2} \ldots + h_N e^{jNwk} z^{-N}$$

N+1 is the length of the kth filter.

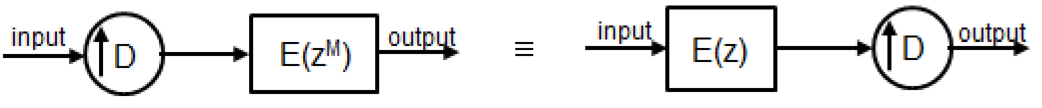
In polyphase form, the equation is as follows:

$$H_k(z) = \begin{bmatrix} 1 & e^{jw_k} & e^{j2w_k} & \ldots & e^{j(M-1)w_k} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1}E_1(z^M) \\ \vdots \\ z^{-(M-1)}E_{M-1}(z^M) \end{bmatrix}$$

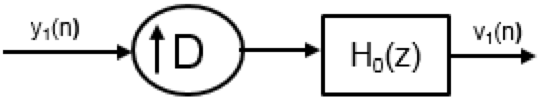
For all M channels in the filter bank, the MIMO transfer function, H(z), is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & e^{jw_1} & e^{j2w_1} & \ldots & e^{j(M-1)w_1} \\ & & \vdots & & \\ 1 & e^{jw_{M-1}} & e^{j2w_{M-1}} & \ldots & e^{j(M-1)w_{M-1}} \end{bmatrix} \begin{bmatrix} E_0(z^M) \\ z^{-1}E_1(z^M) \\ \vdots \\ z^{-(M-1)}E_{M-1}(z^M) \end{bmatrix}$$

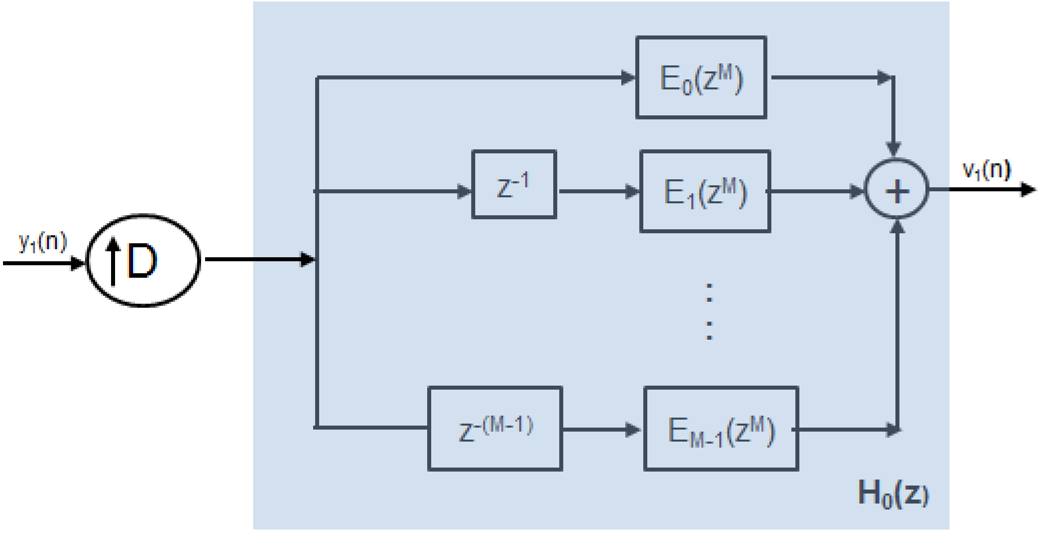
Here is the multirate noble identity for interpolation:



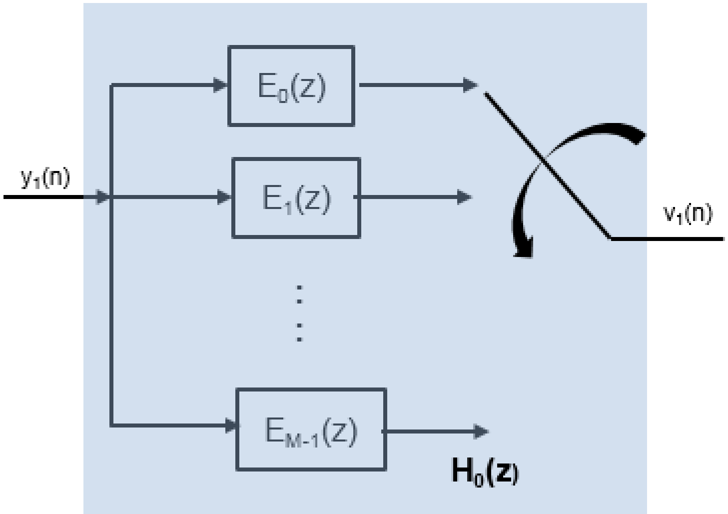
For illustration, consider the first branch of the filter bank that contains the lowpass filter.



Replace H_0(z) with its polyphase representation.



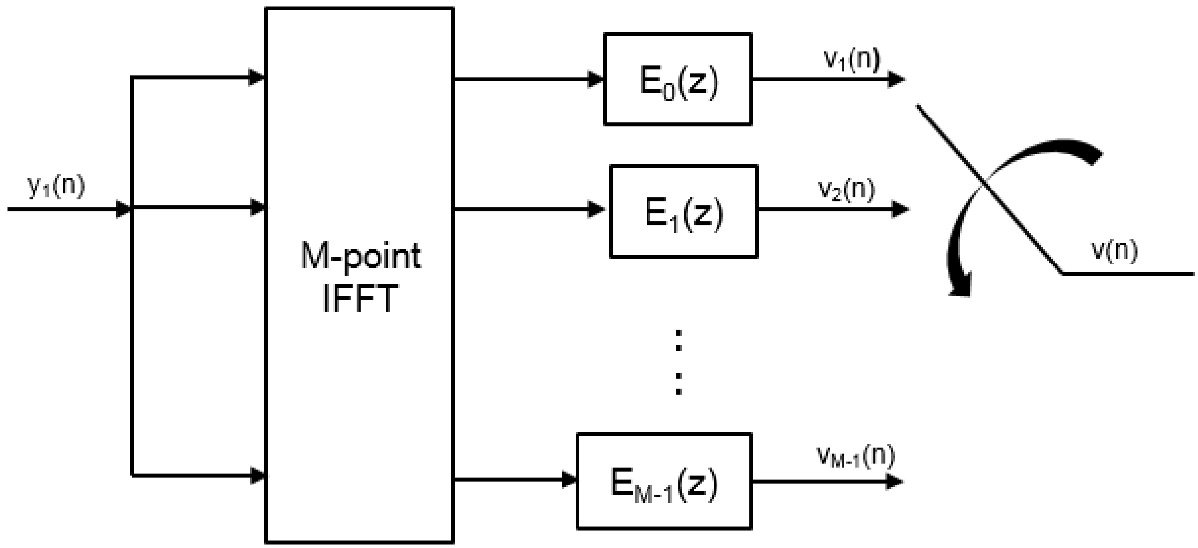
After applying the noble identity for interpolation, you can replace the delays, interpolation factor, and the adder with a commutator switch.



For all the M channels in the filter bank, the MIMO transfer function, $H(z)$, is given by:

$$H(z) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{jw_1} & e^{j2w_1} & \cdots & e^{j(M-1)w_1} \\ & & \vdots & & \\ 1 & e^{jw_{M-1}} & e^{j2w_{M-1}} & \cdots & e^{j(M-1)w_{M-1}} \end{bmatrix} \begin{bmatrix} E_0(z) \\ E_1(z) \\ \vdots \\ E_{M-1}(z) \end{bmatrix}$$

The matrix on the left is an IDFT matrix. With the IDFT matrix, the efficient implementation of the lowpass prototype based filter bank looks like the following.



References

[1] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.

[2] Harris, F.J., Chris Dick, Michael Rice. "Digital Receivers and Transmitters Using Polyphase Filter Banks for Wireless Communications." IEEE Transactions on microwave theory and techniques. Vol. 51, Number 4, April 2003.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

System Objects
dsp.Channelizer | dsp.FIRHalfbandDecimator | dsp.FIRHalfbandInterpolator | dsp.IIRHalfbandInterpolator | dsp.DyadicSynthesisFilterBank

Blocks
`Channel Synthesizer` | `Channelizer` | `Dyadic Analysis Filter Bank` | `Two-Channel Analysis Subband Filter`

Functions
`firpr2chfb`

Introduced in R2016b

coeffs

System object: dsp.ChannelSynthesizer
Package: dsp

Coefficients of prototype lowpass filter

Syntax

```
c = coeffs(synthesizer)
```

Description

`c = coeffs(synthesizer)` returns a structure that contains the coefficients of the prototype lowpass filter. This value is equal to the number of frequency bands times the number of coefficients per band. Number of frequency bands or the number of narrow band signals is given by the number of columns in the input signal. You can run `coeffs` on the synthesizer object only after the object is locked, that is, after you pass the input signal to the object.

Introduced in R2016b

polyphase

System object: dsp.ChannelSynthesizer
Package: dsp

Polyphase matrix

Syntax

```
p = polyphase(synthesizer)
```

Description

`p = polyphase(synthesizer)` returns the polyphase matrix used by the filter bank. Each row in the matrix corresponds to a polyphase branch. The number of columns in *p* corresponds to the number of filter taps per polyphase branch.

Introduced in R2016b

reset

System object: dsp.ChannelSynthesizer
Package: dsp

Reset internal states of System object

Syntax

```
reset(synthesizer)
```

Description

`reset(synthesizer)` resets the states of the input object to their initial values. The initial state values correspond to the initial conditions for the difference equation defining the object. After you call the object using nonzero input data, the filter might have nonzero states. If you continue to pass nonzero input data to the object without first calling `reset`, the object might produce different outputs for an identical input.

Introduced in R2016b

step

System object: dsp.ChannelSynthesizer
Package: dsp

Merge narrowband signals into a broadband signal

Syntax

```
y = step(synthesizer,x)
```

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(synthesizer,x)` merges the narrowband input signals contained as columns in `x` into broadband signal, `y`. The number of columns in `x` corresponds to the number of frequency bands. If `x` is three dimensional, each matrix corresponds to a separate channel. If M is the number of frequency bands, and `x` is a L-by-M matrix, then `y` has dimensions, $L*M$-by-1. If `x` has more than one channel, that is, L-by-M-by-N, then `y` has dimensions $L*M$-by-N. The input `x` can be complex and supports single and double data types.

Introduced in R2016b

tf

System object: dsp.ChannelSynthesizer
Package: dsp

Transfer function

Syntax

`[b,a] = tf(synthesizer)`

Description

`[b,a] = tf(synthesizer)` returns the vector of numerator coefficients, `b`, and the vector of denominator coefficients, `a`, of the prototype lowpass filter.

Introduced in R2016b

dsp.Chirp System object

Package: dsp

Generate swept-frequency cosine (chirp) signal

Description

The `Chirp` object generates a swept-frequency cosine (chirp) signal.

To generate the chirp signal:

1. Define and set up your chirp signal. See "Construction" on page 4-352.
2. Call step to generate the signal according to the properties of `dsp.Chirp`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`chirp = dsp.Chirp` returns a chirp signal, `chirp`, with unity amplitude.

`chirp = dsp.Chirp('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a chirp signal, `chirp`, with each specified property set to the specified value.

Properties

`Type`

Frequency sweep type

Specify the frequency sweep type as `Swept cosine`, `Linear`, `Logarithmic`, or `Quadratic`. This property specifies how the output instantaneous frequency sweep varies over time. The default value is `Linear`.

SweepDirection

Sweep direction

Specify the sweep direction as either `Unidirectional` or `Bidirectional`. The default value is `Unidirectional`.

InitialFrequency

Initial frequency (hertz)

When you set the `Type` on page 4-352 property to `Linear`, `Quadratic`, or `Logarithmic`, this property specifies the initial instantaneous frequency in hertz of the output chirp signal. When you set the `Type` on page 4-352 property to `Logarithmic`, the value of this property is one less than the actual initial frequency of the sweep. Also, when the sweep is logarithmic, the initial frequency must be less than the target frequency, specified by the `TargetFrequency` on page 4-353 property. This property is tunable. The default value is `1000`.

TargetFrequency

Target frequency (hertz)

When you set the `Type` on page 4-352 property to `Linear`, `Quadratic`, or `Logarithmic`, this property specifies the instantaneous frequency of the output signal in hertz at the target time. When you set the `Type` on page 4-352 property to `Swept Cosine`, the target frequency is the instantaneous frequency of the output at half the target time. Also, when the sweep is logarithmic, the target frequency must be greater than the initial frequency, specified by the `InitialFrequency` on page 4-353 property. This property is tunable. The default value is `4000`.

TargetTime

Target time

When you set the `Type` on page 4-352 property to `Linear`, `Quadratic`, or `Logarithmic`, this property specifies the target time in seconds at which the target frequency is reached. When you set the `Type` on page 4-352 property to `Swept cosine`, this property specifies the time at which the sweep reaches $2f_{tgt} - f_{init}$ Hz, where f_{tgt} is the `TargetFrequency` on page 4-353 and f_{init} is the `InitialFrequency` on page 4-353. The target time should not be greater than the sweep time, specified by the `SweepTime` on page 4-354 property. This property is tunable. The default value is `1`.

SweepTime

Sweep time

When you set the `SweepDirection` on page 4-353 property to `Unidirectional`, the sweep time in seconds is the period of the output frequency sweep. When you set the `SweepDirection` on page 4-353 property to `Bidirectional`, the sweep time is half the period of the output frequency sweep. The sweep time should be no less than the target time, specified by the `TargetTime` on page 4-353. This property must be a positive numeric scalar and is tunable. The default value is `1`.

InitialPhase

Initial phase

Specify initial phase of the output in radians at time $t = 0$. This property is tunable. The default value is `0`.

SampleRate

Sample rate

Specify the sampling rate of the output in hertz as a positive numeric scalar. The default value is `8000`.

SamplesPerFrame

Samples per output frame

Specify the number of samples to buffer into each output as a positive integer. The default value is `1`.

OutputDataType

Output data type

Specify the output data type as `double` or `single`. The default value is `double`.

Methods

reset	Reset internal states of chirp object

step	Generate chirp signal

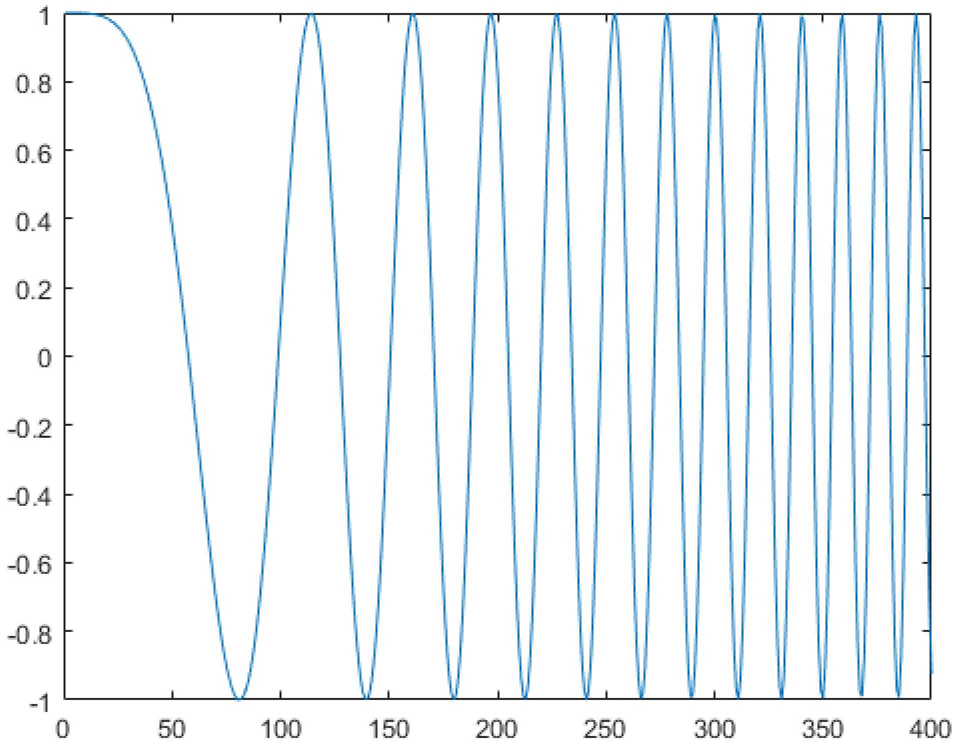
Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Generate a Bidirectional Swept Chirp Signal

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject() becomes step(myObject).

```
chirp = dsp.Chirp(...
    'SweepDirection', 'Bidirectional', ...
    'TargetFrequency', 25, ...
    'InitialFrequency', 0,...
    'TargetTime', 1, ...
    'SweepTime', 1, ...
    'SamplesPerFrame', 400, ...
    'SampleRate', 400);
plot(chirp());
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Chirp` block reference page. The object properties correspond to the block parameters.

See Also

dsp.SineWave

Introduced in R2012a

reset

System object: dsp.Chirp
Package: dsp

Reset internal states of chirp object

Syntax

```
reset(chirp)
```

Description

`reset(chirp)` sets the internal states of the `Chirp` object `chirp` to their initial values. After you reset `chirp`, the frequency sweep restarts from the beginning.

step

System object: dsp.Chirp
Package: dsp

Generate chirp signal

Syntax

`Y = step(chirp)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`Y = step(chirp)` returns a swept-frequency cosine output, `Y`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.CICCompensationDecimator System object

Package: dsp

Compensate for CIC decimation filter using FIR decimator

Description

You can compensate for the shortcomings of a CIC decimator, namely its passband droop and wide transition region, by following it with a compensation decimator. This System object lets you design and use such a filter. `dsp.CICCompensationDecimator` supports fixed-point operations and ARM Cortex code generation.

To compensate for the shortcomings of a CIC filter using an FIR decimator:

1. Define and set up your CIC compensation decimator. See “Construction” on page 4-359.
2. Call step to compensate for the passband droop and wide transition region of the CIC filter according to the properties of `dsp.CICCompensationDecimator`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ciccompdec = dsp.CICCompensationDecimator` returns a System object, `ciccompdec`, that applies an FIR decimator to each channel of an input signal. Using the properties of the object, the decimation filter can be designed to compensate for a preceding CIC filter.

`ciccompdec = dsp.CICCompensationDecimator(Name,Value)` returns a CIC compensation decimator System object, `ciccompdec`, with properties and options specified by one or more `Name,Value` pair arguments.

`ciccompdec = dsp.CICCompensationDecimator(decim,Name,Value)` returns a CIC compensation decimator System object, `ciccompdec`, with the `DecimationFactor` property set to `decim` and additional properties and options specified by one or more `Name,Value` pair arguments.

`ciccompdec = dsp.CICCompensationDecimator(cic,Name,Value)` returns a CIC compensation decimator System object, `ciccompdec`, with the `CICRateChangeFactor`, `CICNumSections`, and `CICDifferentialDelay` properties specified in the `dsp.CICDecimator` System object `cic` and additional properties and options specified by one or more `Name,Value` pair arguments.

`ciccompdec = dsp.CICCompensationDecimator(cic,decim,Name,Value)` returns a CIC compensation decimator System object, `ciccompdec`, with the `CICRateChangeFactor`, `CICNumSections`, and `CICDifferentialDelay` properties specified in the `dsp.CICDecimator` System object `cic`, the `DecimationFactor` property set to `decim`, and additional properties and options specified by one or more `Name,Value` pair arguments.

Properties

`CICDifferentialDelay` — Differential delay of the CIC filter being compensated

1 (default) | positive integer scalar

Specify the differential delay of the CIC filter being compensated as a positive integer scalar. The default is 1.

`CICNumSections` — Number of sections of the CIC filter being compensated

2 (default) | positive integer scalar

Specify the number of sections of the CIC filter being compensated as a positive integer scalar. The default is 2.

`CICRateChangeFactor` — Rate-change factor of the CIC filter being compensated

2 (default) | positive integer scalar

Specify the rate-change factor of the CIC filter being compensated as a positive integer scalar. The default is 2.

`DecimationFactor` — Decimation factor of compensator

2 (default) | positive integer scalar

Specify the decimation factor of the compensator System object as a positive integer scalar. The default is 2.

`DesignForMinimumOrder` — Design filter of minimum order or of specified order
`true` (default) | logical value

Specify whether to design a filter of minimum order or a filter of specified order as a logical scalar. The default is `true`, which corresponds to a filter of minimum order.

`FilterOrder` — Order of decimation compensator filter
12 (default) | positive integer scalar

Specify the order of the decimation compensator filter as a positive integer scalar. This property applies only when you set the `DesignForMinimumOrder` property to `false`. The default is 12.

`PassbandFrequency` — Passband edge frequency in hertz
100 kHz (default) | positive real scalar

Specify the passband edge frequency as a positive real scalar expressed in hertz. `PassbandFrequency` must be less than $F_s/2$, where F_s is the input sample rate. The default is 100 kHz.

`PassbandRipple` — Filter passband ripple in decibels
0.1 dB (default) | positive real scalar

Specify the filter passband ripple as a positive real scalar expressed in decibels. The default is 0.1 dB.

`SampleRate` — Input sample rate in hertz
600 kHz (default) | positive real scalar

Specify the input sample rate as a positive real scalar expressed in hertz. The default is 1200 kHz.

`StopbandAttenuation` — Filter stopband attenuation in decibels
60 dB (default) | positive real scalar

Specify the filter stopband attenuation as a positive real scalar expressed in decibels. The default is 60 dB

`StopbandFrequency` — Stopband edge frequency in hertz
400 kHz (default) | positive real scalar

Specify the stopband edge frequency as a positive real scalar expressed in hertz. `StopbandFrequency` must be less than $F_s/2$, where F_s is the input sample rate. The default is 400 kHz.

Fixed-Point Properties

`CoefficientsDataType` — Word and fraction lengths of coefficients
`numerictype(1,16)` (default) | `numerictype` object

Word and fraction lengths of coefficients, specified as a signed or unsigned `numerictype` object. The default, `numerictype(1,16)` corresponds to a signed numeric type object with 16-bit coefficients and a fraction length determined based on the coefficient values, to give the best possible precision.

This property is not tunable.

Word length of the output is same as the word length of the input. Fraction length of the output is computed such that the entire dynamic range of the output can be represented without overflow. For details on how the fraction length of the output is computed, see "Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters".

`RoundingMethod` — Rounding method for output fixed-point operations
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`

Rounding method for output fixed-point operations, specified as a character vector. For more information on the rounding modes, see "Precision and Range".

Methods

reset	Reset internal states of CIC compensation decimator
step	Compensate for preceding CIC filter

More "Analysis Methods for Filter System Objects" on page 3-2.

You can also type `dsp.CICCompensationDecimator.helpFilterAnalysis` at the command line to obtain a list of multirate discrete-time filter analysis methods supported for `dsp.CICCompensationDecimator` objects.

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

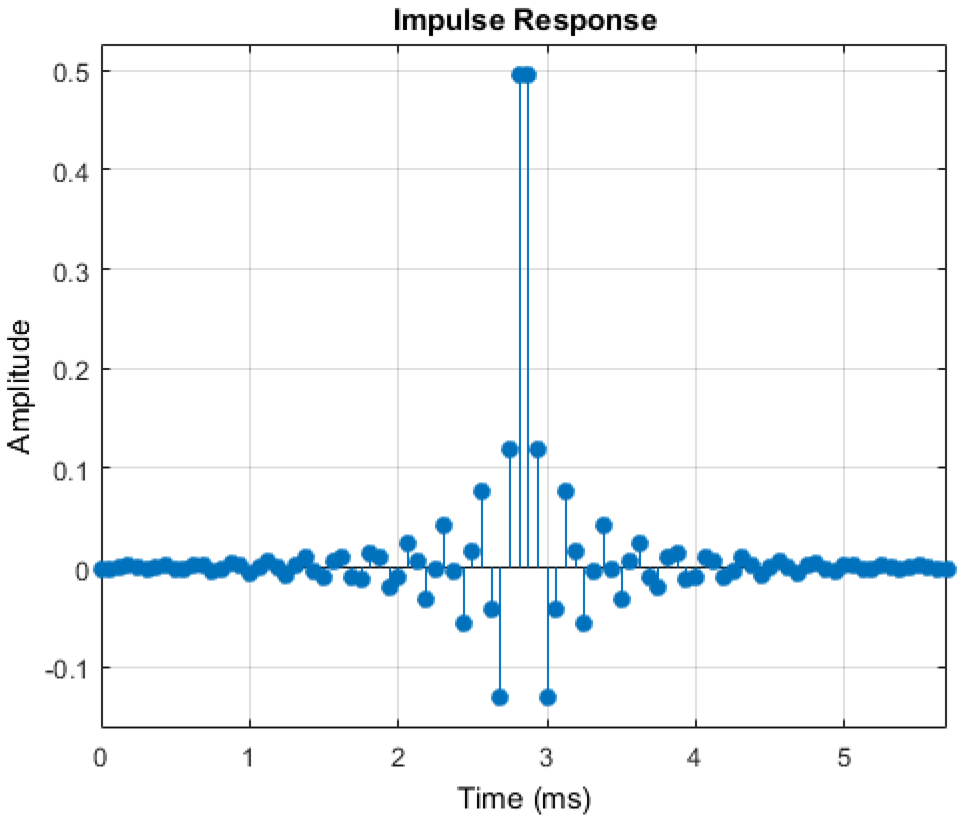
Impulse and Frequency Response of CIC Compensation Decimator

Design an CIC compensation decimator. Specify the decimation factor to be 2, passband frequency to be 4 kHz, stopband frequency to be 4.5 kHz, and the input sample rate to be 16 kHz.

```
fs = 16e3;
fPass = 4e3;
fStop = 4.5e3;

CICCompDecim = dsp.CICCompensationDecimator('DecimationFactor',2,'PassbandFrequency',fP
    'StopbandFrequency',fStop,'SampleRate',fs);
```

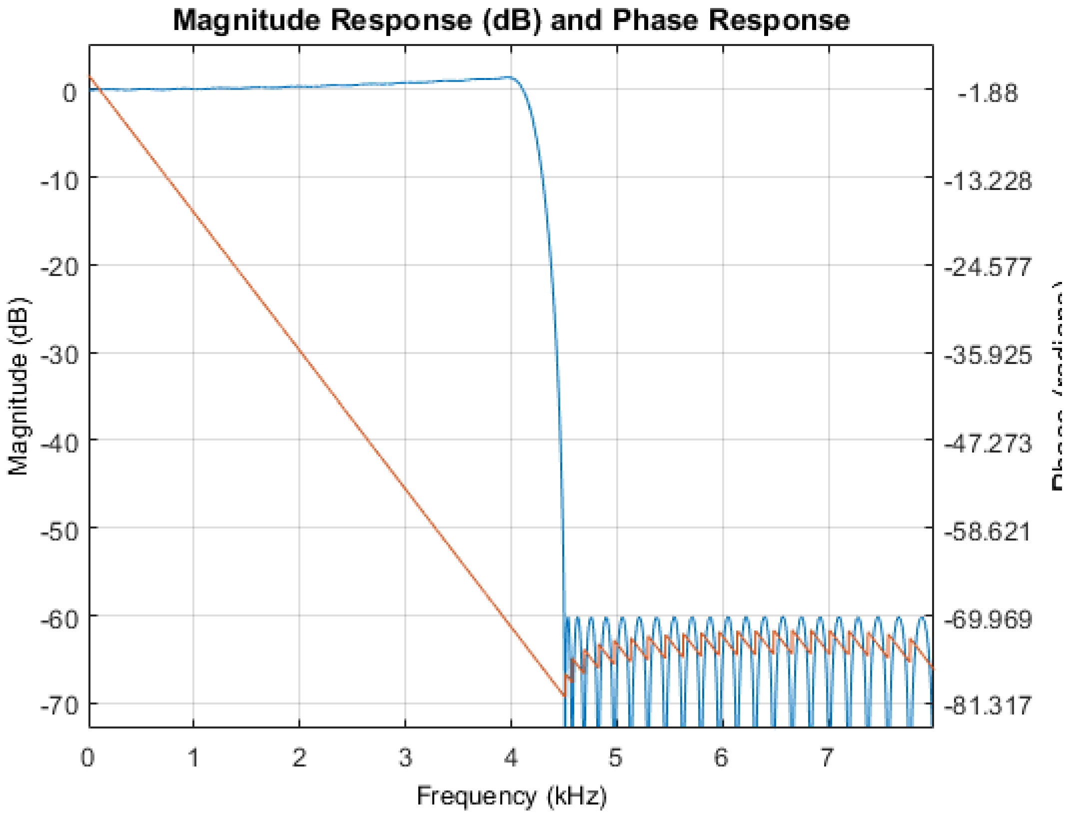
Plot the impulse response. The group delay of the filter is 45.5.

```
fvtool(CICCompDecim,'Analysis','impulse')
```



Plot the magnitude and Phase response.

```
fvtool(CICCompDecim,'Analysis','freq')
```



Compensation Decimator Design

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Design a compensation decimator for an existing CIC decimator having six sections and a decimation factor of 6.

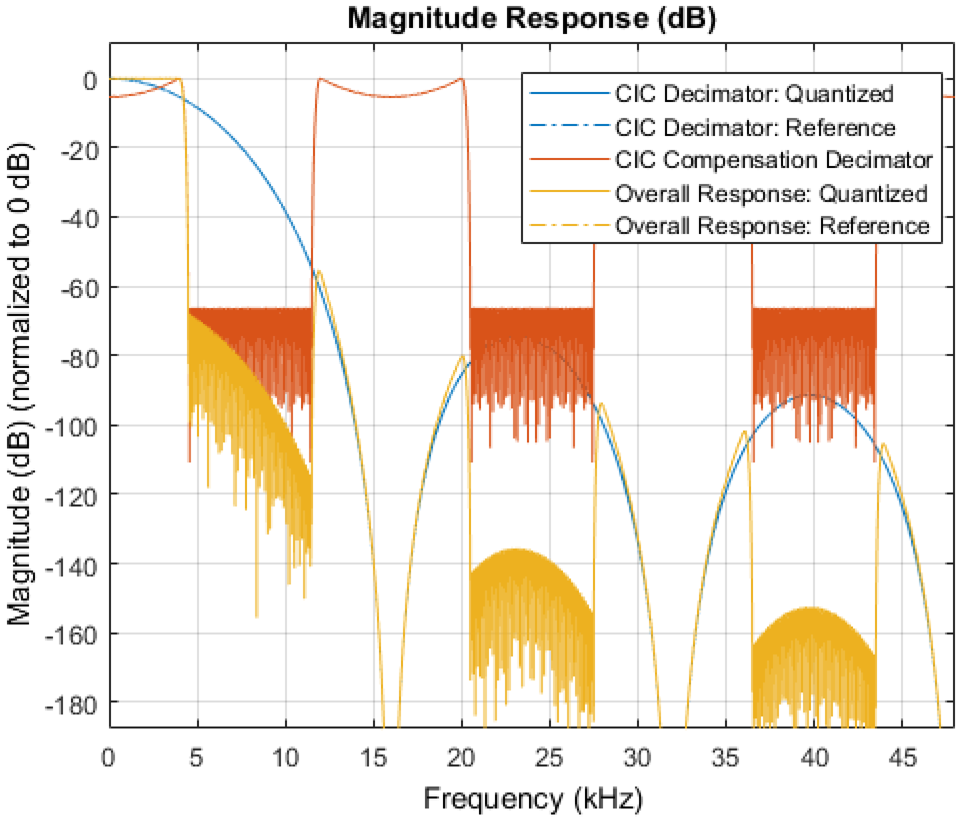
```
CICDecim = dsp.CICDecimator('DecimationFactor',6, ...
    'NumSections',6);
```

Construct the compensation decimator. Specify a decimation factor of 2, an input sample rate of 16 kHz, a passband frequency of 4 kHz, and a stopband frequency of 4.5 kHz.

```
fs = 16e3;
fPass = 4e3;
fStop = 4.5e3;

CICCompDecim = dsp.CICCompensationDecimator(CICDecim, ...
    'DecimationFactor',2,'PassbandFrequency',fPass, ...
    'StopbandFrequency',fStop,'SampleRate',fs);
```

Visualize the frequency response of the cascade. Normalize all magnitude responses to 0 dB.

```
filtCasc = dsp.FilterCascade(CICDecim,CICCompDecim);

f = fvtool(CICDecim, CICCompDecim, filtCasc, ...
    'Fs', [fs*6 fs fs*6]);

f.NormalizeMagnitudeto1 = 'on';
legend(f,'CIC Decimator','CIC Compensation Decimator', ...
    'Overall Response');
```



Apply the design to a 1200-sample random input signal.

```
x = dsp.SignalSource(fi(rand(1200,1),1,16,15),'SamplesPerFrame',120);

y = fi(zeros(0,1),1,32,20);
for ind = 1:10
    x2 = CICDecim(x());
    y = [y;CICCompDecim(x2)];
```

```
end
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

dsp.CICCompensationInterpolator | dsp.CICDecimator | dsp.CICInterpolator

Introduced in R2014b

reset

System object: dsp.CICCompensationDecimator
Package: dsp

Reset internal states of CIC compensation decimator

Syntax

`reset(ciccompdec)`

Description

`reset(ciccompdec)` resets the internal states of a `CICCompensationDecimator` System object, `ciccompdec`, to their initial values.

Input Arguments

`ciccompdec` — CIC compensation decimator

`CICCompensationDecimator` System object

CIC compensation decimator, specified as a `CICCompensationDecimator` System object.

step

System object: dsp.CICCompensationDecimator
Package: dsp

Compensate for preceding CIC filter

Syntax

```
y = step(ciccompdec,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(ciccompdec,x)` returns the filtered and downsampled values, `y`, of the input signal, `x`. The System object treats a $K_i \times N$ input matrix as N independent channels, decimating each channel over the first dimension. The result is a $K_o \times N$ output matrix, where $K_o = K_i / M$ and M is the decimation factor. The object supports any real or complex floating-point or fixed-point input, with the exception of complex unsigned fixed-point data.

Input Arguments

`ciccompdec` — CIC compensation decimator

`CICCompensationDecimator` System object

CIC compensation decimator, specified as a `CICCompensationDecimator` System object.

`x` — Input signal

vector | matrix

Input signal, specified as a vector or matrix.

Output Arguments

y — Filtered and downsampled signal

vector | matrix

Filtered and downsampled signal, returned as a vector or matrix.

See Also

See Also

dsp.CICCompensationInterpolator

dsp.CICCompensationInterpolator System object

Package: dsp

Compensate for CIC interpolation filter using FIR interpolator

Description

You can compensate for the shortcomings of a CIC interpolator, namely its passband droop and wide transition region, by preceding it with a compensation interpolator. This System object lets you design and use such a filter. `dsp.CICCompensationInterpolator` supports fixed-point operations and ARM Cortex code generation.

To compensate for the shortcomings of a CIC filter using an FIR interpolator:

1. Define and set up your CIC compensation interpolator. See “Construction” on page 4-372.
2. Call step to compensate for the passband droop and wide transition region of the CIC filter according to the properties of `dsp.CICCompensationInterpolator`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ciccompint = dsp.CICCompensationInterpolator` returns a System object, `ciccompint`, that applies an FIR interpolator to each channel of an input signal. Using the properties of the object, the interpolation filter can be designed to compensate for a subsequent CIC filter.

`ciccompint = dsp.CICCompensationInterpolator(Name,Value)` returns a CIC compensation interpolator System object, `ciccompint`, with properties and options specified by one or more `Name,Value` pair arguments.

`ciccompint = dsp.CICCompensationInterpolator(interp,Name,Value)` returns a CIC compensation interpolator System object, `ciccompint`, with the `InterpolationFactor` property set to `interp` and additional properties and options specified by one or more `Name,Value` pair arguments.

`ciccompint = dsp.CICCompensationInterpolator(cic,Name,Value)` returns a CIC compensation interpolator System object, `ciccompint`, with the `CICRateChangeFactor`, `CICNumSections`, and `CICDifferentialDelay` properties specified in the `dsp.CICInterpolator` System object `cic` and additional properties and options specified by one or more `Name,Value` pair arguments.

`ciccompint = dsp.CICCompensationInterpolator(cic,interp,Name,Value)` returns a CIC compensation interpolator System object, `ciccompint`, with the `CICRateChangeFactor`, `CICNumSections`, and `CICDifferentialDelay` properties specified in the `dsp.CICInterpolator` System object `cic`, the `InterpolationFactor` property set to `interp`, and additional properties and options specified by one or more `Name,Value` pair arguments.

Properties

`CICDifferentialDelay` — Differential delay of the CIC filter being compensated

1 (default) | positive integer scalar

Specify the differential delay of the CIC filter being compensated as a positive integer scalar. The default is 1.

`CICNumSections` — Number of sections of the CIC filter being compensated

2 (default) | positive integer scalar

Specify the number of sections of the CIC filter being compensated as a positive integer scalar. The default is 2.

`CICRateChangeFactor` — Rate-change factor of the CIC filter being compensated

2 (default) | positive integer scalar

Specify the rate-change factor of the CIC filter being compensated as a positive integer scalar. The default is 2.

`DesignForMinimumOrder` — Design filter of minimum order or of specified order

`true` (default) | logical value

Specify whether to design a filter of minimum order or a filter of specified order as a logical scalar. The default is `true`, which corresponds to a filter of minimum order.

`FilterOrder` — Order of interpolation compensator filter

12 (default) | positive integer scalar

Specify the order of the interpolation compensator filter as a positive integer scalar. This property applies only when you set the `DesignForMinimumOrder` property to `false`. The default is 12.

`InterpolationFactor` — Interpolation factor of compensator

2 (default) | positive integer scalar

Specify the interpolation factor of the compensator System object as a positive integer scalar. The default is 2.

`PassbandFrequency` — Passband edge frequency in hertz

100 kHz (default) | positive real scalar

Specify the passband edge frequency as a positive real scalar expressed in hertz. `PassbandFrequency` must be less than $F_s/2$, where F_s is the output sample rate. The default is 100 kHz.

`PassbandRipple` — Filter passband ripple in decibels

0.1 dB (default) | positive real scalar

Specify the filter passband ripple as a positive real scalar expressed in decibels. The default is 0.1 dB.

`SampleRate` — Input sample rate in hertz

600 kHz (default) | positive real scalar

Specify the input sample rate as a positive real scalar expressed in hertz. The default is 600 kHz.

`StopbandAttenuation` — Filter stopband attenuation in decibels

60 dB (default) | positive real scalar

Specify the filter stopband attenuation as a positive real scalar expressed in decibels. The default is 60 dB

`StopbandFrequency` — Stopband edge frequency in hertz

400 kHz (default) | positive real scalar

Specify the stopband edge frequency as a positive real scalar expressed in hertz. StopbandFrequency must be less than $F_s/2$, where F_s is the output sample rate. The default is 400 kHz.

Fixed-Point Properties

CoefficientsDataType — Word and fraction lengths of coefficients
numerictype(1,16) (default) | numerictype object

Word and fraction lengths of coefficients, specified as a signed or unsigned numerictype object. The default, numerictype(1,16) corresponds to a signed numeric type object with 16-bit coefficients and a fraction length determined based on the coefficient values, to give the best possible precision.

This property is not tunable.

Word length of the output is same as the word length of the input. Fraction length of the output is computed such that the entire dynamic range of the output can be represented without overflow. For details on how the fraction length of the output is computed, see "Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters".

RoundingMethod — Rounding method for output fixed-point operations
'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Rounding method for output fixed-point operations, specified as a character vector. For more information on the rounding modes, see "Precision and Range".

Methods

reset	Reset internal states of CIC compensation interpolator
step	Compensate for subsequent CIC filter

More "Analysis Methods for Filter System Objects" on page 3-2.

You can also type dsp.CICCompensationInterpolator.helpFilterAnalysis at the command line to obtain a list of multirate discrete-time filter analysis methods supported for dsp.CICCompensationInterpolator objects.

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

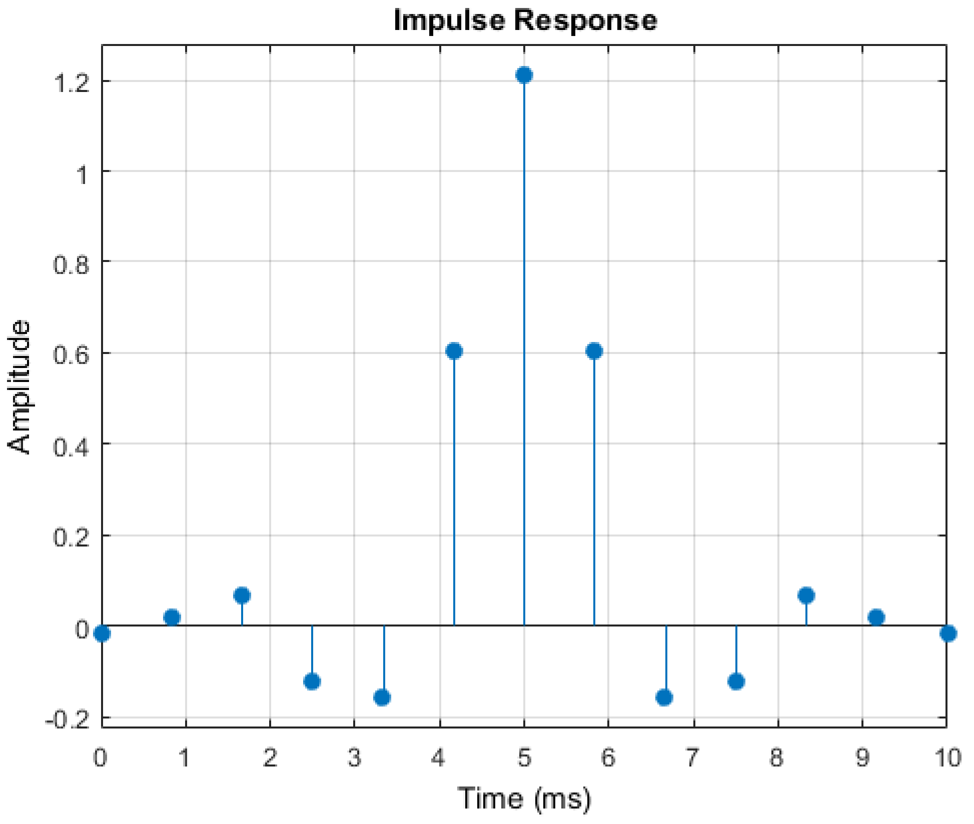
Impulse and Frequency Response of CIC Compensation Interpolator

Design an CIC compensation interpolator. Specify the interpolation factor to be 2, passband frequency to be 200 Hz, stopband frequency to be 500 Hz, and the input sample rate to be 600 Hz.

```
fs = 600;
fPass = 200;
fStop = 500;

CICCompInterp = dsp.CICCompensationInterpolator('InterpolationFactor',2,'PassbandFreque
    'StopbandFrequency',fStop,'SampleRate',fs);
```

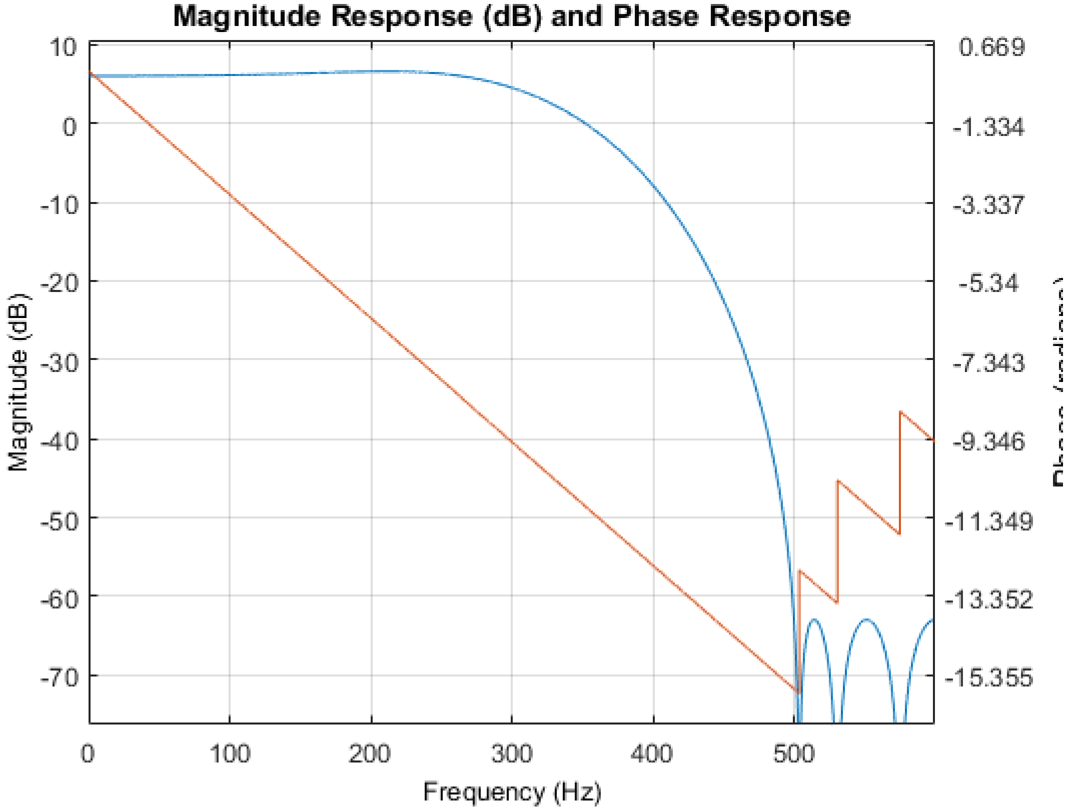
Plot the impulse response. The zeroth order coefficient is delayed 6 samples, which is equal to the group delay of the filter.

```
fvtool(CICCompInterp,'Analysis','impulse')
```



Plot the magnitude and Phase response.

```
fvtool(CICCompInterp,'Analysis','freq')
```



Compensation Interpolator Design

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Design a compensation interpolator for an existing CIC interpolator having six sections and an interpolation factor of 16.

```
CICInterp = dsp.CICInterpolator('InterpolationFactor',16, ...
    'NumSections',6);
```

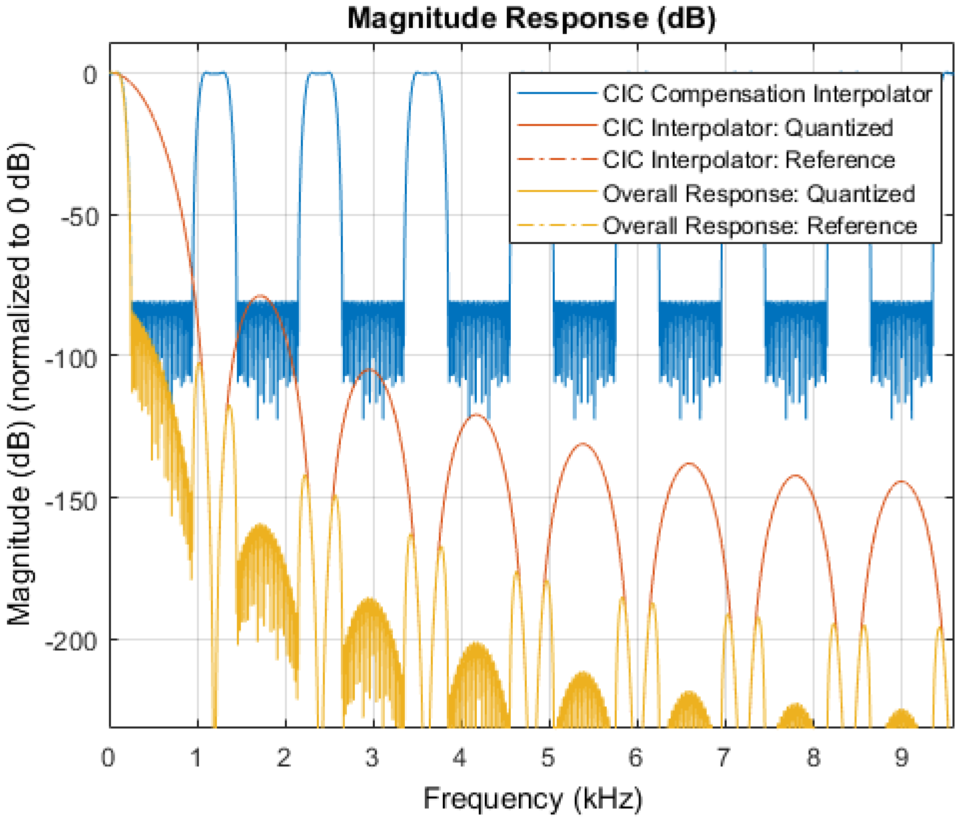
Construct the compensation interpolator. Specify an interpolation factor of 2, an input sample rate of 600 Hz, a passband frequency of 100 Hz, and a stopband frequency of 250

Hz. Set the minimum attenuation of alias components in the stopband to be at least 80 dB.

```
fs = 600;
fPass = 100;
fStop = 250;
ast = 80;

CICCompInterp = dsp.CICCompensationInterpolator(CICInterp, ...
    'InterpolationFactor',2,'PassbandFrequency',fPass, ...
    'StopbandFrequency',fStop,'StopbandAttenuation',ast, ...
    'SampleRate',fs);
```

Visualize the frequency response of the cascade. Normalize all magnitude responses to 0 dB.

```
FC = dsp.FilterCascade(CICCompInterp, CICInterp);

f = fvtool(CICCompInterp,CICInterp,FC, ...
    'Fs', [fs*2 fs*16*2 fs*16*2]);

f.NormalizeMagnitudeto1 = 'on';
legend(f,'CIC Compensation Interpolator','CIC Interpolator', ...
    'Overall Response');
```



Apply the design to a 1000-sample random input signal.

```
x = dsp.SignalSource(fi(rand(1000,1),1,16,15),'SamplesPerFrame',100);

y = fi(zeros(0,1),1,32,20);
for ind = 1:10
    x2 = CICCompInterp(x());
    y = [y;CICInterp(x2)];
```

```
end
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

dsp.CICCompensationDecimator | dsp.CICDecimator | dsp.CICInterpolator

Introduced in R2014b

reset

System object: dsp.CICCompensationInterpolator
Package: dsp

Reset internal states of CIC compensation interpolator

Syntax

```
reset(ciccompint)
```

Description

`reset(ciccompint)` resets the internal states of a `CICCompensationInterpolator` System object, `ciccompint`, to their initial values.

Input Arguments

`ciccompint` — CIC compensation interpolator
`CICCompensationInterpolator` System object

CIC compensation interpolator, specified as a `CICCompensationInterpolator` System object.

step

System object: dsp.CICCompensationInterpolator
Package: dsp

Compensate for subsequent CIC filter

Syntax

`y = step(ciccompint,x)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(ciccompint,x)` outputs the upsampled and filtered values, `y`, of the input signal, `x`. The System object treats a $K_i \times N$ input matrix N independent channels, interpolating each channel over the first dimension. The result is a $K_o \times N$ output matrix, where $K_o = K_i \times L$ and L is the interpolation factor. The object supports any real or complex floating-point or fixed-point input, with the exception of complex unsigned fixed-point data.

Input Arguments

`ciccompint` — CIC compensation interpolator

`CICCompensationInterpolator` System object

CIC compensation interpolator, specified as a `CICCompensationInterpolator` System object.

`x` — Input signal

vector | matrix

Input signal, specified as a vector or matrix.

Output Arguments

y — Upsampled and filtered signal
vector | matrix

Upsampled and filtered signal, returned as a vector or matrix.

dsp.CICDecimator System object

Package: dsp

Decimate input using Cascaded Integrator-Comb filter

Description

The `CICDecimator` object decimates inputs using a Cascaded Integrator-Comb filter. Fixed-point inputs and outputs to the object have signed fixed-point data types. To use fixed-point data types, you must have a Fixed-Point Designer license. Inputs and outputs to the System object can be single or double data types, in addition to the fixed-point data type. Property settings related to fixed-point data types are ignored for single or double data types.

Note: To use the filter analysis methods on `dsp.CICDecimator` System object, you must have the Fixed-Point Designer license. To get a list of the filter analysis methods this object supports, type `dsp.CICDecimator.helpFilterAnalysis` in the MATLAB command prompt.

To decimate inputs using a CIC filter:

1. Define and set up your System object. See “Construction” on page 4-386.
2. Call step to decimate the input according to the properties of `dsp.CICDecimator`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`cicdecim= dsp.CICDecimator` returns a `CICDecimator` System object, `cicdecim`, that you can use to decimate the input with a cascaded integrator-comb (CIC) decimation filter.

`cicdecim = dsp.CICDecimator('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a `CICDecimator` object, `cicdecim`, with each specified property set to the value you specify.

`cicdecim = dsp.CICDecimator(`*`R`*`,`*`M`*`,`*`N`*`,'`*`PropertyName`*`',` *`PropertyValue`*`, ...)` returns a `CICDecimator` object, `cicdecim`, with the `DecimationFactor` on page 4-386 property set to *R*, the `DifferentialDelay` on page 4-386 property set to *M*, the `NumSections` on page 4-386 property set to *N*, and any other specified properties set to the values you specify.

Properties

`DecimationFactor`

Decimation factor of filter

Specify a positive integer amount by which the object decimates the input. The default is `2`.

`DifferentialDelay`

Differential delay of filter comb sections

Specify a positive integer delay value for the object to use in each comb section of the filter. The default is `1`.

`NumSections`

Number of integrator and comb sections

Specify the number of integrator and comb sections in the CIC filter as a positive integer value. The default is `2`.

FixedPointDataType

Fixed-point property setting

Specify the fixed-point data type as one of | `Full precision` | `Minimum section word lengths` | `Specify word lengths` | `Specify word and fraction lengths` |. The default is `Full precision`. When you set this property to:

- `Full precision` – the `CICDecimator` object automatically determines the word and fraction lengths of the filter sections and output.
- `Minimum section word length` – the object automatically determines the word and fraction lengths of the filter sections and the fraction length of the output. You must specify the `OutputWordLength` on page 4-388.
- `Specify word lengths` – the object automatically determines the fraction lengths of the filter sections and the output. You must specify values for the `OutputWordLength` and the `SectionWordLengths` on page 4-387 properties.
- `Specify word and fraction lengths`, – the object does not automatically determine word and fraction lengths for the filter sections or the output. You must specify them using the `OutputFractionLength` on page 4-388, `OutputWordLength`, `SectionFractionLengths` on page 4-387, and `SectionWordLengths` properties.

SectionWordLengths

Word length of each filter section

Specify the fixed-point word length for each section of the CIC filter as a scalar or a vector of length `2*NumSections` on page 4-386. This property applies only when you set the `FixedPointDataType` on page 4-387 property to either `Specify word lengths` or `Specify word and fraction lengths`. The default is `[16 16 16 16]`.

SectionFractionLengths

Fraction length of each filter section

Specify the fixed-point fraction length for each section of the CIC filter as a scalar or a vector of length `2*NumSections` on page 4-386. This property applies only when you set the `FixedPointDataType` on page 4-387 property to `Specify word and fraction lengths`. The default is `0`.

OutputWordLength

Word length of filter output

Specify the fixed-point word length for the filter output. This property applies only when you set the FixedPointDataType on page 4-387 property to Minimum section word lengths, Specify word lengths, orSpecify word and fraction lengths. The default is 32.

OutputFractionLength

Fraction length of filter output

Specify the fixed-point fraction length for the output of the CIC filter. This property applies only when you set the FixedPointDataType on page 4-387 property to Specify word and fraction lengths. The default is 0.

Methods

freqz	Frequency response
fvtool	Open filter visualization tool
impz	Impulse response
phasez	Unwrapped phase response
reset	Reset filter states of CIC decimator object
step	Decimate input using Cascaded Integrator-Comb filter

More “Analysis Methods for Filter System Objects” on page 3-2.

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Decimate a Signal Using CICDecimator Object

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Decimate signal by a factor of 4 (i.e., downsample the signal from 44.1 kHz to 11.025 kHz).

```
cicdec = dsp.CICDecimator(4);
cicdec.FixedPointDataType = 'Minimum section word lengths';
cicdec.OutputWordLength = 16;
```

Create fixed-point sinusoidal input signal

```
Fs = 44.1e3;              % Original sampling frequency
n = (0:1023)';            % 1024 samples, 0.0232 sec signal
x = fi(sin(2*pi*1e3/Fs*n),true,16,15);
```

Create SignalSource System object

```
src = dsp.SignalSource(x, 64);
```

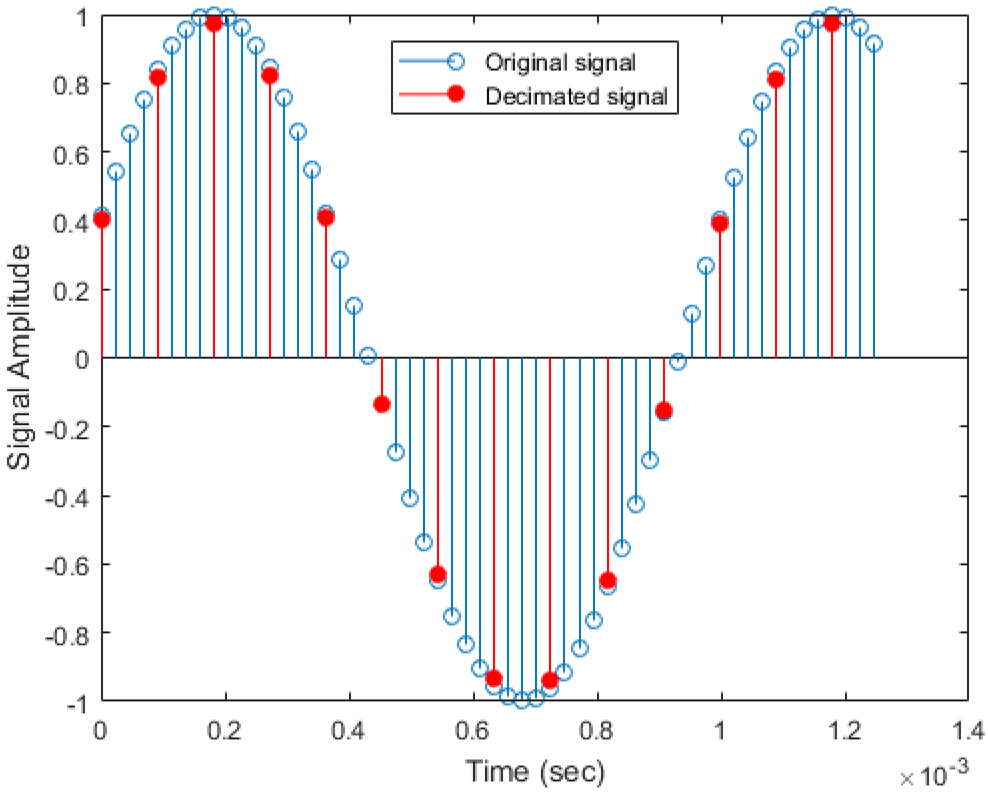
Decimate output with 16 samples per frame

```
y = zeros(16,16);
for ii=1:16
     y(ii,:) = cicdec(src());
end
```

Plot first frame of original and decimated signals. Output latency is 2 samples.

```
gainCIC = ...
   (cicdec.DecimationFactor*cicdec.DifferentialDelay)^cicdec.NumSections;
stem(n(1:56)/Fs, double(x(4:59))); hold on;
stem(n(1:14)/(Fs/cicdec.DecimationFactor),double(y(1,3:end))/gainCIC,'r','filled');
xlabel('Time (sec)');ylabel('Signal Amplitude');
legend('Original signal', 'Decimated signal', 'location', 'north');
```

```
hold off;
```



Algorithms

A Cascaded Integrator-Comb (CIC) decimation filter with comb and integrator sections is implemented when the input is fixed-point data type. When the input data type is double or single the `dsp.CICDecimator` object implements an *N*-section CIC interpolation filter as an FIR filter with a response that corresponds to a cascade of *N* boxcar filters.

This object implements the algorithm, inputs, and outputs described on the `CIC Decimation` block reference page. The object properties correspond to the block properties, except:

- The **Rate options** block parameter is not supported by the `dsp.CICDecimator` object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.FIRDecimator | dsp.CICInterpolator | dsp.CICCompensationInterpolator | dsp.CICCompensationDecimator

Introduced in R2012a

freqz

System object: dsp.CICDecimator
Package: dsp

Frequency response

Syntax

```
[h,w] = freqz(cicdecim)
[h,w] = freqz(cicdecim,n)
[h,w] = freqz(cicdecim,Name,Value)
freqz(cicdecim)
```

Description

`[h,w] = freqz(cicdecim)` returns the complex, 8192–element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample.

`[h,w] = freqz(cicdecim,n)` returns the complex, n-element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[h,w] = freqz(cicdecim,Name,Value)` returns the frequency response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`freqz(cicdecim)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the filter System object `cicdecim`.

fvtool

System object: dsp.CICDecimator
Package: dsp

Open filter visualization tool

Syntax

```
fvtool(cicdecim)
fvtool(cicdecim,'Arithmetic',ARITH,...)
```

Description

`fvtool(cicdecim)` performs an analysis and computes the magnitude response of the filter System object `cicdecim`.

`fvtool(cicdecim,'Arithmetic',ARITH,...)` analyzes the filter System object `cicdecim`, based on the arithmetic specified in the ARITH input. ARITH can be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The 'Arithmetic' input is only relevant for the analysis of filter System objects.

impz

System object: dsp.CICDecimator
Package: dsp

Impulse response

Syntax

```
[h,t] = impz(cicdecim)
[h,t] = impz(cicdecim,Name,Value)
impz(cicdecim)
```

Description

`[h,t] = impz(cicdecim)` returns the impulse response `h`, and the corresponding time points `t` at which the impulse response of`cicdecim` is computed.

`[h,t] = impz(cicdecim,Name,Value)` returns the impulse response `h`, and the corresponding time points`t` , with additional options specified by one or more Name, Value pair arguments.

`impz(cicdecim)` uses FVTool to plot the impulse response of the filter System object `cicdecim`.

Note: You can use `impz` for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

phasez

System object: dsp.CICDecimator
Package: dsp

Unwrapped phase response

Syntax

```
[phi,w] = phasez(cicdecim)
[phi,w] = phasez(cicdecim,n)
[phi,w] = phasez(cicdecim,Name,Value)
phasez(cicdecim)
```

Description

`[phi,w] = phasez(cicdecim)` returns the 8192–element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample.

`[phi,w] = phasez(cicdecim,n)` returns the n-element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[phi,w] = phasez(cicdecim,Name,Value)` returns the phase response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`phasez(cicdecim)` uses FVTool to plot the phase response of the filter System object `cicdecim`.

reset

System object: dsp.CICDecimator
Package: dsp

Reset filter states of CIC decimator object

Syntax

```
reset(cicdecim)
```

Description

`reset(cicdecim)` resets the filter states of the `CICDecimator` System object `cicdecim` to zero.

step

System object: dsp.CICDecimator
Package: dsp

Decimate input using Cascaded Integrator-Comb filter

Syntax

`Y = step(cicdecim,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(cicdecim,X)` decimates the fixed-point input `X` to produce a fixed-point output `Y` using the `CICDecimator` System object `cicdecim`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.CICInterpolator System object

Package: dsp

Interpolate signal using Cascaded Integrator-Comb filter

Description

The `CICInterpolator` object interpolates inputs using a Cascaded Integrator-Comb filter. Fixed-point inputs and outputs to the object have signed fixed-point data types. To use fixed-point data types, you must have a Fixed-Point Designer license. Inputs and outputs to the System object can be single or double data types, in addition to the fixed-point data type. Property settings related to fixed-point data types are ignored for single or double data types.

Note: To use the filter analysis methods on `dsp.CICInterpolator` System object, you must have the Fixed-Point Designer license. To get a list of the filter analysis methods this object supports, type `dsp.CICInterpolator.helpFilterAnalysis` in the MATLAB command prompt.

To interpolate inputs using a CIC filter:

1. Define and set up your System object. See "Construction" on page 4-399.
2. Call step to interpolate the input according to the properties of `dsp.CICInterpolator`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`cicinterp = dsp.CICInterpolator` returns a System object, `cicinterp`, that you can use to interpolate the input with a cascaded integrator-comb (CIC) interpolation filter.

`cicinterp = dsp.CICInterpolator('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a `CIC` interpolation object, `cicinterp`, with each specified property set to the value you specify.

`cicinterp = dsp.CICInterpolator(`*`R`*`,`*`M`*`,`*`N`*`,'`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a `CICInterpolator` object, `cicinterp`, with the `InterpolationFactor` on page 4-399 property set to *`R`*, the `DifferentialDelay` on page 4-399 property set to *`M`*, the `NumSections` on page 4-399 property set to *`N`*, and other specified properties set to the values you specify.

Properties

`InterpolationFactor`

Interpolation factor of filter

Specify a positive integer amount by which the object interpolates the input signal. The default is `2`.

`DifferentialDelay`

Differential delay of filter comb sections

Specify a positive integer delay value for the object to use in each comb section of the filter. The default is `1`.

`NumSections`

Number of integrator and comb sections

Specify the number of integrator and comb sections in the CIC filter as a positive integer value. The default is `2`.

FixedPointDataType

Fixed-point property setting

Specify the fixed-point data type as one of | `Full precision` | `Minimum section word lengths` | `Specify word lengths` | `Specify word and fraction lengths` |. The default is `Full precision`. When you set this property to:

- `Full precision` – the `CICInterpolator` object automatically determines the word and fraction lengths of the filter sections and output.
- `Minimum section word length` – the object automatically determines the word and fraction lengths of the filter sections and the fraction length of the output. You must specify the `OutputWordLength` on page 4-401.
- `Specify word lengths` – the object automatically determines the fraction lengths of the filter sections and the output. You must specify values for the `OutputWordLength` and the `SectionWordLengths` on page 4-400 properties.
- `Specify word and fraction lengths` – the object does not automatically determine word and fraction lengths for the filter sections or the output. You must specify them using the `OutputFractionLength` on page 4-401, `OutputWordLength`, `SectionFractionLengths` on page 4-400, and `SectionWordLengths` properties.

SectionWordLengths

Word length of each filter section

Specify the fixed-point word length for each section of the CIC filter as a scalar or a vector of length `2*NumSections` on page 4-386. This property applies only when you set the `FixedPointDataType` on page 4-400 property to `Specify word lengths` or `Specify word and fraction lengths`. The default is `[16 16 16 16]`.

SectionFractionLengths

Fraction length of each filter section

Specify the fixed-point fraction length for each section of the CIC filter as a scalar or a vector of length `2*NumSections` on page 4-386. This property applies only when you set the `FixedPointDataType` on page 4-400 property to `Specify word and fraction lengths`. The default is `0`.

OutputWordLength

Word length of filter output

Specify the fixed-point word length for the filter output. This property applies only when you set the `FixedPointDataType` on page 4-400 property to `Minimum section word lengths`, `Specify word lengths`, or`Specify word and fraction lengths`. The default is `32`.

OutputFractionLength

Fraction length of filter output

Specify the fixed-point fraction length for the output of the CIC filter. This property applies only when you set the `FixedPointDataType` on page 4-400 property to `Specify word and fraction lengths`. The default is `0`.

Methods

freqz	Frequency response
fvtool	Open filter visualization tool
impz	Impulse response
phasez	Unwrapped phase response
reset	Reset internal states of CIC Interpolator object
step	Interpolate signal using Cascaded Integrator-Comb filter

More “Analysis Methods for Filter System Objects” on page 3-2.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object

Common to All System Objects	
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Interpolate a Signal Using CICInterpolator Object

Interpolate signal by a factor of 2 (upsample the signal from 22.05 kHz to 44.1 kHz).

```
cicint = dsp.CICInterpolator(2);
```

Create fixed-point sinusoidal input signal

```
Fs = 22.05e3;      % Original sampling frequency
n = (0:511)';      % 512 samples, 0.0113 sec signal
x = fi(sin(2*pi*1e3/Fs*n),true,16,15);
```

Create SignalSource System object (TM).

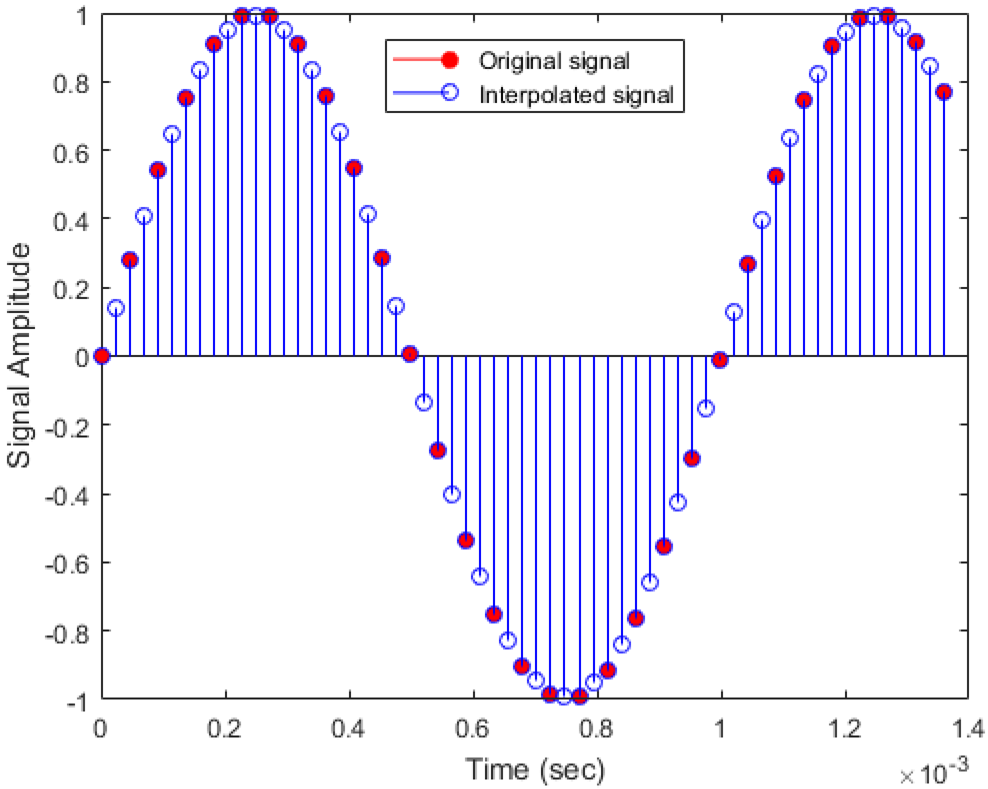
```
src = dsp.SignalSource(x, 32);
```

Interpolate output with 64 samples per frame

```
y = zeros(16,64);
for ii=1:16
     y(ii,:) = cicint(src());
end
```

Plot first frame of original and interpolated signals. Output latency is 2 samples.

```
gainCIC = ...
   (cicint.InterpolationFactor*cicint.DifferentialDelay)...
   ^cicint.NumSections/cicint.InterpolationFactor;
stem(n(1:31)/Fs, double(x(1:31)),'r','filled'); hold on;
stem(n(1:61)/(Fs*cicint.InterpolationFactor), ...
   double(y(1,4:end))/gainCIC,'b');
xlabel('Time (sec)');ylabel('Signal Amplitude');
legend('Original signal', 'Interpolated signal',...
   'location', 'north');
hold off;
```



Algorithms

A Cascaded Integrator-Comb (CIC) interpolation filter with comb and integrator sections is implemented when the input is fixed-point data type. When the input data type is double or single the `dsp.CICInterpolator` object implements an *N*-section CIC interpolation filter as an FIR filter with a response that corresponds to a cascade of *N* boxcar filters.

This object implements the algorithm, inputs, and outputs described on the `CIC Interpolation` block reference page. The object properties correspond to the block properties, except:

- The **Framing** block parameter is not supported by the `dsp.CICInterpolation` object. The object always maintains the input frame rate.
- The **Rate options** block parameter is not supported by the `dsp.CICInterpolation` object.
- The **Input processing** block parameter is not supported by the `dsp.CICInterpolation` object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

dsp.FIRInterpolator | dsp.CICDecimator | dsp.CICCompensationInterpolator | dsp.CICCompensationDecimator

Introduced in R2012a

freqz

System object: dsp.CICInterpolator
Package: dsp

Frequency response

Syntax

```
[h,w] = freqz(cicinterp)
[h,w] = freqz(cicinterp,n)
[h,w] = freqz(cicinterp,Name,Value)
freqz(cicinterp)
```

Description

`[h,w] = freqz(cicinterp)` returns the complex, 8192–element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample.

`[h,w] = freqz(cicinterp,n)` returns the complex, n-element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[h,w] = freqz(cicinterp,Name,Value)` returns the frequency response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`freqz(cicinterp)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the filter System object `cicinterp`.

fvtool

System object: dsp.CICInterpolator
Package: dsp

Open filter visualization tool

Syntax

```
fvtool(cicinterp)
fvtool(cicinterp,'Arithmetic',ARITH,...)
```

Description

`fvtool(cicinterp)` performs an analysis and computes the magnitude response of the filter System object `cicinterp`.

`fvtool(cicinterp,'Arithmetic',ARITH,...)` analyzes the filter System object `cicinterp`, based on the arithmetic specified in the ARITH input. ARITH can be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The 'Arithmetic' input is only relevant for the analysis of filter System objects.

impz

System object: dsp.CICInterpolator
Package: dsp

Impulse response

Syntax

```
[h,t] = impz(cicinterp)
[h,t] = impz(cicinterp,Name,Value)
impz(cicinterp)
```

Description

`[h,t] = impz(cicinterp)` returns the impulse response `h`, and the corresponding time points `t` at which the impulse response of`cicinterp` is computed.

`[h,t] = impz(cicinterp,Name,Value)` returns the impulse response `h`, and the corresponding time points`t` , with additional options specified by one or more Name, Value pair arguments.

`impz(cicinterp)` uses FVTool to plot the impulse response of the filter System object `cicinterp`.

Note: You can use `impz` for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

---

phasez

System object: dsp.CICInterpolator
Package: dsp

Unwrapped phase response

Syntax

```
[phi,w] = phasez(cicinterp)
[phi,w] = phasez(cicinterp,n)
[phi,w] = phasez(cicinterp,Name,Value)
phasez(cicinterp)
```

Description

`[phi,w] = phasez(cicinterp)` returns the 8192–element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample.

`[phi,w] = phasez(cicinterp,n)` returns the n-element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[phi,w] = phasez(cicinterp,Name,Value)` returns the phase response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`phasez(cicinterp)` uses FVTool to plot the phase response of the filter System object `cicinterp`.

reset

System object: dsp.CICInterpolator
Package: dsp

Reset internal states of CIC Interpolator object

Syntax

`reset(cicinterp)`

Description

`reset(cicinterp)` resets the internal states of the `CICInterpolator` System object `cicinterp` to zero.

step

System object: dsp.CICInterpolator
Package: dsp

Interpolate signal using Cascaded Integrator-Comb filter

Syntax

```
Y = step(cicinterp,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(cicinterp,X)` interpolates the fixed-point input `X` to produce a fixed-point output `Y` using the `CICInterpolator` System object `cicinterp`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.ColoredNoise System object

Package: dsp

Colored noise generator

Description

The `ColoredNoise` object generates pink noise and other colored noise signals. The form of the PSD is $1/|f|^{\alpha}$ with the exponent, α, a real number in the interval [-2,2].

To generate a colored noise signal:

1. Define and set up your colored noise generator. See "Construction" on page 4-411.
2. Call step to generate the colored noise signal according to the properties of `dsp.ColoredNoise`. The signal is an array with the number of rows given by the `SamplesPerFrame` property. The number of columns is given by the `NumChannels` property.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`cn = dsp.ColoredNoise` returns a colored noise generator, `cn`, with default settings. Calling step with the default property settings generates a pink noise signal.

`cn = dsp.ColoredNoise('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a colored noise generator, `cn`, with each specified property set to the specified value.

`cn = dsp.ColoredNoise(POW,SAMP,CHAN,'`*`PropertyName`*`',`*`PropertyValue`*`)` returns a colored noise generator, `cn`, with `InverseFrequencyPower` equal to `POW`, `SamplesPerFrame` equal to `SAMP`, and `NumChannels` equal to `CHAN`. Other specified properties are set to the specified values.

Properties

`Color`

Noise color

Specify the color of the noise as one of `'pink'` | `'white'` | `'brown'` | `'blue'` | `'purple'` | `'custom'`. Each color is associated with an inverse frequency power of the generated noise sequence.

- For pink noise, set `Color` to `'pink'`. The inverse frequency power of pink noise is equal to `1`.
- For white noise, set `Color` to `'white'`. The inverse frequency power of white noise is equal to `0`.
- For brown noise (also known as red or Brownian noise), set `Color` to `'brown'`. The inverse frequency power of brown noise is equal to `2`.
- For blue noise (also known as azure noise), set `Color` to `'blue'`. The inverse frequency power of blue noise is equal to `-1`.
- For purple noise (also known as violet noise), set `Color` to `'purple'`. The inverse frequency power of purple noise is equal to `-2`.
- For noise with a custom inverse frequency power, set `Color` to `'custom'`. In this case, the inverse frequency power is equal to the value specified in `InverseFrequencyPower`. This is the default value.

  `InverseFrequencyPower` can be any value in the interval [`-2,2`]. `InverseFrequencyPower` is applicable only when `Color` is set to `'custom'`.

`InverseFrequencyPower`

Inverse PSD exponent

Specify the inverse PSD exponent, a, as a real number in the interval `[-2 2]`. The inverse exponent defines the PSD of the random process by $1/|f|^{\alpha}$. The default value of this property is `1`. Values of `InverseFrequencyPower` greater than 0 generate lowpass noise with a singularity (pole) at f=0. These processes exhibit long memory. Values of `InverseFrequencyPower` less than 0 generate highpass noise with increments that are negatively correlated. These processes are referred to as anti-persistent. Special cases include:

- An `InverseFrequencyPower` value of 1 generates a pink noise. This is the default value.
- An `InverseFrequencyPower` value of 2 generates a Brownian process.
- An `InverseFrequencyPower` value of 0 generates a white noise process with a flat PSD.
- An `InverseFrequencyPower` value of -1 generates a blue noise process.
- An `InverseFrequencyPower` value of -2 generates a violet (purple) noise process.

This property is applicable only when `Color` is set to `'custom'`.

In a log-log plot of power as a function of frequency, processes generated by `dsp.ColoredNoise` exhibit an approximate linear relationship with slope equal to -α.

NumChannels

Number of output channels

Specify the number of output channels as a positive integer. This property determines the number of columns of the signal. The default value of this property is 1.

SamplesPerFrame

Samples per frame

Specify the number of samples per frame as a positive integer. This property determines the number of rows of the signal. The default value of this property is 1024.

RandomStream

Source of random number stream

Specify the source of the random number stream as `'Global Stream'` or `'mt19937ar with seed'`. If you set `RandomStream` to `'Global Stream'`, the current random number generator settings are used. The current settings of the random number generator are returned by `rng`. The default value of this property is `'Global Stream'`.

Seed

Initial seed

Specify the initial seed of the mt19937ar random number generator as a nonnegative integer. This property only applies when `RandomStream` is `'mt19937ar with seed'`. The default value of this property is 67.

`OutputDataType`

Output data type

Specify the output data type as `'double'` or `'single'`. The default value of this property is `'double'`.

Methods

reset	Reset random number generator seed
step	Generate colored noise signal

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Measure Pink Noise Power in Octave Bands

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject() becomes step(myObject).

This example shows how pink noise has approximately equal power in octave bands.

Generate a single-channel signal of pink noise that is 44100 samples in length. Set the random number generator to the default settings for reproducible results.

```
pinkNoise = dsp.ColoredNoise(1,44.1e3,1);
```

```
rng default;
x = pinkNoise();
```

Assume the sampling frequency is 44.1 kHz. Measure the power in octave bands beginning with 100-200 Hz and ending with 6.400-12.8 kHz. Display the results in a table.

```
beginfreq = 100;
endfreq = 200;
count = 1;
while(endfreq<=44.1e3/2)
    freqinterval(count,:) = [beginfreq endfreq];
    Pwr(count) = bandpower(x,44.1e3,[beginfreq endfreq]);
    beginfreq = endfreq;
    endfreq = 2*endfreq;
    count = count+1;
end
Pwr = Pwr(:);
table(freqinterval,Pwr)
```

```
ans =

  7×2 table

    freqinterval       Pwr
    _____________    _______

     100      200    0.19436
     200      400    0.18472
     400      800    0.20873
     800     1600     0.2177
    1600     3200    0.21887
    3200     6400    0.23617
    6400    12800    0.23526
```

The pink noise has roughly equal power in octave bands. Repeat the preceding example with `'InverseFrequencyPower'` equal to 0, which generates a white noise signal. A white noise signal has a flat power spectral density, or equal power per unit frequency. Set the random number generator to the default settings for reproducible results.

```
whiteNoise = dsp.ColoredNoise(0,44.1e3,1);
rng default;
x = whiteNoise();
```

Assume the sampling frequency is 44.1 kHz. Measure the power in octave bands beginning with 100-200 Hz and ending with 6.400-12.8 kHz. Display the results in a table.

```
beginfreq = 100;
endfreq = 200;
count = 1;
while(endfreq<=44.1e3/2)
    freqinterval(count,:) = [beginfreq endfreq];
    Pwr(count) = bandpower(x,44.1e3,[beginfreq endfreq]);
    beginfreq = endfreq;
    endfreq = 2*endfreq;
    count = count+1;
end
Pwr = Pwr(:);
table(freqinterval,Pwr)
```

```
ans =

  7×2 table

    freqinterval        Pwr
    _____________    _________

     100      200    0.0031417
     200      400    0.0073833
     400      800     0.017421
     800     1600     0.035926
    1600     3200     0.071139
    3200     6400      0.15183
    6400    12800      0.28611
```

White noise has approximately equal power per unit frequency. Therefore, you expect octave bands to have an unequal distribution of power. Because the width of an octave band increases with increasing frequency, the power per octave band increases for a white noise.

PSD of Pink Noise Realization

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject() becomes step(myObject).

Generate a pink noise signal 2048 samples in length. Assume a sampling rate of 1 Hz. Obtain an estimate of the power spectral density using Welch's overlapped segment averaging.

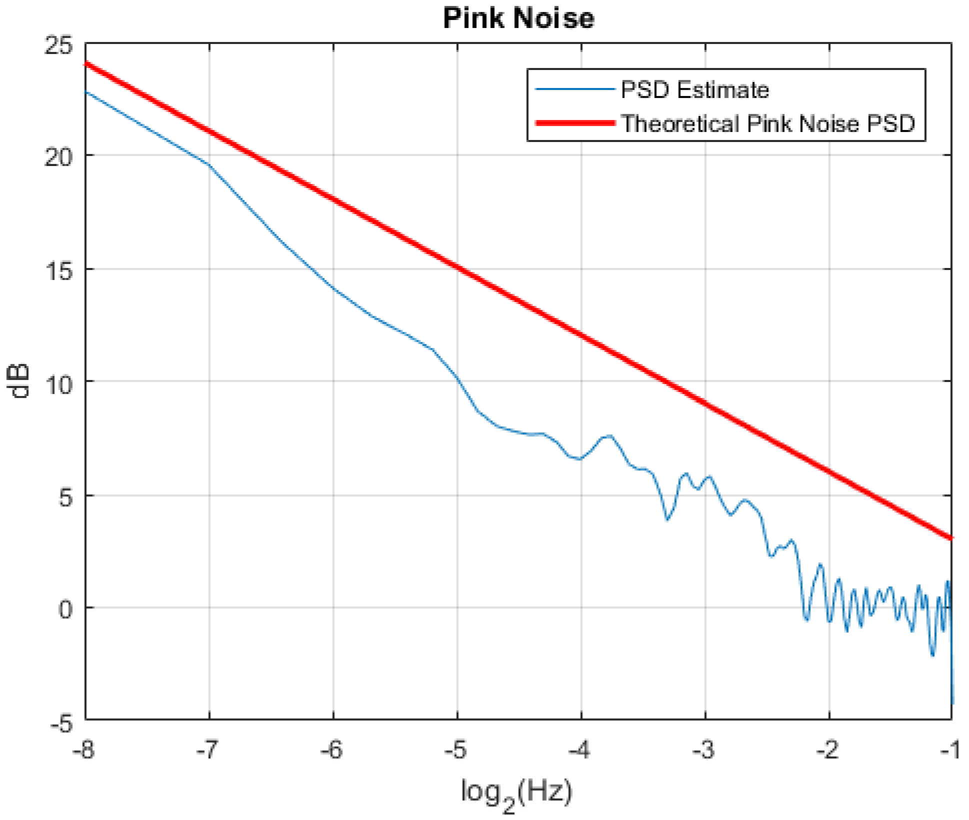
```
cn = dsp.ColoredNoise('Color','pink','SamplesPerFrame',2048);
x = cn();
Fs = 1;
[Pxx,F] = pwelch(x,hamming(128),[],[],Fs,'psd');
```

Construct the theoretical PSD of the pink noise process.

```
PSDPink = 1./F(2:end);
```

Display the Welch PSD estimate of the noise along with the theoretical PSD on a log-log plot. Plot the frequency axis as the logarithm to the base 2 to clearly show the octaves. Plot the PSD estimate in dB, $10\log_{10}$.

```
plot(log2(F(2:end)),10*log10(Pxx(2:end)));
hold on;
plot(log2(F(2:end)),10*log10(PSDPink),'r','linewidth',2)
xlabel('log_2(Hz)'); ylabel('dB');
title('Pink Noise');
grid on;
legend('PSD Estimate','Theoretical Pink Noise PSD');
```



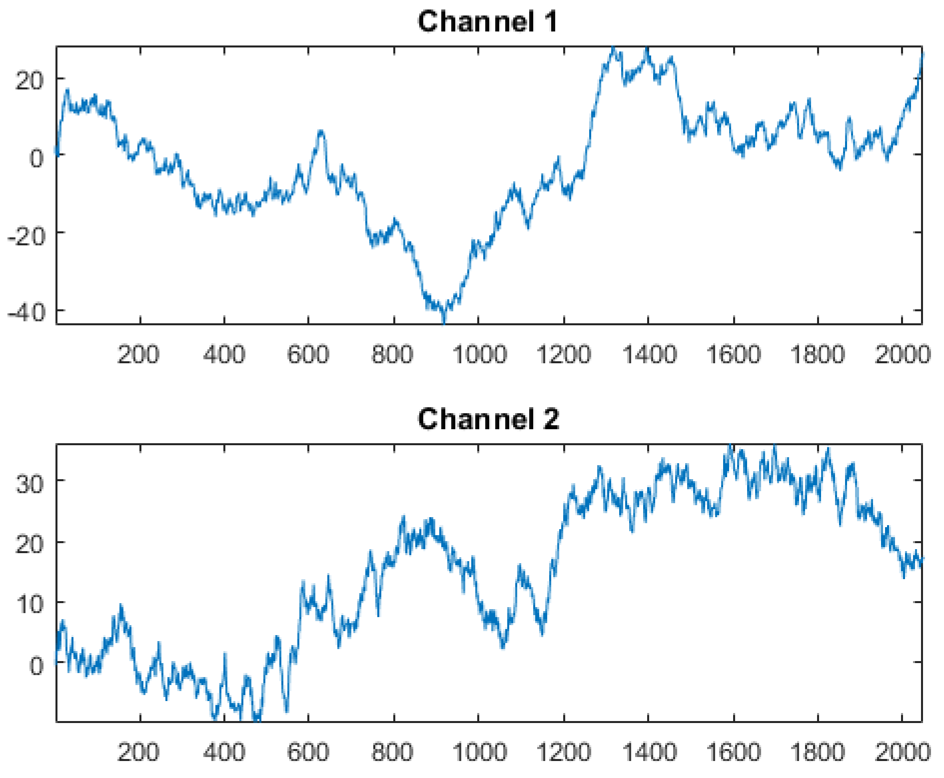
Two-Channel Brownian Noise

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject() becomes step(myObject).

Generate two channels of Brownian noise by setting `Color` to `brown` and `NumChannels` to 2.

```
cn = dsp.ColoredNoise('Color','brown','SamplesPerFrame',2048,...
    'NumChannels',2);
x = cn();
subplot(2,1,1)
plot(x(:,1)); title('Channel 1'); axis tight;
```

```
subplot(2,1,2)
plot(x(:,2)); title('Channel 2'); axis tight;
```

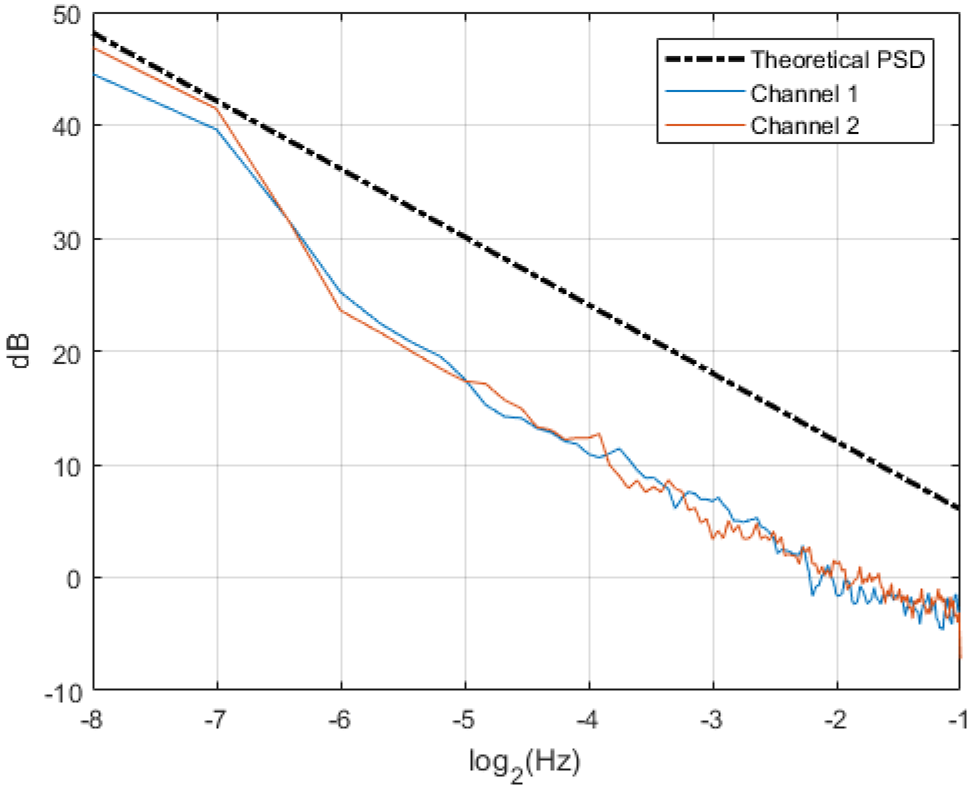


Assume a sampling frequency of 1 Hz. Obtain Welch PSD estimates for both channels.

```
Fs = 1;
for nn = 1:size(x,2)
[Pxx(:,nn),F] = pwelch(x(:,nn),hamming(128),[],[],Fs,'psd');
end
```

Construct the theoretical PSD of a Brownian process. Plot the theoretical PSD along with both realizations on a log-log plot. Use the logarithm to the base 2 for the frequency axis and plot the power spectral densities in dB.

```
PSDBrownian = 1./F(2:end).^2;
figure;
plot(log2(F(2:end)),10*log10(PSDBrownian),'k-.','linewidth',2);
hold on;
plot(log2(F(2:end)),10*log10(Pxx(2:end,:)));
xlabel('log_2(Hz)'); ylabel('dB');
grid on;
legend('Theoretical PSD','Channel 1', 'Channel 2');
```



Add Pink Noise at 0 dB SNR

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For `dsp.ColoredNoise` System object™, myObject() becomes step(myObject). For all other objects, myObject(x) becomes step(myObject,x).

This example shows how to stream in an audio file and add pink noise at a 0 dB SNR. The example reads in frames of an audio file 1024 samples in length, measures the RMS value of the audio frame, and adds pink noise with the same RMS value as the audio frame.

Set up the System objects. Set the number of `'SamplesPerFrame'` for both the file reader and the colored noise generator to 1024 samples. Set `Color` to `pink` to generate pink noise with a $1/|f|$ power spectral density.

```
N = 1024;
afr = dsp.AudioFileReader('Filename','speech_dft.mp3','SamplesPerFrame',N);
adw = audioDeviceWriter('SampleRate',afr.SampleRate);
cn = dsp.ColoredNoise('Color','pink','SamplesPerFrame',N);
rms = dsp.RMS;
```

Stream the audio file in 1024 samples at a time. Measure the signal RMS value for each frame, generate a frame of pink noise equal in length, and scale the RMS value of the pink noise to match the signal. Add the scaled noise to the signal and play the output.

```
while ~isDone(afr)
    audio = afr();
    speechRMS = rms(audio);
    noise = cn();
    noiseRMS = rms(noise);
    noise = noise*(speechRMS/noiseRMS);
    sigPlusNoise = audio+noise;
    adw(sigPlusNoise);
end

release(afr);
release(adw);
```

Averaged Power Spectrum of Pink Noise

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For `dsp.ColoredNoise` System object™, myObject() becomes step(myObject). For all other objects, myObject(x) becomes step(myObject,x).

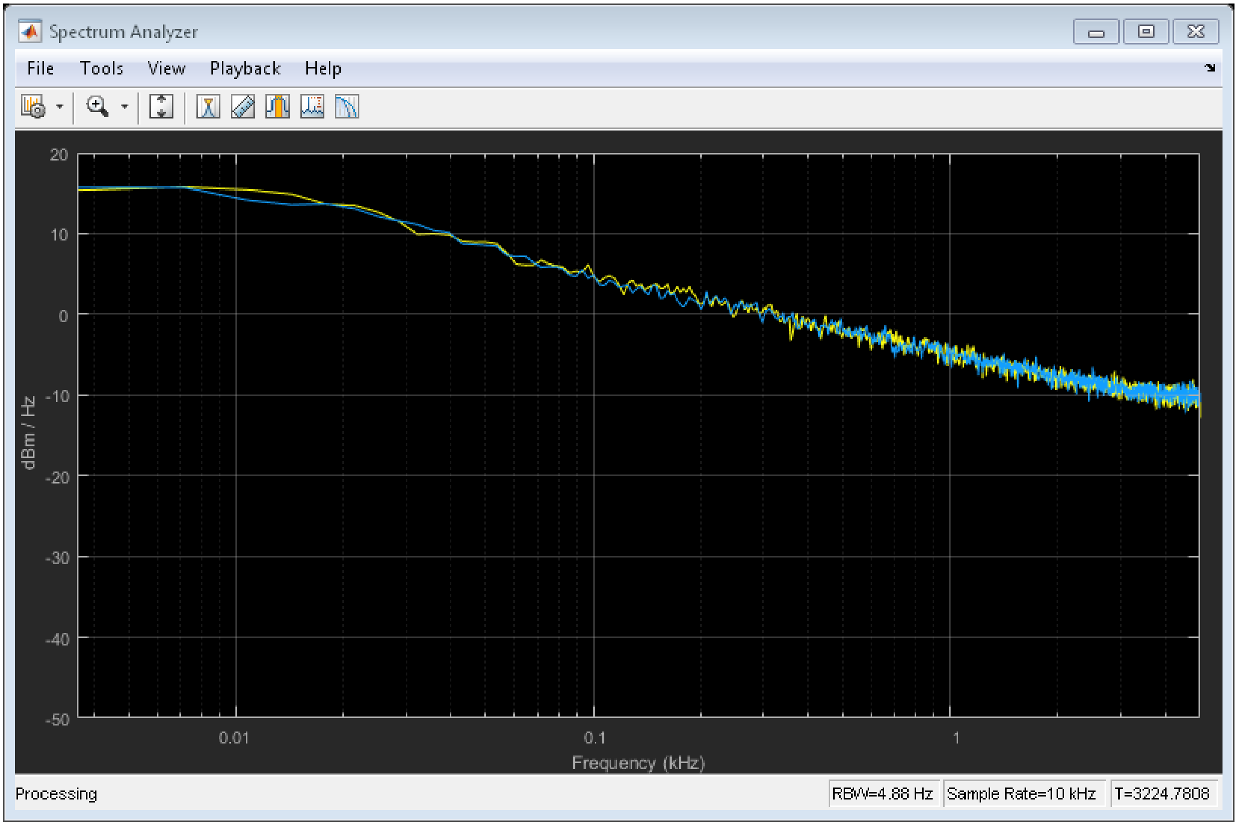
This example shows how to generate two-channels of pink noise and compute the power spectrum based on a running average of 50 PSD estimates.

Set up the colored noise generator to generate two-channels of pink noise with 1024 samples. Set up the spectrum analyzer to compute modified periodograms using a Hamming window and 50% overlap. Obtain a running average of the PSD using 50 spectral averages.

```
pinkNoise = dsp.ColoredNoise(1,1024,2);
sa = dsp.SpectrumAnalyzer('SpectrumType','Power density', ...
    'OverlapPercent',50,'Window','Hamming', ...
    'SpectralAverages',50,'PlotAsTwoSidedSpectrum',false, ...
    'FrequencyScale','log','YLimits',[-50 20]);
```

Run the simulation for 30 seconds.

```
tic
while toc < 30
    pink = pinkNoise();
    sa(pink);
end
```



Algorithms

AR Generation Method

`dsp.ColoredNoise` generates colored noise using a white noise input in an autoregressive model (AR) of order 63.

$$\sum_{k=0}^{63} a_k y(n-k) = w(n)$$

where a_0=1 and w(n) is a zero-mean white noise process.

The AR coefficients for k≥1 are generated according to the following recursive formula with α the inverse PSD exponent

$$a_k = (k - 1 - \tfrac{\alpha}{2})\frac{a_{k-1}}{k} \quad k = 1, 2, \ldots$$

The AR method used in `dsp.ColoredNoise` is detailed on pp. 820–822 in [2].

Definitions

Colored Noise Processes

Many phenomena in diverse fields such as hydrology and finance produce time series with power spectral density (PSD) functions that follow a power law of the form

$$S(f) = \frac{L(f)}{|f|^{\alpha}}$$

where α is a real number in the interval [-2,2] and $L(f)$ is a positive slowly-varying or constant function. Plotting the power spectral density of such processes on a log-log plot displays an approximate linear relationship between the log frequency and log PSD with slope equal to -α

$$\ln S(f) = -\alpha \ln|f| + \ln L(f).$$

It is often convenient to plot the PSD in dB as function of the log frequency to base 2 in order to characterize the slope in dB/octave. Rewriting the preceding equation, you obtain

$$10 \log S(f) = -10\alpha \frac{\ln(2)\log_2(f)}{\ln(10)} + 10\frac{\ln(L(f))}{\ln(10)}$$

with the slope in dB/octave given by

$$-10\alpha \frac{\ln(2)\log_2(f)}{\ln(10)}$$

If α>0, *S(f)* goes to infinity as the frequency, *f*, approaches 0. Stochastic processes with power spectral densities of this form exhibit long memory. Long-memory processes have autocorrelations that persist for a long time as opposed to decaying exponentially like many common time-series models. If α<0, the process is antipersistent and exhibits negative correlation between increments [1].

Special examples of $\frac{1}{|f|^{\alpha}}$ processes include:

- α=0 — white noise where *L(f)* is a constant proportional to the process variance.
- α=1 — pink, or flicker noise. Pink noise has equal energy per octave. See "Measure Pink Noise Power in Octave Bands" on page 4-414 for a demonstration. Accordingly, the power spectral density of pink noise decreases 3 dB per octave.
- α=2 — brown noise, or Brownian motion. Brownian motion is a nonstationary process with stationary increments. You can think of Brownian motion as the integral of a white noise process. Even though Brownian motion is nonstationary, you can still define a generalized power spectrum, which behaves like $\frac{1}{|f|^{2}}$. Accordingly, power in a brown noise decreases 6 dB per octave.
- α= -1 — blue noise. The power spectral density of blue noise increases 3 dB per octave.
- α= -2 — violet, or purple noise. The power spectral density of violet noise increases 6 dB per octave. You can think of violet noise as the derivative of white noise process.

References

[1] Beran, J., Feng, Y., Ghosh, S., and Kulik, R. *Long-Memory Processes: Probabilistic Properties and Statistical Methods*, Springer, 2013.

[2] Kasdin, N.J. "Discrete Simulation of Colored Noise and Stochastic Processes and $1/f^{\alpha}$ Power Law Noise Generation", *Proceedings of the IEEE*, Vol. 83, No. 5, 1995, pp. 802-827.

See Also

`randn` | `Colored Noise`

Introduced in R2014a

reset

System object: dsp.ColoredNoise
Package: dsp

Reset random number generator seed

Syntax

```
reset(cn)
```

Description

`reset(cn)` resets the random number generator stream if the `RandomStream` property is set to `'mt19937ar with seed'`. The seed of the random number generator is the value of the `Seed` property.

step

System object: dsp.ColoredNoise
Package: dsp

Generate colored noise signal

Syntax

```
x = step(cn)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`x = step(cn)` generates a colored noise signal for the number of channels specified by the `NumChannels` property. If the value of `NumChannels` is 1, `x` is a column vector. If the value of `NumChannels` is M >1, `x` is a matrix with M columns. The number of samples (rows) in `x` is equal to the value of `SamplesPerFrame` .

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

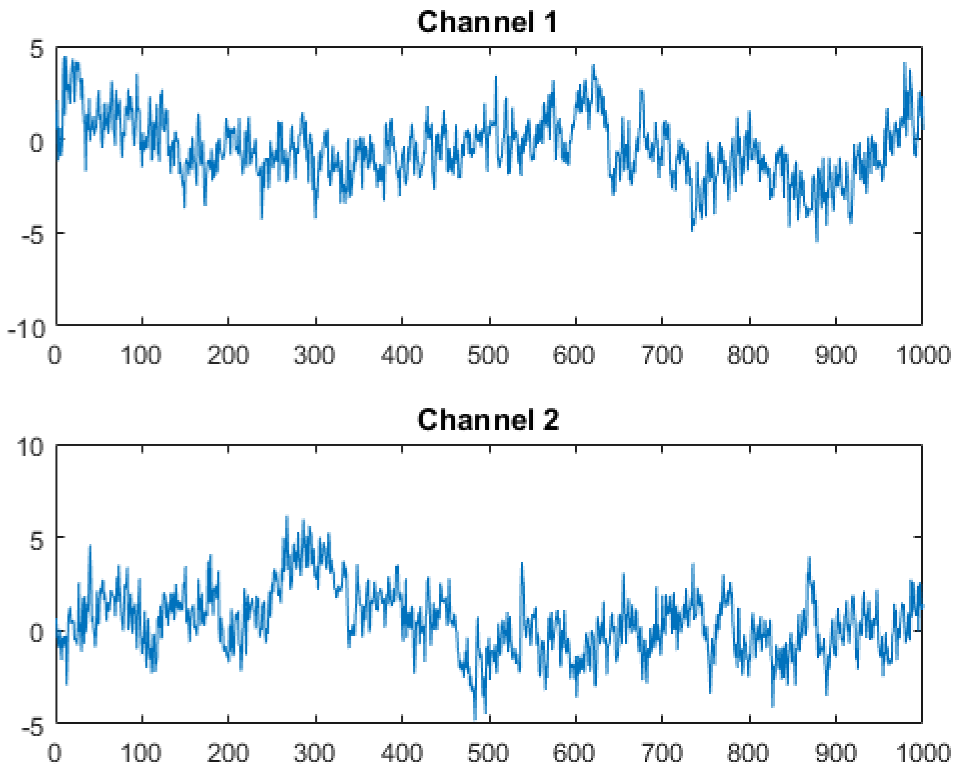
Examples

Two Channels of Pink Noise

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For `dsp.ColoredNoise` System object™, myObject() becomes step(myObject). For all other objects, myObject(x) becomes step(myObject,x).

Generate two channels of pink noise 1000 samples in length.

```
NumChannels = 2;
NumSamples = 1000;
cn = dsp.ColoredNoise('InverseFrequencyPower',1,'NumChannels',2,...
    'SamplesPerFrame',NumSamples);
x = cn();
subplot(2,1,1)
plot(x(:,1)); title('Channel 1');
subplot(2,1,2)
plot(x(:,2)); title('Channel 2');
```


Channel 1
5
0
-5
-10
0
100
200
300
400
500
600
700
800
900
1000
Channel 2
10
5
0
-5
0
100
200
300
400
500
600
700
800
900
1000

dsp.Convolver System object

Package: dsp

Convolution of two inputs

Description

The `Convolver` computes the convolution of two inputs.

To compute the convolution of two inputs:

1 Define and set up your convolver. See “Construction” on page 4-431.
2 Call step to compute the convolution according to the properties of `dsp.Convolver`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`conv = dsp.Convolver` returns a convolver object, `conv`, that convolves two inputs. For N-D arrays, the convolver computes the convolution column-wise. For arrays, the inputs must have an equal number of columns. If one input is a vector and the other is an N-D array, the convolver computes the convolution of the vector with each column of the N-D array. Convolving inputs of length N and M results in a sequence of length N+M–1. Convolving matrices of size M-by-N and P-by-N results in a matrix of size (M+P–1)-by-N.

`conv = dsp.Convolver('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a convolver object, `conv`, with each property set to the specified value.

Properties

`Method`

Domain for computing convolutions

Specify the domain in which the convolver performs the convolutions as `Time Domain`, `Frequency Domain`, or `Fastest`. Computing convolutions in the time domain minimizes memory use. Computing convolutions in the frequency domain may require fewer computations depending on the input length. If the value of this property is `Fastest`, the object computes convolutions in the domain which minimizes the number of computations. The default is `Time Domain`.

Fixed-Point Properties

`FullPrecisionOverride`

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see "Full Precision for Fixed-Point System Objects".

`RoundingMethod`

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`. This property applies only if the object is not in full precision mode.

`OverflowAction`

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`. This property applies only if the object is not in full precision mode.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as `Full precision`, `Same as first input`, or `Custom`. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `ProductDataType` on page 4-433 property is `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Full precision`, `Same as product`, `Same as first input`, or `Custom`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `AccumulatorDataType` on page 4-433 property is `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as `Same as accumulator`, `Same as product`, `Same as first input`, or `Custom`. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `OutputDataType` on page 4-433 property is `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Convolution of inputs

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

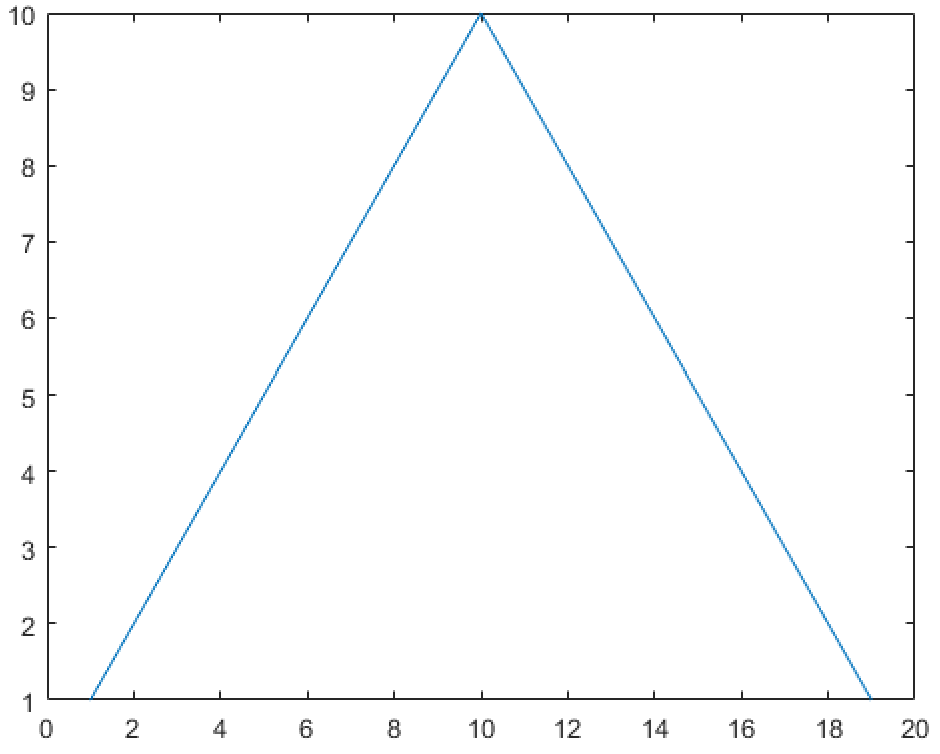
Convolve Two Rectangular Sequences

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
conv = dsp.Convolver;
x = ones(10,1);
y = conv(x, x);
```

Result is a triangular sequence

```
plot(y);
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Convolution` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.Crosscorrelator | dsp.Autocorrelator

Introduced in R2012a

step

System object: dsp.Convolver
Package: dsp

Convolution of inputs

Syntax

```
Y = step(conv,A,B)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(conv,A,B)` convolves the inputs `A` and `B`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.DCBlocker System object

Package: dsp

Remove DC component

Description

The `DCBlocker` object filters or blocks the DC component of an incoming signal.

To filter the DC component of a signal:

1. Define and set up your DC blocker object. See “Construction” on page 4-438.
2. Call step to filter the DC component of a signal according to the properties of `dsp.DCBlocker`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`dcblker = dsp.DCBlocker` creates a DC blocker System object, `dcblker`, that removes the DC component of each channel, i.e., column, of an input signal.

`dcblker = dsp.DCBlocker(Name,Value)` creates a DC blocker object, `dcblker`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`Algorithm`

DC offset algorithm type

Specify the DC offset estimating algorithm as one of `IIR | FIR | CIC | Subtract mean`. You can visualize the IIR, FIR, and CIC responses using the `fvtool` method. The default is `IIR`.

- `IIR` uses a recursive estimate based on a narrow, lowpass elliptic filter. You specify the filter order using the `Order` property and you set the bandwidth using the `NormalizedBandwidth` property. This algorithm typically uses less memory than FIR and is more efficient.
- `FIR` uses a non-recursive, moving-average estimate based on a finite number of past input samples that is set using the `Length` property. This algorithm typically uses more memory than IIR and is less efficient.
- `CIC` uses a lowpass filter that does not employ any multipliers. You specify the bandwidth of the filter using the `NormalizedBandwidth` property. If the `Algorithm` property is `CIC`, then fixed-point data must be input to the `step` function.
- `Subtract mean` computes the means of the columns of the input matrix and subtracts the means from the input. This method does not retain state between inputs.

NormalizedBandwidth

Normalized bandwidth of the lowpass IIR elliptic filter or the lowpass CIC filter

Specify the normalized bandwidth of the IIR or CIC filter used to estimate the DC component of the input signal as a real scalar greater than 0 and less than 1. This property applies when the `Algorithm` property is set to `IIR` or `CIC`. The default value is `0.001`.

Order

Order of the lowpass IIR elliptic filter

Specify the order of the IIR elliptic filter used to estimate the DC level. This property applies when the `Algorithm` property is set to `IIR`. Use an integer greater than 3. The default value is `6`.

Length

Number of past input samples for the FIR algorithm

Specify the number of past inputs used to estimate the running mean. This property applies when the `Algorithm` property is set to `FIR`. Use a positive integer. The default value is `50`.

Methods

fvtool	Show the frequency response of the filter used by the `DCBlocker` System object
reset	Reset states of the `DCBlocker` System object
step	Blocks DC components of input signal

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Remove DC Component and Display Results

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

This example shows how to use the DC Blocker System object to remove an input signal's DC component using three of the estimation algorithms.

Create a signal composed of a 15 Hz tone, a 25 Hz tone, and a DC offset.

```
t = (0:0.001:100)';
x = sin(30*pi*t) + 0.33*cos(50*pi*t) + 1;
```

Create three DC Blocker objects for the `IIR`, `FIR`, and `Subtract mean` estimation algorithms.

```
dc1 = dsp.DCBlocker('Algorithm','IIR','Order', 6);
```

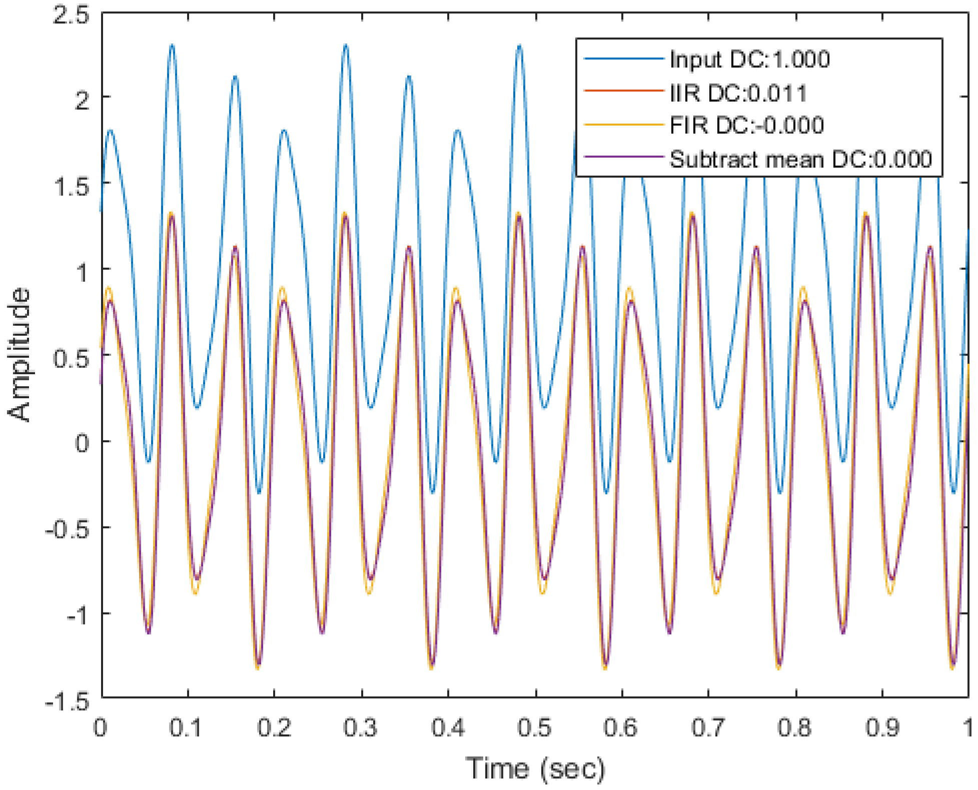
```
dc2 = dsp.DCBlocker('Algorithm','FIR','Length', 100);
dc3 = dsp.DCBlocker('Algorithm','Subtract mean');
```

For each second of time, pass the input signal through the DC blockers. By implementing the DC blockers in 1 second increments, you can observe differences in the convergence times.

```
for idx = 1 : 100
    range = (1:1000) + 1000*(idx-1);
    y1 = dc1(x(range));          % IIR estimate
    y2 = dc2(x(range));          % FIR estimate
    y3 = dc3(x(range));          % Subtract mean
end
```

Plot the input and output data for the three DC blockers for the first second of time and show the mean value for each signal. Looking at the mean values for the three algorithm types, you can see that the `FIR` and `Subtract mean` algorithms converge more quickly.

```
plot(t(1:1000),x(1:1000),...
    t(1:1000),y1, ...
    t(1:1000),y2, ...
    t(1:1000),y3);
xlabel('Time (sec)')
ylabel('Amplitude')
legend(sprintf('Input DC:%.3f',mean(x)), ...
    sprintf('IIR DC:%.3f',mean(y1)), ...
    sprintf('FIR DC:%.3f',mean(y2)), ...
    sprintf('Subtract mean DC:%.3f',mean(y3)));
```



Frequency Response Before and After DC Blocker

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

This example shows a comparison of the spectrum of an input signal with a DC offset to the spectrum of the same signal after application of a DC blocker using the `FIR` estimation algorithm.

Create an input signal composed of three tones and a DC offset of 1. Set the sampling frequency to 1 kHz and set the signal duration to 100 seconds.

```
fs = 1000;
```

```
t = (0:1/fs:100)';
x = sin(30*pi*t) + 0.67*sin(40*pi*t) + 0.33*sin(50*pi*t) + 1;
```

Create a DC blocker object with the `Algorithm` property set to `FIR`.

```
dcblk = dsp.DCBlocker('Algorithm','FIR','Length',100);
```

Create a `SpectrumAnalyzer` System object with power units set to `dBW` and a frequency range of `[-30 30]` to display the frequency response of the input signal. Using the `clone` method, create a second spectrum analyzer to display the response of the output. Then, use the `Name` property to label the two objects.

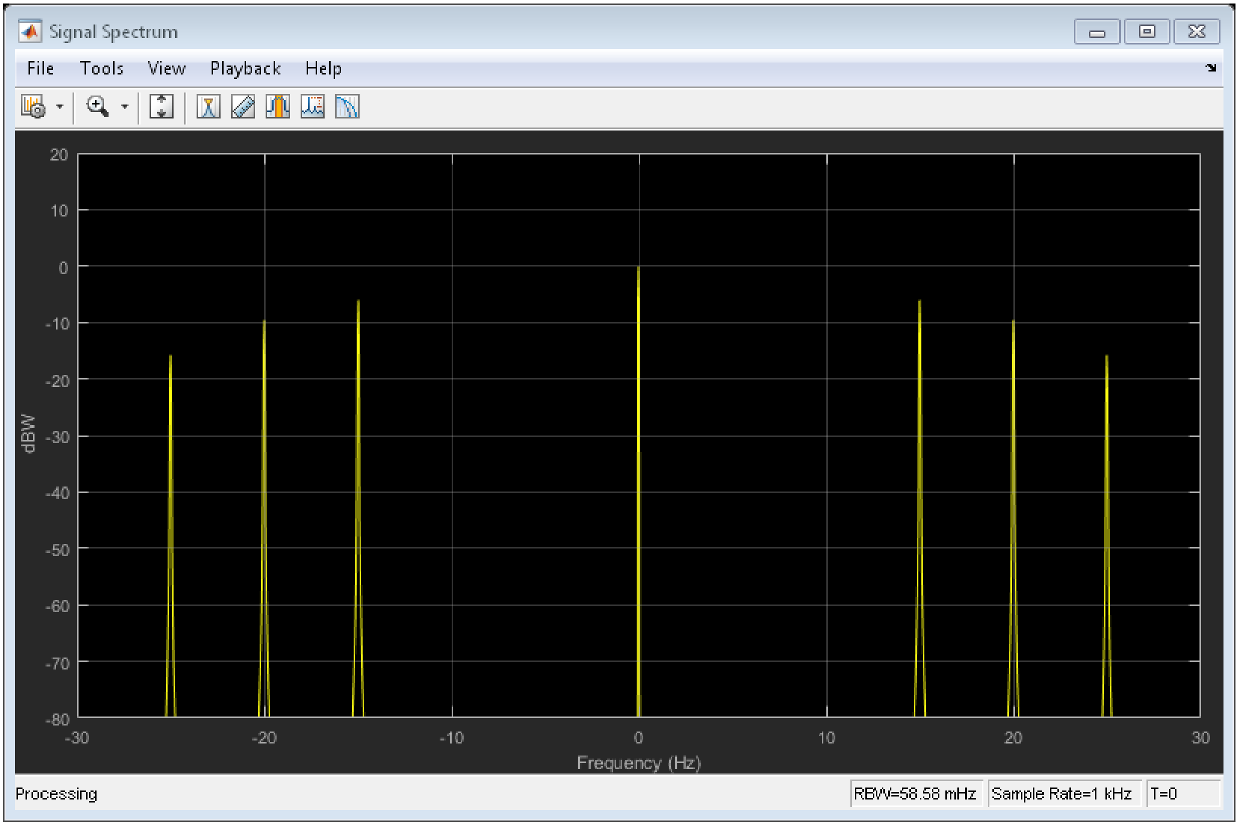
```
hsa = dsp.SpectrumAnalyzer('SampleRate',fs, ...
    'PowerUnits','dBW','FrequencySpan','Start and stop frequencies',...
    'StartFrequency',-30,'StopFrequency',30);

hsb = clone(hsa);

hsa.Name = 'Signal Spectrum';
hsb.Name = 'Signal Spectrum after DC Blocker';
```

Pass the input signal, `x`, through the DC blocker to generate the output signal, `y`.

```
y = dcblk(x);
```

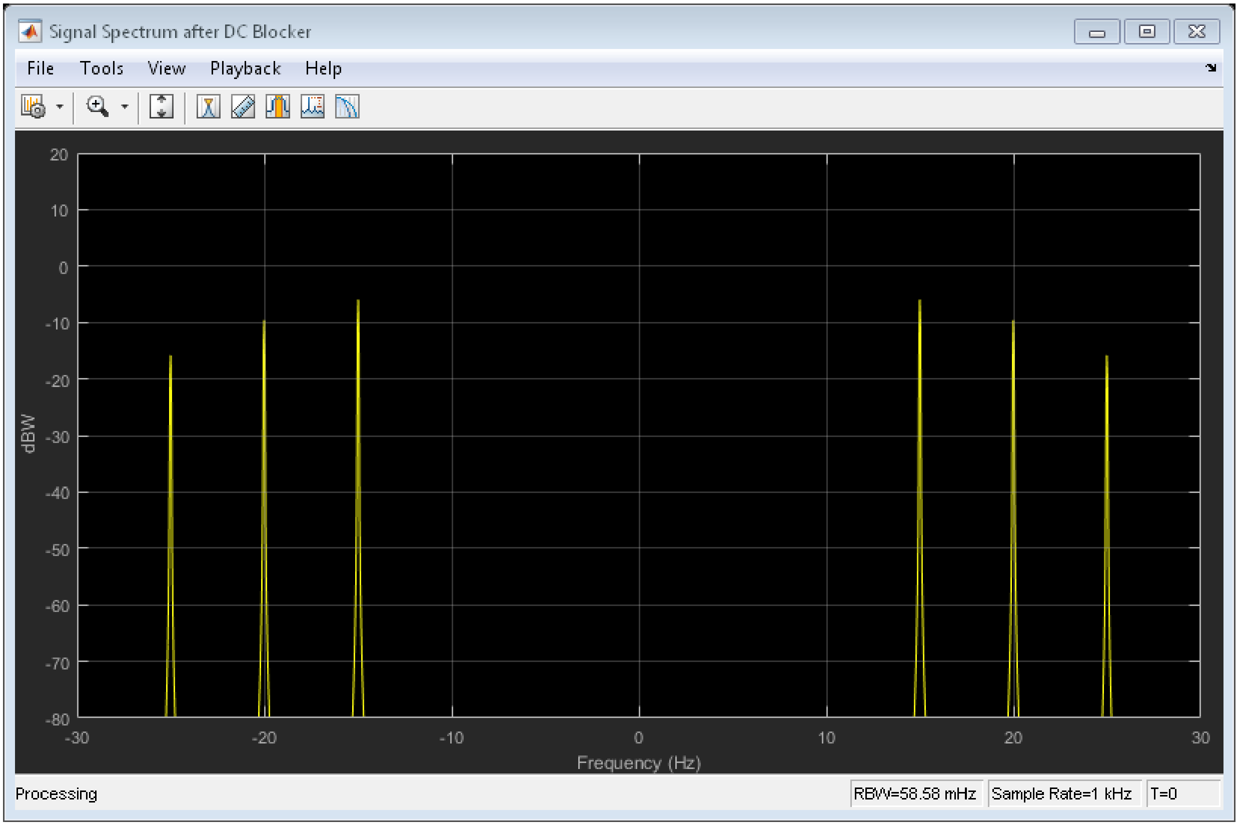
Use the spectrum analyzer, `hsa`, to display the frequency characteristics of the input signal. Note that the tones at 15 Hz, 20 Hz, and 25 Hz, and the DC component are clearly visible.

```
hsa(x)
```



Use the second spectrum analyzer, `hsb`, to display the frequency characteristics of the output signal. Note that the DC component has been removed.

```
hsb(y)
```



Remove DC Offset from Fixed-Point Data

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

This example shows how to use the DC Blocker to remove a DC offset from a fixed-point signal. The DC Blocker uses the CIC lowpass filtering method to estimate the DC offset.

Generate random binary data.

```
data = randi([0 1],1.2e5,1);
```

Modulate the data by creating a 64-QAM modulator System object and running its algorithm.

```
mod = comm.RectangularQAMModulator('ModulationOrder',64, ...
                                    'BitInput',true);
modOut = mod(data);
```

Determine the constellation reference points for the modulator.

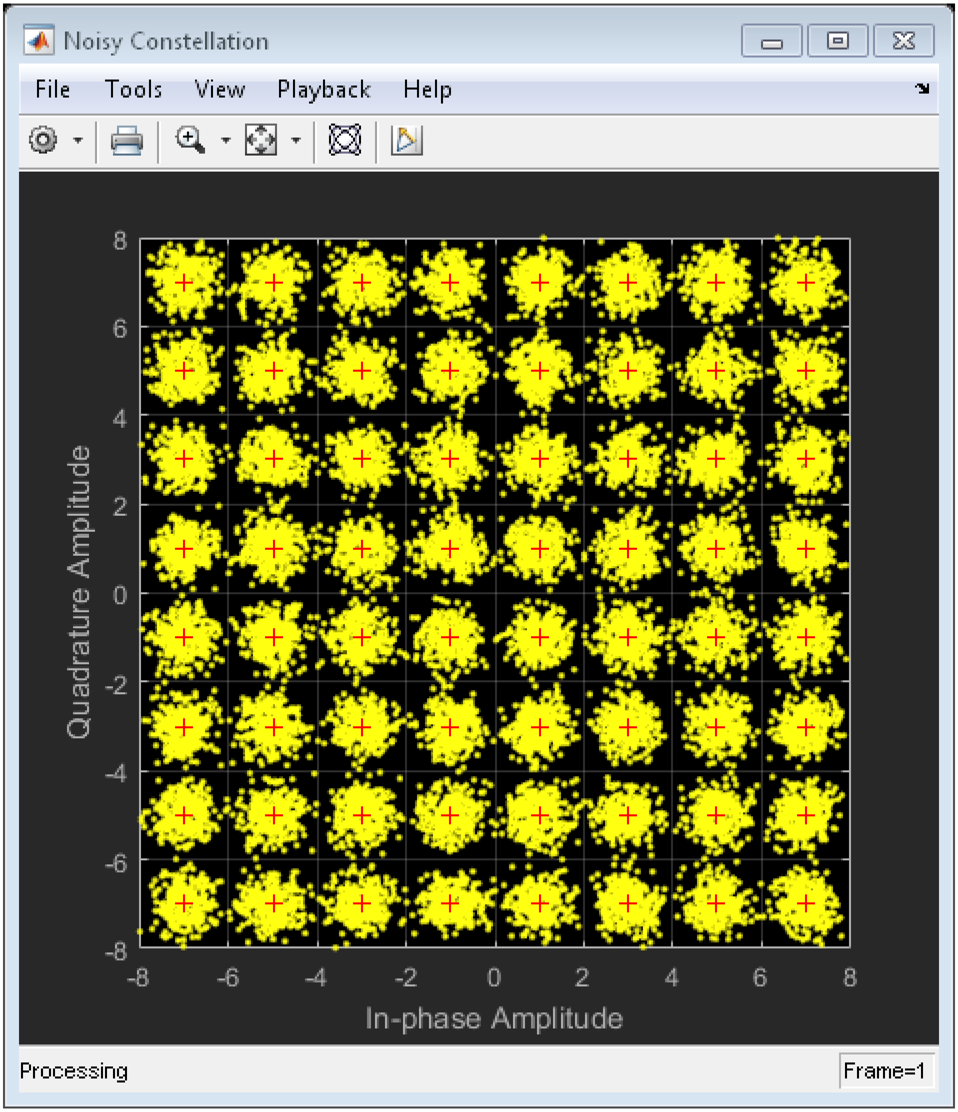
```
cRefPts = constellation(mod);
```

Add AWGN to the modulated signal using the appropriate System object and running its algorithm.

```
noise = comm.AWGNChannel('EbNo', 14.75, ...
                          'BitsPerSymbol', 6, ...
                          'SignalPower', 42, ...
                          'SamplesPerSymbol', 1);
noisyOut = noise(modOut);
```

Display the scatter plot of the noisy signal by using a Constellation Diagram object. The red '+' markers show the ideal symbol locations.

```
constDiag = comm.ConstellationDiagram('Name','Noisy Constellation',...
    'ReferenceConstellation',cRefPts, ...
    'XLimits',[-8 8],'YLimits',[-8 8]);
constDiag(noisyOut)
```

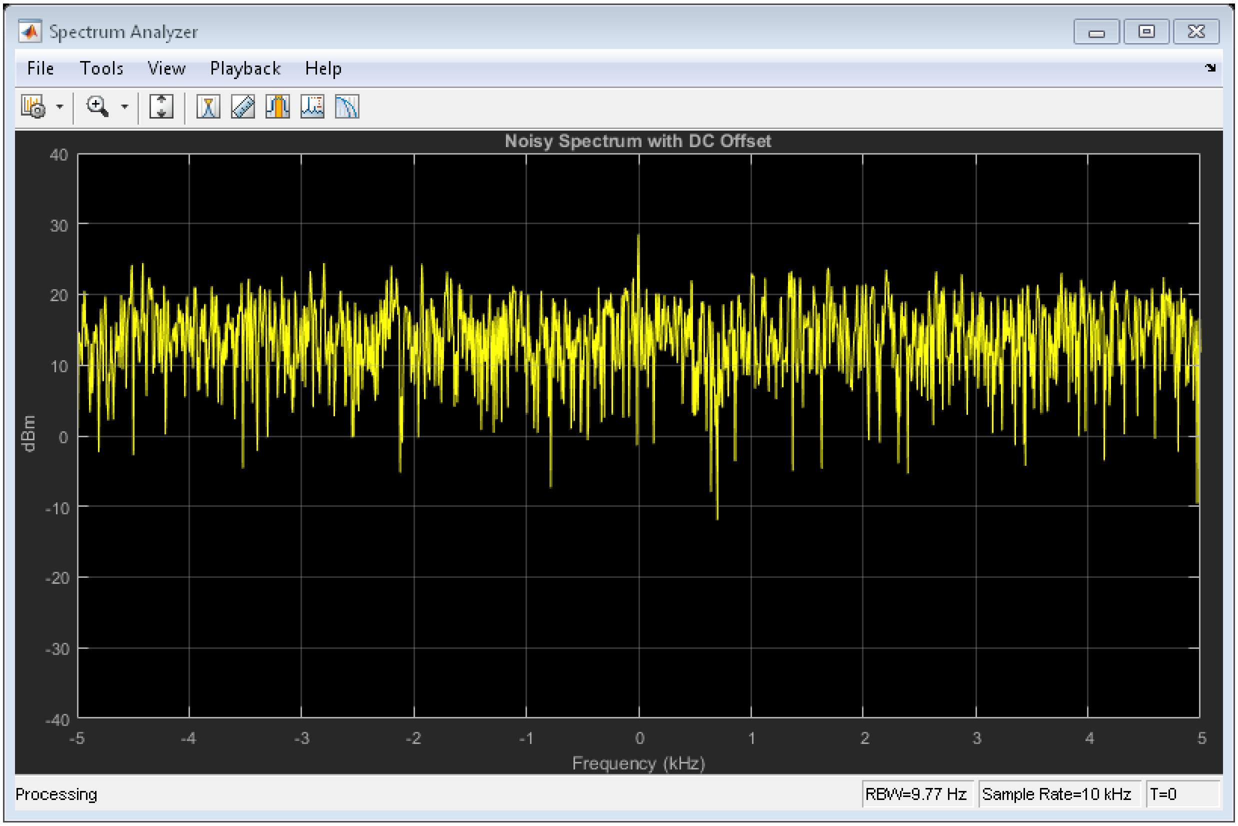


Add a DC offset of 1 to the modulated signal.

```
noisyOut = noisyOut + 1;
```

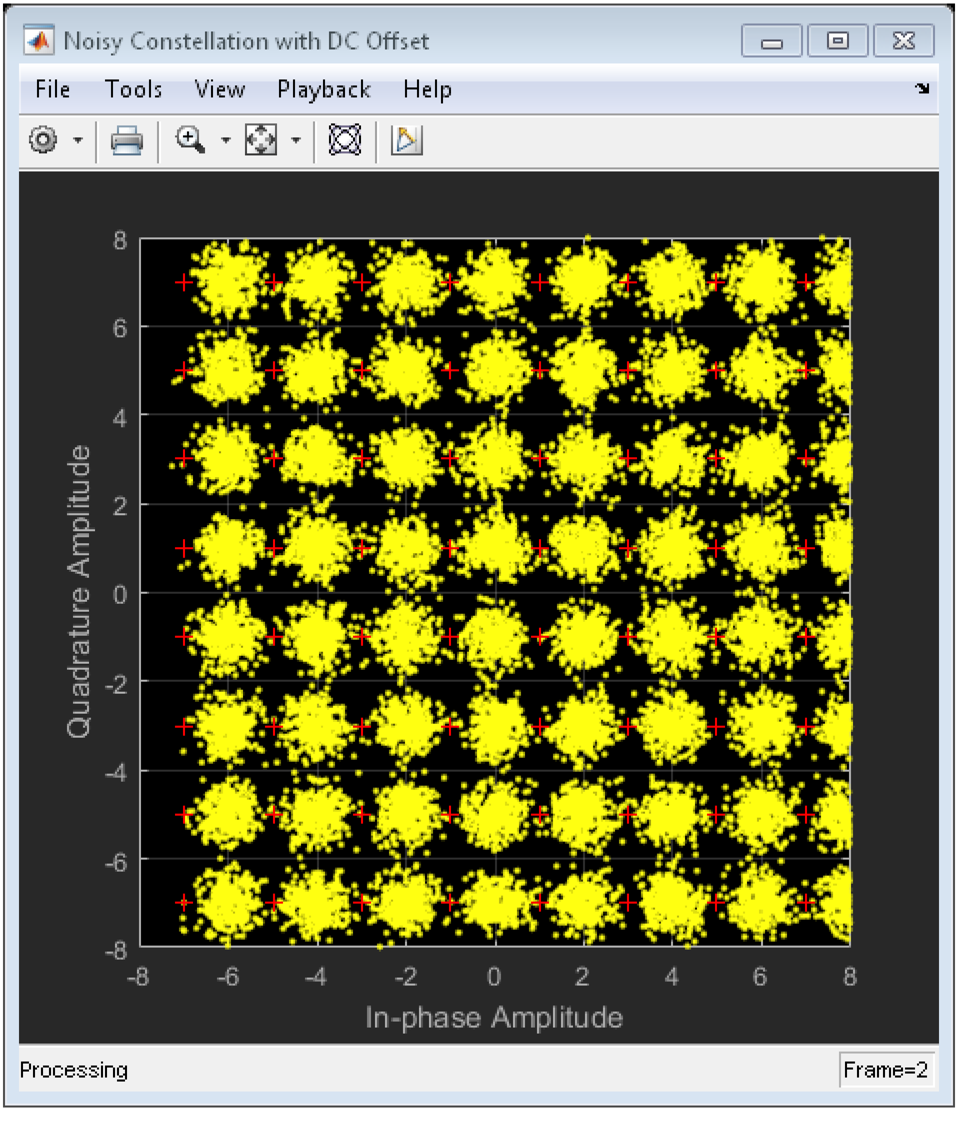
Display the spectrum of the signal. The spike at 0 kHz is due to the introduced offset.

```
specAna = dsp.SpectrumAnalyzer(...
  'YLimits',[-40,40], ...
  'Title','Noisy Spectrum with DC Offset');
specAna(noisyOut);
```



View the effect of the DC offset on the constellation. Observe that the constellation has shifted one unit to the right.

```
constDiag(noisyOut)
constDiag.Name = 'Noisy Constellation with DC Offset';
```



Convert the noisy signal to a signed, fixed-point object that has a 16-bit word length and an 11-bit fraction length.

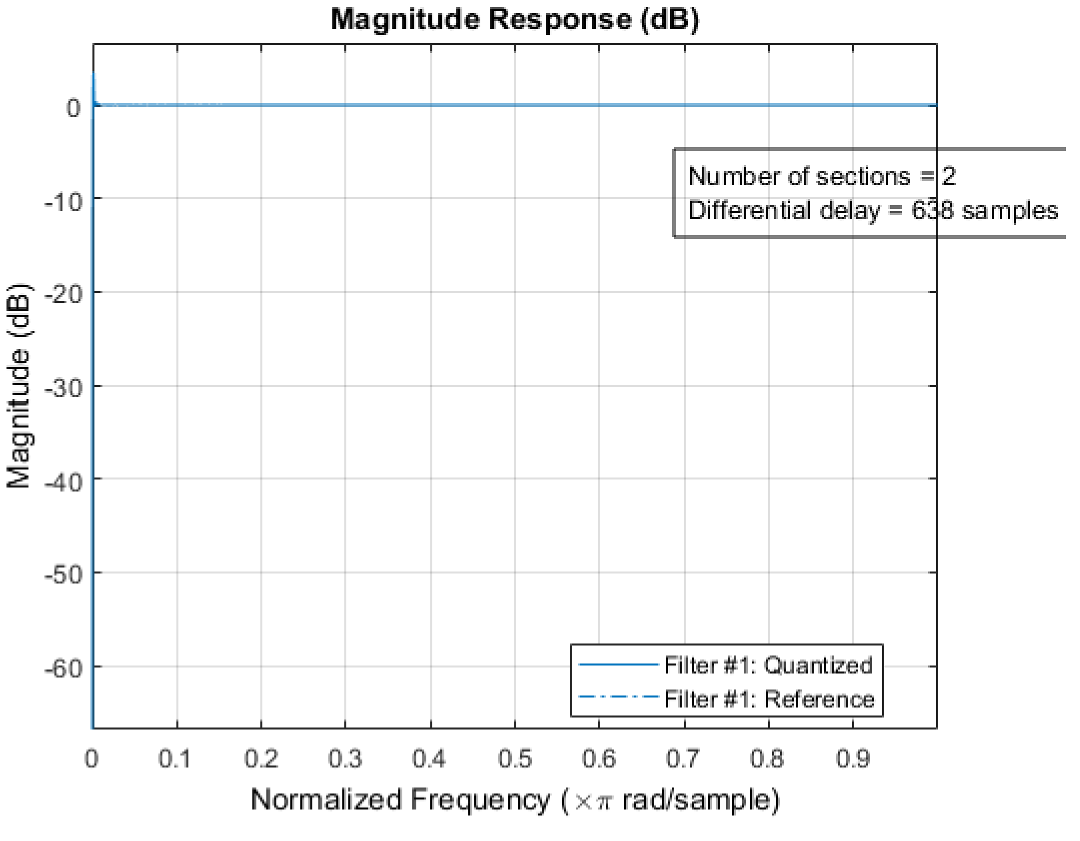
```
noisyOut = fi(noisyOut,1,16,11);
```

Remove the offset by creating a DC Blocker object. Set the `Algorithm` property of the object to `CIC`.

```
dcblk = dsp.DCBlocker('Algorithm','CIC');
```

Visualize the frequency response of the CIC estimating filter.

```
fvtool(dcblk)
```

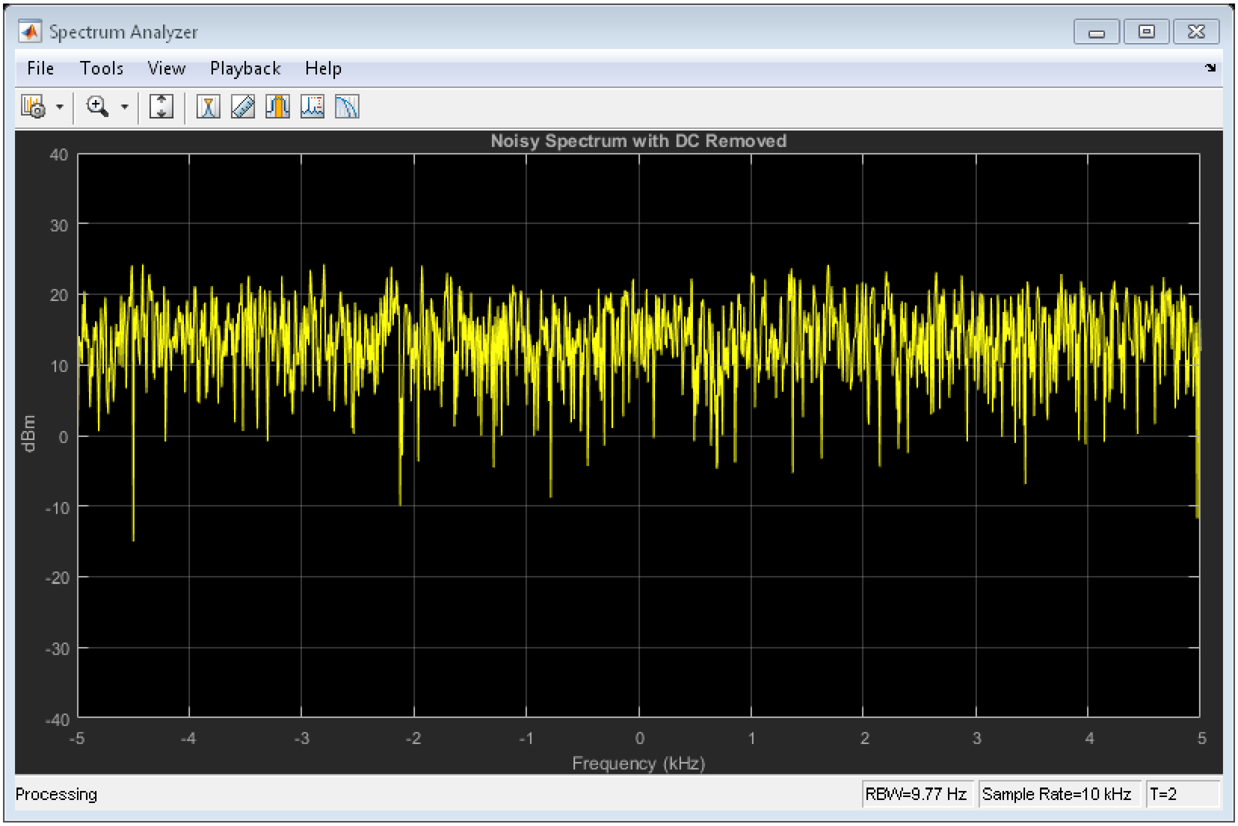


Pass the offset, noisy signal through the DC Blocker and convert its output to a double.

```
dcBlockerOutFxP = dcblk(noisyOut);
dcBlockerOut = double(dcBlockerOutFxP);
```

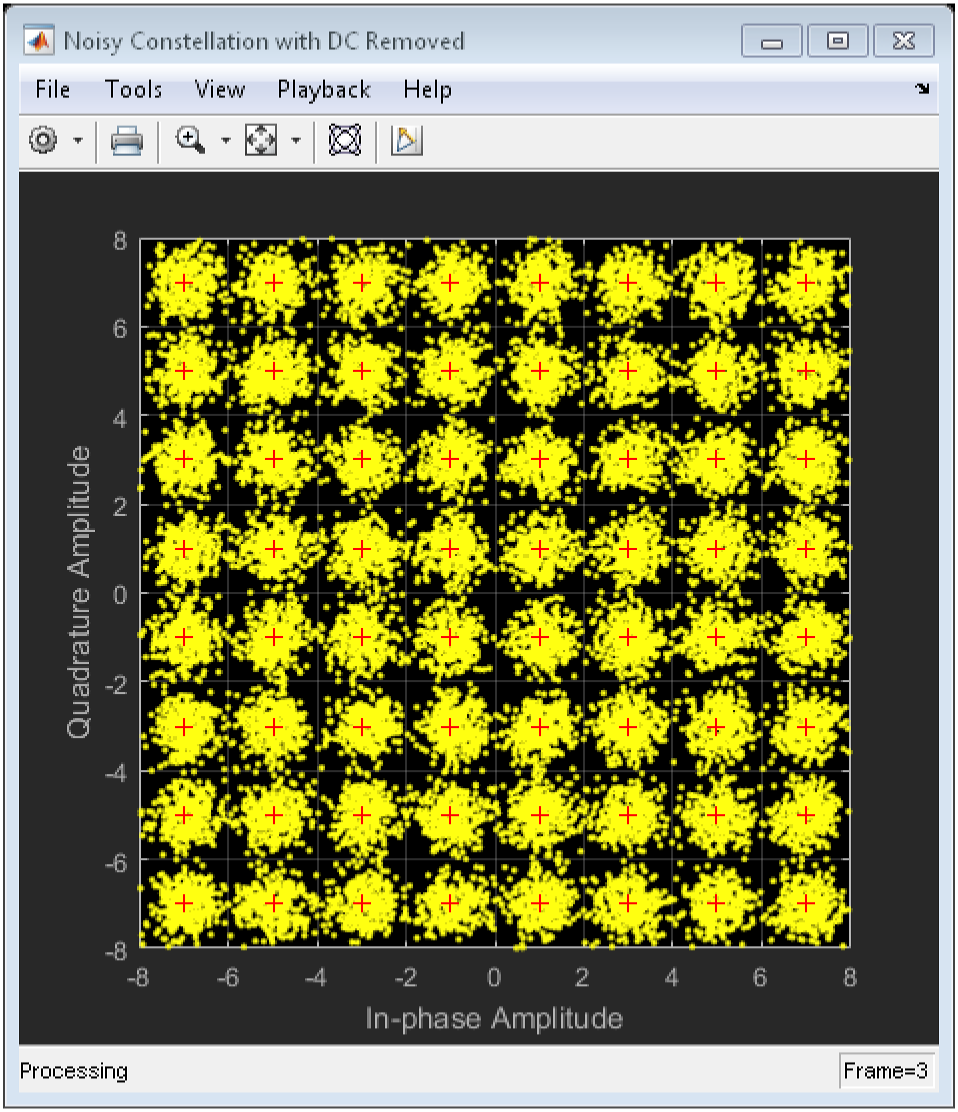
Plot the signal spectrum to show the effect of removing the DC offset. The spike at 0 kHz has been removed.

```
specAna(dcBlockerOut);
specAna.Title = 'Noisy Spectrum with DC Removed';
```



Plot the constellation and verify that the signal shifted back to the left.

```
constDiag(dcBlockerOut)
constDiag.Name = 'Noisy Constellation with DC Removed';
```



Noisy Constellation with DC Removed
File
Tools
View
Playback
Help
8
6
4
2
0
-2
-4
-6
-8
Quadrature Amplitude
-8
-6
-4
-2
0
2
4
6
8
In-phase Amplitude
Processing
Frame=3

To see the effects of removing a DC offset, vary the value of the `NormalizedBandwidth` property.

Algorithms

The `DCBlocker` System object subtracts the DC component from the input signal. The DC component is estimated by one of the following:

- Passing the input signal through an IIR lowpass elliptical filter
- Passing the input signal through an FIR filter that uses a non-recursive, moving average from a finite number of past input samples
- Passing the input signal through a CIC filter. Because the CIC filter amplifies the signal, the filter gain is estimated and subtracted from the DC estimate.
- Computing the mean value of the input signal

The elliptical IIR filter has a passband ripple of 0.1 dB and a stopband attenuation of 60 dB. You specify the normalized bandwidth and filter order.

The FIR filter coefficients are given as `ones(1,Length)/Length`, where you specify the `Length` parameter. The FIR filter structure is a direct form 1 transposed.

The Cascaded Integrator-Comb (CIC) filter consists of two integrator-comb pairs. This helps to ensure that the peak of the first sidelobe of the filter response is attenuated by at least 25 dB relative to the peak of the main lobe. The normalized 3-dB bandwidth is used to calculate the differential delay. The delay is used to determine the gain of the CIC filter. The inverse of the filter gain is used as a multiplier which is applied to the output of the CIC filter. This ensures that the aggregate gain of the DC estimate is 0 dB.

The aggregate magnitude response of the filter and the multiplier is characterized by the following equation:

$$\left|H(e^{j\omega})\right| = \left|\frac{\sin(M\frac{\pi}{2}B_{norm})}{M\sin(\frac{\pi}{2}B_{norm})}\right|^N$$

- B_{norm} is the normalized bandwidth such that $0 < B_{norm} < 1$.
- M is the differential delay in samples.
- N is the number of sections, equal to 2.

The differential delay is found by setting M to the smallest integer such that $|H(e^{j\omega})| < 1/\sqrt{2}$. Once M is known, the gain of the CIC filter is calculated as M^N. Therefore, to precisely compensate for the filter gain, the multiplier is set to $(1/M)^N$.

Selected Bibliography

[1] Nezami, M., "Performance Assessment of Baseband Algorithms for Direct Conversion Tactical Software Defined Receivers: I/Q Imbalance Correction, Image Rejection, DC Removal, and Channelization", MILCOM, 2002.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

dsp.BiquadFilter | dsp.FIRFilter | `DC Blocker` | dsp.CICDecimator

Introduced in R2014a

fvtool

System object: dsp.DCBlocker
Package: dsp

Show the frequency response of the filter used by the `DCBlocker` System object

Syntax

```
fvtool(dcblker)
```

Description

`fvtool(dcblker)` visualizes the frequency response of the DC blocker object, `dcblker`. With the fvtool, you can visualize the magnitude response, phase response, group delay, impulse response, pole-zero plot, and even view the coefficients of `dcblker`.

reset

System object: dsp.DCBlocker
Package: dsp

Reset states of the `DCBlocker` System object

Syntax

`reset(dcblker)`

Description

`reset(dcblker)` resets the states of the `DCBlocker` object, `dcblker`.

step

System object: dsp.DCBlocker
Package: dsp

Blocks DC components of input signal

Syntax

```
Y = step(dcblker,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(dcblker,X)` removes the DC component from the input signal, `X`, and returns the output, `Y`. The input signal is a numeric, signed matrix, with time samples recorded in rows and independent channels of data recorded in columns.

dsp.Counter System object

Package: dsp

Count up or down through specified range of numbers

Description

The `Counter` object counts up or down through a specified range of numbers.

To count up or down through a specified range of numbers:

1. Define and set up your counter. See "Construction" on page 4-458.
2. Call step to count up or down according to the properties of `dsp.Counter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`count = dsp.Counter` returns a counter System object, `count`, that counts up when the input is nonzero.

`count = dsp.Counter('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a counter System object, `count`, with each specified property set to the specified value.

Properties

`Direction`

Count up or down

Specify the counter direction as `Up` or `Down`. The default is `Up`. This property is tunable.

CountEventInputPort

Add input to specify a count event

Set this property to `true` to enable a count event input for the internal counter. The internal counter increments or decrements whenever the count event input satisfies the condition you specify in the `CountEventCondition on page 4-459` property. When you set this property to `false`, the internal counter is free running, that is, the counter increments or decrements on every call to the `step` method. The default is `true`.

CountEventCondition

Condition that increments, decrements, or resets internal counter

Specify the event at the count event input that increments or decrements the counter as `Rising edge`, `Falling edge`, `Either edge` or `Non-zero`.

If you set the `ResetInputPort on page 4-460` and `CountEventInputPort on page 4-459` properties to `true`, the counter is reset when the event you specify for the `CountEventCondition` occurs. This property applies only when you set the `CountEventInputPort on page 4-459` property to `true`.

CounterSizeSource

Source of counter size data type

Specify the source of the counter size data type as `Property` or `Input port`. The default is `Property`.

CounterSize

Range of integer values to count through

Specify the range of integer values to count through before recycling to zero as `8 bits`, `16 bits`, `32 bits` or `Maximum`. The default is `Maximum`.

MaximumCount

Counter's maximum value

Specify the counter's maximum value as a numeric scalar value. This property applies only when you set the `CounterSizeSource on page 4-` property to `Property` and the `CounterSize on page 4-` property to `Maximum`. The default is `255`. This property is tunable.

`InitialCount`

Counter initial value

Specify the initial value for the counter. The default is `0`. This property is tunable.

`CountOutputPort`

Output count

Set this property to `true` to enable output of the internal count. The default is `true`. You cannot set both `CountOutputPort` and `HitOutputPort on page 4-460` to `false` at the same time.

`HitOutputPort`

Output hit events

Set this property to `true` to enable output of the hit events. You cannot set both `CountOutputPort` on page 4-460 and `HitOutputPort` to `false` at the same time. The default is `true`.

`HitValues`

Values whose occurrence in count produce a true hit output

Specify an integer scalar or a vector of integers, whose occurrences in the count you want flagged as a hit. This property applies only when you set the `HitOutputPort on page 4-460` property to `true`.

`ResetInputPort`

Add input to enable internal counter reset

When you set this property to `true`, specify a reset input to the `step` method. When the reset input receives the event you specify for the `CountEventCondition on page 4-459` property, the counter resets. If you set the `CountEventInputPort on page 4-459` property to `false`, the counter resets whenever the reset input is not zero.

`SamplesPerFrame`

Number of samples in each output frame

Specify the number of samples in each output frame. This property applies only when you set the `CountEventInputPort on page 4-459` property to `false`, indicating a free-running counter. The default is `1`.

`CountOutputDataType`

Data type of count output

Specify the data type of the count output, CNT, as `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32` or `uint32`. This property applies when you set the `CountOutputPort on page 4-460` property to `true`. The default is `double`.

Methods

reset	Reset internal states of Counter object
step	Increment or decrement the internal counter

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Count the Rising Edges

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Use `dsp.Counter` System object™ to count at every rising edge of the input signal.

```
count = dsp.Counter('MaximumCount', 5, ...
```

```
    'CountOutputPort', true, ...
    'HitOutputPort', false, ...
    'ResetInputPort', false);
sgnl = [0 1 0 1 0 1 0 1 0 1 0 1 ];
for ii = 1:length(sgnl)
    cnt(ii) = count(sgnl(ii));
end
display(cnt);
```

```
cnt =

     0     1     1     2     2     3     3     4     4     5     5     0
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Counter` block reference page. The object properties correspond to the block parameters.

- The `CountEventCondition` object property does not have a free-running option. Set the `CountEventInputPort` property to `false` to obtain the free-running option.
- The `CounterSizeSource` and `CounterSize` object properties correspond to the **Counter size** block parameter.
- The `CountOutputPort` and `HitOutputPort` correspond to the **Output** block parameter.
- There is no object property that corresponds to the **Hit data type** block parameter. The output type is logical in MATLAB. (This logical is different from the popup logical in the block. For the object, logical corresponds to Boolean in the block.)

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

Introduced in R2012a

reset

System object: dsp.Counter
Package: dsp

Reset internal states of Counter object

Syntax

`reset(count)`

Description

`reset(count)` resets the internal states of the counter object `count` to their initial values.

step

System object: dsp.Counter
Package: dsp

Increment or decrement the internal counter

Syntax

```
[CNT,HIT] = step(count,EVENT,RESET)
CNT = step(count,EVENT,RESET)
HIT = step(count,EVENT,RESET)
[...] = step(count)
[...] = step(count,EVENT)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[CNT,HIT] = step(count,EVENT,RESET)` increments, decrements, or resets the internal counter as specified by the values of the `EVENT` and `RESET` inputs. The output argument `CNT` denotes the present value of the counter. A trigger event at the `EVENT` input causes the counter to increment or decrement. A trigger event at the `RESET` input resets the counter to its initial state.

`CNT = step(count,EVENT,RESET)` returns the current value of the count when you set the `CountOutputPort` property to `true`, and the `HitOutputPort` property to `false`.

`HIT = step(count,EVENT,RESET)` returns a Boolean value indicating whether the count has reached any of the values specified by the `HitValues` property. This condition applies when you set the `HitOutputPort` property to `true` and the `CountOutputPort` property to `false`.

`[...] = step(count)` increments or decrements the free-running internal counter when you set the `CountEventInputPort` property to `false` and the `ResetInputPort` property to `false`.

`[...] = step(count,EVENT)` increments or decrements the internal counter when the `EVENT` input matches the event you specify for the `CountEventCondition` property and you set the `ResetInputPort` property to `false`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.CoupledAllpassFilter System object

Package: dsp

Coupled allpass IIR filter

Description

The `CoupledAllpassFilter` object implements a coupled allpass filter structure composed of two allpass filters connected in parallel. Each allpass branch can contain multiple sections. The overall filter output is computed by adding the output of the two respective branches. An optional second output can also be returned, which is power complementary to the to the first. For example, from the frequency domain perspective, if the first output implements a lowpass filter, the second output implements the power complementary highpass filter. For real signals, the power complementary output is computed by subtracting the output of the second branch from the first. `CoupledAllpassFilter` supports double- and single-precision floating point and allows you to choose between different realization structures. This System object also supports complex coefficients, multichannel variable length input, and tunable filter coefficient values.

To filter each channel of the input:

1. Define and set up your Coupled Allpass Filter. See “Construction” on page 4-468.
2. Call step to filter each channel of the input according to the properties of `dsp.CoupledAllpassFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`caf = dsp.CoupledAllpassFilter` returns a coupled allpass filter System object, `caf`, that filters each channel of the input signal independently. The coupled allpass filter uses the default inner structures and coefficients.

`caf = dsp.CoupledAllpassFilter('PropertyName',PropertyValue, ...)` returns a Coupled allpass filter System object, `caf`, with each property set to the specified value.

`caf = CoupledAllpassFilter(AllpassCoefficients1,AllpassCoefficients2)` returns a coupled allpass filter System object, `caf`, with `Structure` set to `'Minimum multiplier'`. The allpass coefficients for each of the two branches are set to their two corresponding specified values.

`caf = CoupledAllpassFilter(Structure,AllpassCoefficients1,AllpassCoefficients2)` returns a coupled allpass filter System object, `caf`, with `Structure` set to a specified value. This value can be `'Minimum multiplier'` | `'Wave Digital Filter'` | `'Lattice'`. The coefficients relevant to the specified structure are set to the values provided.

Properties

`Structure`

Internal structure of allpass branches

Specify the internal structure of allpass branches as one of `'Minimum multiplier'` | `'Wave Digital Filter'` | `'Lattice'`. Each structure uses a different pair of coefficient values, independently stored in the relevant object property.

`AllpassCoefficients1`

Allpass polynomial coefficients of branch 1

Specify the polynomial filter coefficients for the first allpass branch. This property is applicable only if you set the `Structure` property to `'Minimum multiplier'`. This property can accept values either in the form of a row vector (single-section configuration) or a cell array with as many cells as filter sections. The default value is `[0 0.5]`. This property is tunable.

WDFCoefficients1

Wave Digital Filter coefficients of branch 1

Specify the Wave Digital Filter coefficients for the first allpass branch. This property is applicable only if you set the `Structure` property to `'Wave Digital Filter'`. This property can accept values either in the form of a row vector (single-section configuration) or a cell array with as many cells as filter sections. The default value is `[0.5 0]`. This property is tunable.

LatticeCoefficients1

Lattice coefficients of branch 1

Specify the allpass lattice coefficients for the first allpass branch. This property is applicable only if you set the `Structure` property to `'Lattice'`. This property can accept values either in the form of a row vector (single-section configuration) or a cell array with as many cells as filter sections. The default value is `[0.5 0]`. This property is tunable.

Delay

length in samples for branch 1

`Delay` is the integer number of the delay taps in the top branch. This property is applicable only if you set the `PureDelayBranch` property to `true`. This property is a scalar positive integer, and is tunable.

Gain1

Independent Branch 1 Phase Gain

`Gain1` is the individual branch phase gain. This property can accept only values equal to `'1'`, `'-1'`, `'0+i'`, or `'0-i'`. The default value is `1`. This property is nontunable.

AllpassCoefficients2

Allpass polynomial coefficients of branch 2

Specify the polynomial filter coefficients for the second allpass branch. This property is applicable only if you set the `Structure` property to `'Minimum Multiplier'`.

This property can accept values either in the form of a row vector (single-section configuration) or a cell array with as many cells as filter sections. The default value is [ ]. This property is tunable.

WDFCoefficients2

Wave Digital Filter coefficients of branch 2

This property is applicable only if you set the `Structure` property to `'Wave Digital Filter'`. This property can accept values either in the form of a row vector (single-section configuration) or a cell array with as many cells as filter sections. The default value is [ ]. This property is tunable.

LatticeCoefficients2

Lattice coefficients of branch 2

Specify the allpass lattice coefficients for the second allpass branch. This property is applicable only if you set the `Structure` property to `'Lattice'`. This property can accept values either in the form of a row vector (single-section configuration) or a cell array with as many cells as filter sections. The default value is [ ]. This property is tunable.

Gain2

Independent Branch 2 Phase Gain

Specify the value of the independent phase gain applied to branch 2. This property can accept only values equal to `'1', '–1', '0+1i'` or `'0–1i'`. The default value is `1`. This property is nontunable.

Beta

Coupled phase gain

Specify the value of the phasor gain in complex conjugate form, in each of the two branches, and in complex coefficient configuration. This property is applicable only when the selected `Structure` property supports complex coefficients. The absolute value of this property should be `1` and its default value is `1`. This property is tunable.

PureDelayBranch

Replace allpass filter in first branch with pure delay

If you set `PureDelayBranch` to true, the property holding the coefficients for the first allpass branch is disabled and Delay becomes enabled. You can use this property to improve performance, when one of the two allpass branches is known to be a pure delay (e.g. for halfband filter designs)

ComplexConjugateCoefficients

Allow inferring coefficients of second allpass branch as complex conjugate of first

When the input signal is real, this property triggers the use of an optimized structural realization. This property is only enabled if the currently selected structure supports complex coefficients. Use it only if the filter coefficients are actually complex.

Methods

getBranches	Return internal allpass branches
reset	Reset internal states of Coupled Allpass filter
step	Filter input with CoupledAllpas filter object

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

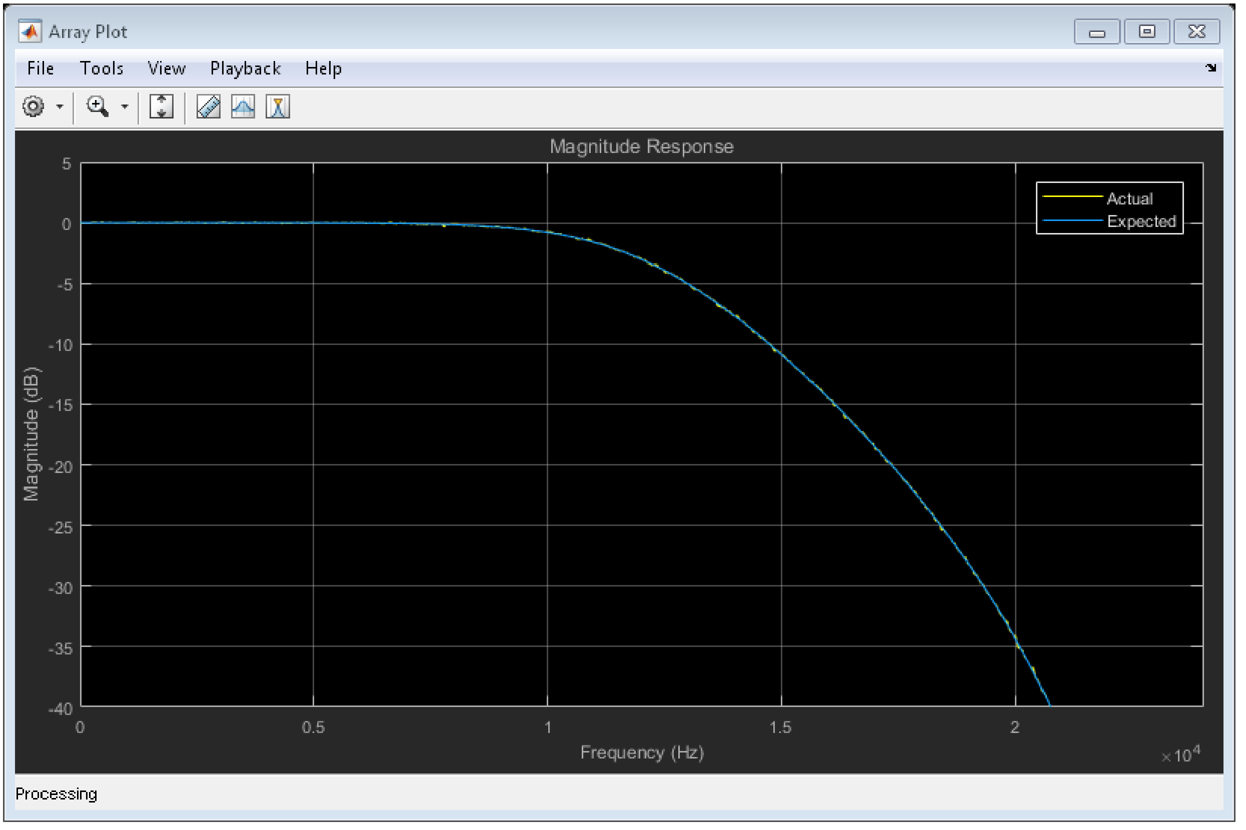
Examples

Allpass Realization of a Butterworth Lowpass Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Realize a Butterworth lowpass filter of order 3. Use a coupled allpass structure with inner minimum multiplier structure.

```
Fs = 48000;    % in Hz
Fc = 12000;    % in Hz
frameLength = 1024;
[b, a] = butter(3,2*Fc/Fs);
AExp = [freqz(b,a,frameLength/2); NaN];
[c1, c2] = tf2ca(b,a);
caf = dsp.CoupledAllpassFilter(c1(2:end),c2(2:end));
tfe = dsp.TransferFunctionEstimator('FrequencyRange', 'onesided',...
    'SpectralAverages', 2);
aplot = dsp.ArrayPlot('PlotType', 'Line',...
    'YLimits', [-40 5],...
    'YLabel', 'Magnitude (dB)',...
    'SampleIncrement', Fs/frameLength,...
    'XLabel', 'Frequency (Hz)',...
    'Title', 'Magnitude Response',...
    'ShowLegend', true,'ChannelNames',{'Actual','Expected'});
Niter = 200;
for k = 1:Niter
    in = randn(frameLength,1);
    out = caf(in);
    A = tfe(in,out);
    aplot(db([A,AExp]));
end
```



Allpass Realization of an Elliptic Highpass Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Remove a low-frequency sinusoid using an elliptic highpass filter design implemented through a coupled allpass structure.

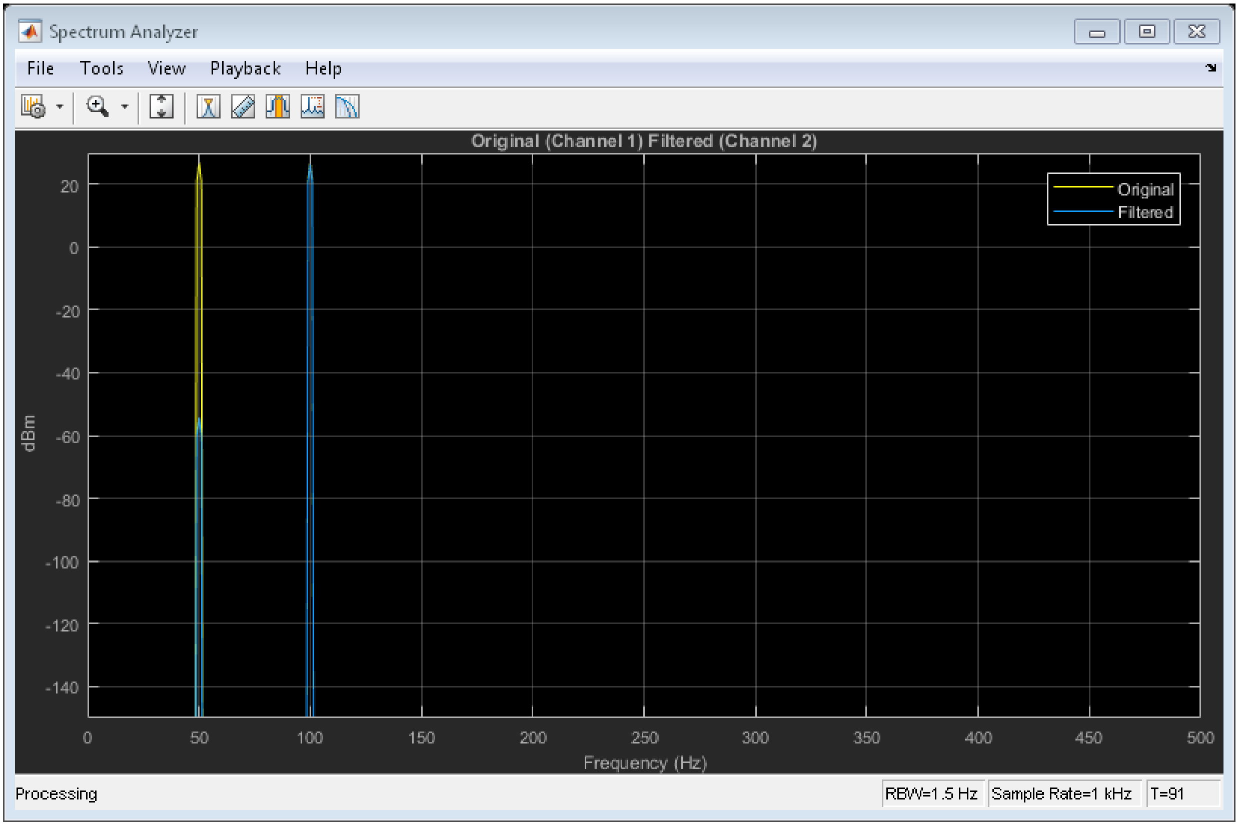
Initialize

```
Fs = 1000;
f1 = 50; f2 = 100;
Fpass = 70; Apass = 1;
```

```
Fstop = 60; Astop = 80;
filtSpecs = fdesign.highpass(Fstop,Fpass,Astop,Apass,Fs);
hpSpec = design(filtSpecs,'ellip','FilterStructure','cascadeallpass',...
    'SystemObject',true);
frameLength = 1000;
nFrames = 100;
sine   = dsp.SineWave('Frequency',[f1,f2],'SampleRate',Fs,...
  'SamplesPerFrame',frameLength); % Input composed of two sinusoids.
sa = dsp.SpectrumAnalyzer('SampleRate',Fs,'YLimits',[-150 30],...
  'PlotAsTwoSidedSpectrum',false,'ShowLegend',true,...
  'FrequencyResolutionMethod', 'WindowLength','WindowLength',1000,...
  'FFTLengthSource', 'Property','FFTLength', 1000,...
  'Title','Original (Channel 1) Filtered (Channel 2)',...
  'ChannelNames',{'Original','Filtered'});
```

Simulate

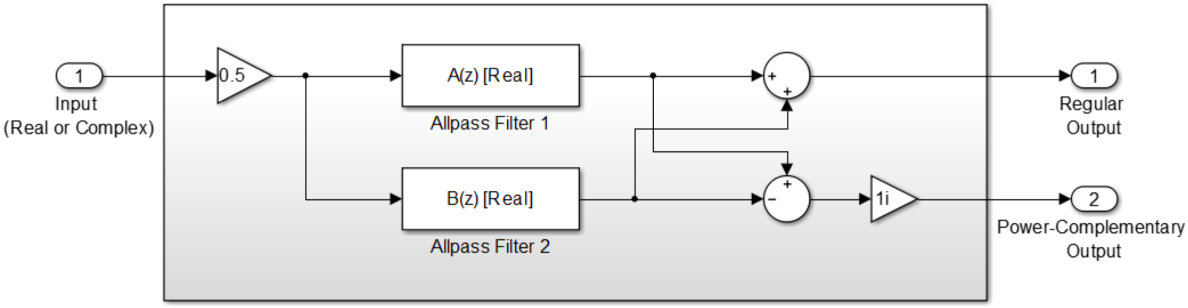
```
for k = 1:nFrames
  original = sum(sine(),2); % Add the two sinusoids together
  filtered = hpSpec(original);
  sa([original,filtered]);
end
```



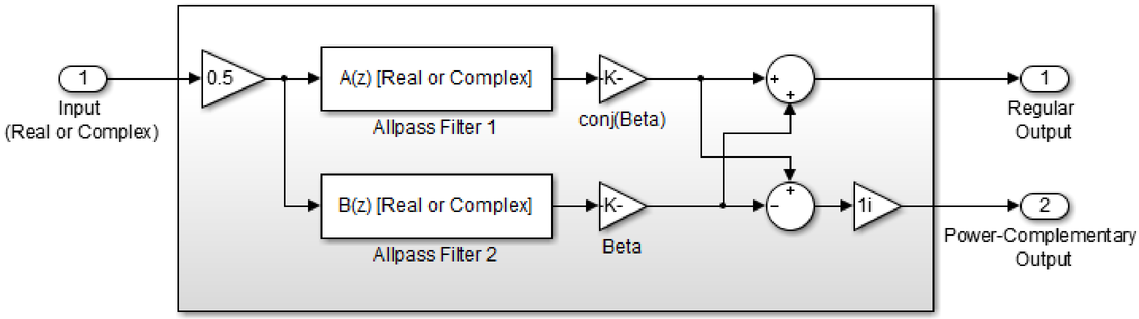
Algorithms

The following three figures summarize the main structures supported by `dsp.CoupledAllpassFilter`.

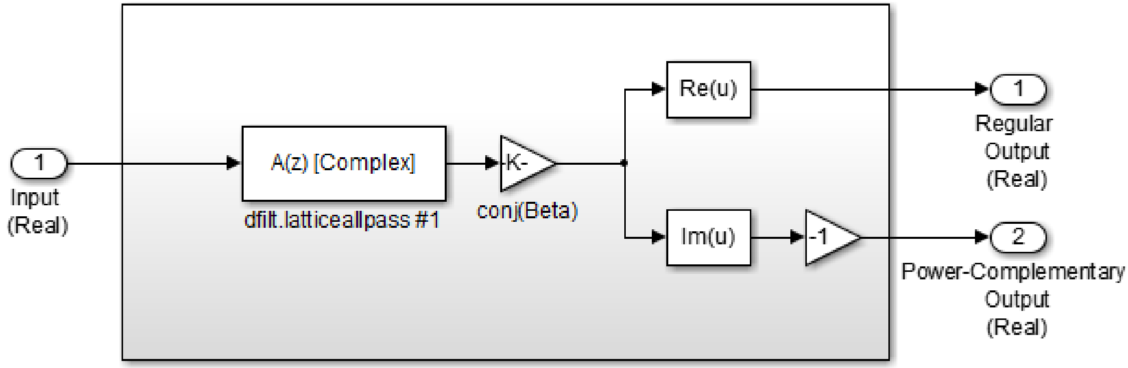
- Minimum Multiplier and WDF



- Lattice



- Lattice with Complex Conjugate Coefficients



References

[1] Regalia, Philip A., Mitra, Sanjit K., and P.P Vaidyanathan " The Digital All-Pass Filter: A Versatile Signal Processing Building Block." *Proceedings of the IEEE 1988,* Vol. 76, No. 1, pp. 19–37.

[2] Mitra, Sanjit K., and James F. Kaiser, *"Handbook for Digital Signal Processing"* New York: John Wiley & Sons, 1993.

See Also

dsp.AllpassFilter | dsp.IIRFilter | dsp.BiquadFilter

Introduced in R2013b

getBranches

System object: dsp.CoupledAllpassFilter
Package: dsp

Return internal allpass branches

Syntax

```
getBranches(caf)
```

Description

`getBranches(caf)` returns copies of the internal allpass branches, as a two-field structure. Each branch is an instance of dsp.AllpassFilter.

reset

System object: dsp.CoupledAllpassFilter
Package: dsp

Reset internal states of Coupled Allpass filter

Syntax

```
reset(caf)
```

Description

`reset(caf)` resets the internal states of the Coupled Allpass filter object, `caf`, to their initial values. The reset method is always a no-op for unlocked System objects, as the states may not be allocated when the object is not locked.

step

System object: dsp.CoupledAllpassFilter
Package: dsp

Filter input with CoupledAllpas filter object

Syntax

```
Y = step(caf,X)
[Y1,...,YN] = step(caf,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(caf,X)` processes the input data, `X` to produce the output, `Y`, for System object, `caf`.

`[Y1,...,YN] = step(caf,X)` produces `N` outputs.

Every System object has a step method. The step method processes the input data according to the object algorithm. The number of input and output arguments depends on the algorithm, and may depend also on one or more property settings. The step method for some objects accepts fixed-point (fi) inputs (but not for this object).

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change

nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.CrossSpectrumEstimator System object

Package: dsp

Estimate cross-spectral density

Description

The `dsp.CrossSpectrumEstimator` computes the cross- spectrum density of a signal, using the Welch algorithm and the Periodogram method.

To implement the cross-spectrum estimation object:

1. Define and set up your cross-spectrum estimator object. See “Construction” on page 4-482.
2. Call step to implement the estimator according to the properties of `dsp.CrossSpectrumEstimator`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`cse = dsp.CrossSpectrumEstimator` returns a System object, `cse`, that computes the cross-power spectrum of real or complex signals using the periodogram method and Welch’s averaged, modified periodogram method.

`cse = dsp.CrossSpectrumEstimator('`*`PropertyName`*`',` *`PropertyValue`*`,...)` returns a `Cross-Spectrum Estimator` System object, `cse`, with each specified property name set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

SampleRate

Sample rate of input

Specify the sample rate of the input, in hertz, as a finite numeric scalar. The default value is `1 Hz`. The sample rate is the rate at which the signal is sampled in time.

SpectralAverages

Number of spectral averages

Specify the number of spectral averages as a positive, integer scalar. The Cross-Spectrum Estimator computes the current cross-spectral estimate by averaging the last `N` estimates. `N` is the number of spectral averages defined in the `SpectralAverages` property. The default is `8`.

FFTLengthSource

Source of the FFT length value

Specify the source of the FFT length value as one of `'Auto' | 'Property'`. The default is `'Auto'`. If you set this property to `'Auto'`, the Cross-Spectrum Estimator sets the FFT length to the input frame size. If you set this property to `'Property'`, then you specify the number of FFT points using the `FFTLength` property.

FFTLength

FFT Length

Specify the length of the FFT that the Cross-Spectrum Estimator uses to compute cross-spectral estimates as a positive, integer scalar. This property applies when you set the `FFTLengthSource` property to `'Property'`. The default value is `128`.

Window

Window function

Specify a window function for the cross-spectral estimator as one of `'Rectangular' | 'Chebyshev' | 'Flat Top' | 'Hamming' | 'Hann' | 'Kaiser'`. The default value is `'Hann'`.

SidelobeAttenuation

Side lobe attenuation of window

Specify the side lobe attenuation of the window as a real, positive scalar, in decibels (dB). This property applies when you set the `Window` property to `'Chebyshev'` or `'Kaiser'`. The default is `60 dB`.

FrequencyRange

Frequency range of the cross-spectrum estimate

Specify the frequency range of the cross-spectrum estimator as one of `'twosided' | 'onesided' | 'centered'`.

If you set the `FrequencyRange` to `'onesided'`, the cross-spectrum estimator computes the onesided spectrum of real input signals, `x` and `y`. If the FFT length, NFFT, is even, the length of the cross-spectrum estimate is `NFFT/2+1` and is computed over the interval `[0,SampleRate/2]`. If NFFT is odd, the length of the cross-spectrum estimate is equal to `(NFFT+1)/2` and the interval is `[0,SampleRate/2]`.

If you set the `FrequencyRange` to `'twosided'`, the cross-spectrum estimator computes the twosided spectrum of complex or real input signals, `x` and `y`. The length of the cross-spectrum estimate is equal to NFFT. This value is computed over `[0, SampleRate]`.

If you set the `FrequencyRange` to `'centered'`, the cross-spectrum estimator computes the centered twosided spectrum of complex or real input signals, `x` and `y`. The length of the cross-spectrum estimate is equal to NFFT and it is computed between `[-SampleRate/2, SampleRate/2]` and `(-SampleRate/2, SampleRate/2)` for even and odd lengths, respectively. The default value is `'Twosided'`.

Methods

getFrequencyVector	Get the vector of frequencies at which the cross-spectrum is estimated
getRBW	Get the resolution bandwidth of the cross-spectrum
reset	Reset the internal states of the cross-spectrum estimator

step	Estimate the cross-spectrum of a signal

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Compute the Cross-Power Spectrum of Two Noisy Sine Waves

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Generate two sine waves.

```
sin1 = dsp.SineWave('Frequency',200, 'SampleRate', 1000);
sin1.SamplesPerFrame = 1000;
sin2 = dsp.SineWave('Frequency',100, 'SampleRate', 1000);
sin2.SamplesPerFrame =  1000;
```

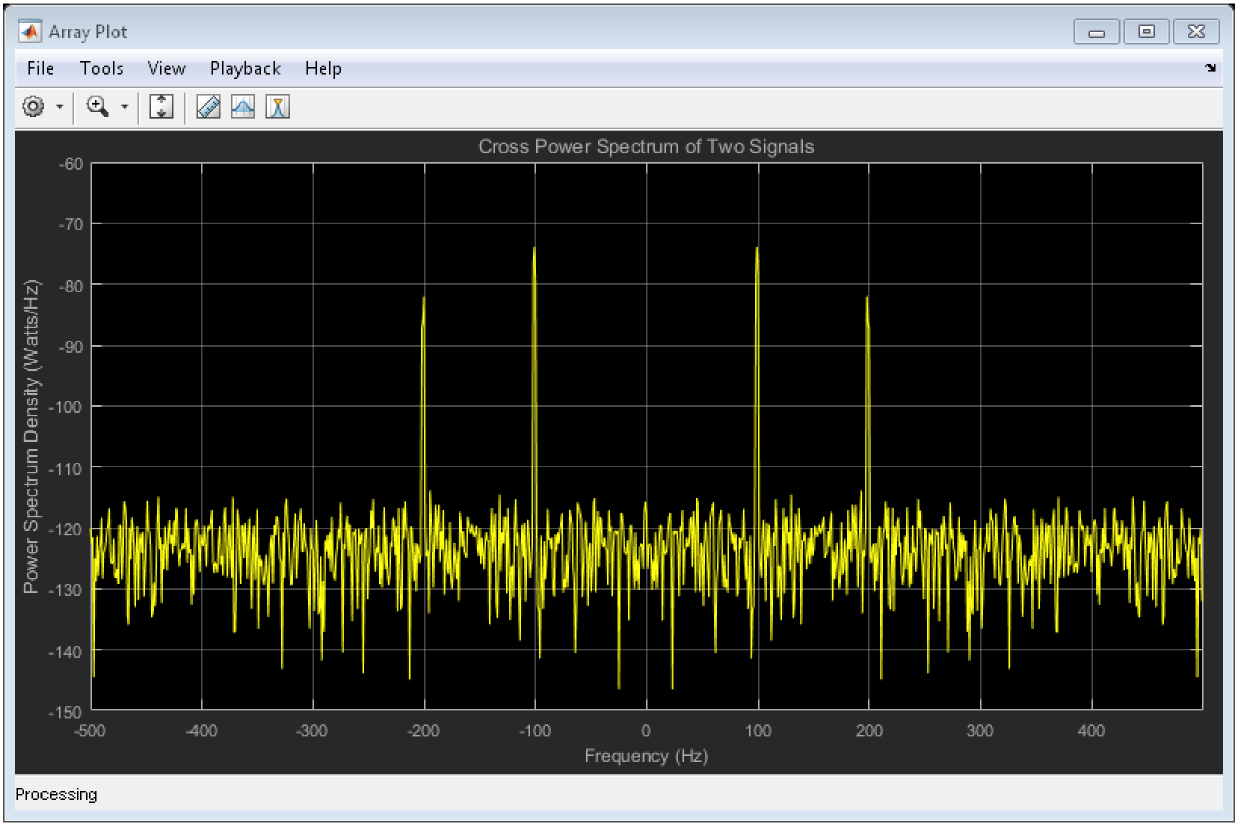
Use the Cross-Spectrum Estimator to compute the cross-spectrum of the signals. Also, use the Array Plot to display the spectra.

```
cse = dsp.CrossSpectrumEstimator('SampleRate', sin1.SampleRate,...
    'FrequencyRange','centered');
aplot = dsp.ArrayPlot('PlotType','Line','XOffset',-500,'YLimits',...
    [-150 -60],'YLabel','Power Spectrum Density (Watts/Hz)',...
    'XLabel','Frequency (Hz)',...
    'Title','Cross Power Spectrum of Two Signals');
```

Add random noise to the sine waves. Stream in the data, and plot the cross-power spectrum of the two signals.

```
for ii = 1:10
```

```
x = sin1() + 0.05*randn(1000,1);
y = sin2() + 0.05*randn(1000,1);
Pxy = cse(x, y);
aplot(20*log10(abs(Pxy)));
end
```



Algorithms

Given two signals `x` and `y` as inputs. We first window the two inputs, and scale them by the window power. We then take FFT of the signals, calling them `X` and `Y`. This is followed by taking the cross correlation of the FFT, i.e., `Z = X*conj(Y)`. We average the last N number of `Z`'s, and scale the answer by the sample rate.

For further information, refer to the “Algorithms” on page 2-1592 section in Spectrum Analyzer, which uses the same algorithm.

References

[1] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996.

[2] Kay, Steven M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1999.

[3] Stoica, Petre, and Randolph L. Moses. *Spectral Analysis of Signals*. Englewood Cliffs, NJ: Prentice Hall, 2005.

[4] Welch, P. D. "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short Modified Periodograms". *IEEE Transactions on Audio and Electroacoustics*. Vol. 15, No. 2, June 1967, pp. 70–73.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- When the FFT length is not a power of two, the executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- When the FFT length is a power of two, you can generate standalone C and C++ code from this System object.

See Also

See Also

System Objects

dsp.SpectrumEstimator | dsp.SpectrumAnalyzer | dsp.TransferFunctionEstimator

Blocks

`Cross Spectrum Estimator`

Introduced in R2013b

getFrequencyVector

System object: dsp.CrossSpectrumEstimator
Package: dsp

Get the vector of frequencies at which the cross-spectrum is estimated

Syntax

`getFrequencyVector(cse)`

Description

`getFrequencyVector(cse)` returns the vector of frequencies at which the cross-spectrum is estimated.

If you set the `FrequencyRange` to `'onesided'` and the FFT length, `NFFT`, is even, the frequency vector is of length `NFFT/2+1`, and covers the interval `[0,SampleRate/2]`. If you set the `FrequencyRange` to `'onesided'` and `NFFT` is odd, the frequency vector is of length `(NFFT+1)/2` and covers the interval `[0,SampleRate/2]`. If you set the `FrequencyRange` to `'twosided'`, the frequency vector is of length `NFFT` and covers the interval `[0, SampleRate]`. If you set the `FrequencyRange` to `'centered'`, the frequency vector is of length `NFFT` and covers the range `[-SampleRate/2, SampleRate/2]` and `[-SampleRate/2, SampleRate/2]` for even and odd length `NFFT`, respectively.

getRBW

System object: dsp.CrossSpectrumEstimator
Package: dsp

Get the resolution bandwidth of the cross-spectrum

Syntax

```
getRBW(cse)
```

Description

`getRBW(cse)` returns the resolution bandwidth of the cross-spectrum.

The resolution bandwidth, `RBW`, is the smallest positive frequency, or frequency interval, that can be resolved. It is equal to `ENBW*SampleRate/ L`, where `L` is the input length, and `ENBW` is the two-sided equivalent noise bandwidth of the window (in Hz). For example, if `SampleRate=100`, `L=1024`, and `Window='Hann'`, `RBW=enbw(hann(1024))*100/1024`.

reset

System object: dsp.CrossSpectrumEstimator
Package: dsp

Reset the internal states of the cross-spectrum estimator

Syntax

`reset(sysObj)`

Description

`reset(sysObj)` resets the internal states of the System object, `sysObj`, to their initial values. The reset method is always a no-op for unlocked System objects, as the states may not be allocated when the object is not locked.

step

System object: dsp.CrossSpectrumEstimator
Package: dsp

Estimate the cross-spectrum of a signal

Syntax

```
Y = step(sysObj,x)
[Y1,...,YN] = step(sysObj,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(sysObj,x)` processes the input data, `x`, to produce the output, `Y`, from the System object, `sysObj`. `[Y1,...,YN] = step(sysObj,x)` produces `N` outputs.

The columns of `x` are treated as independent channels.

Every System object has a step method. The step method processes the input data according to the object algorithm. The number of input and output arguments depends on the algorithm, and may depend also on one or more property settings. The step method for some objects accepts fixed-point (fi) inputs.

Calling step on an object puts that object into a locked state. When locked, you cannot change nontunable properties or any input characteristics (size, data type and complexity) without reinitializing (unlocking and relocking) the object.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.Crosscorrelator System object

Package: dsp

Cross-correlation of two inputs

Description

The `Crosscorrelator` returns the cross-correlation sequence for two discrete-time deterministic inputs. This object can also return the cross-correlation sequence estimate for two discrete-time, jointly wide-sense stationary (WSS), random processes.

To obtain the cross-correlation for two discrete-time deterministic inputs:

1. Define and set up your cross-correlator. See “Construction” on page 4-494.
2. Call step to compute the cross-correlation according to the properties of `dsp.Crosscorrelator`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`xcorr = dsp.Crosscorrelator` returns a cross-correlator object, `xcorr`, that computes the cross-correlation of two inputs. For N-D arrays, the cross-correlator computes the correlation column-wise. The inputs must have an equal number of columns. If one input is a vector and the other is an N-D array, the cross-correlator computes the cross-correlation of the vector with each column of the N-D array. Cross correlating inputs of length N and M results in a cross-correlation sequence of length $N+M-1$. Cross correlating matrices of size M-by-N and P-by-N results in a matrix of cross-correlation sequences of size $M+P-1$-by-N.

`xcorr = dsp.Crosscorrelator('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a cross-correlator, `xcorr`, with each property set to the specified value.

Properties

Method

Domain for computing correlations

Specify the domain for computing correlation as `Time Domain`, `Frequency Domain`, or `Fastest`. Computing correlations in the time domain minimizes memory use. Computing correlations in the frequency domain may require fewer computations than computing in the time domain depending on the input length. If the value of this property is `Fastest`, the cross-correlator operates in the domain which minimizes the number of computations. The default is `Time Domain`.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of Wrap or Saturate. The default is Wrap. This property applies only if the object is not in full precision mode.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of Full Precision, Same as first input, or Custom. The default is Full Precision.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when the ProductDataType on page 4-496 property is Custom. The default is numerictype([],32,30).

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as Full Precision, Same as product, Same as first input, or Custom. The default is Full Precision.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when the AccumulatorDataType on page 4-496 property is Custom. The default is numerictype([],32,30).

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of Same as accumulator, Same as product, Same as first input, or Custom. The default is Same as accumulator.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `OutputDataType` on page 4-496 property is `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Cross-correlation sequence

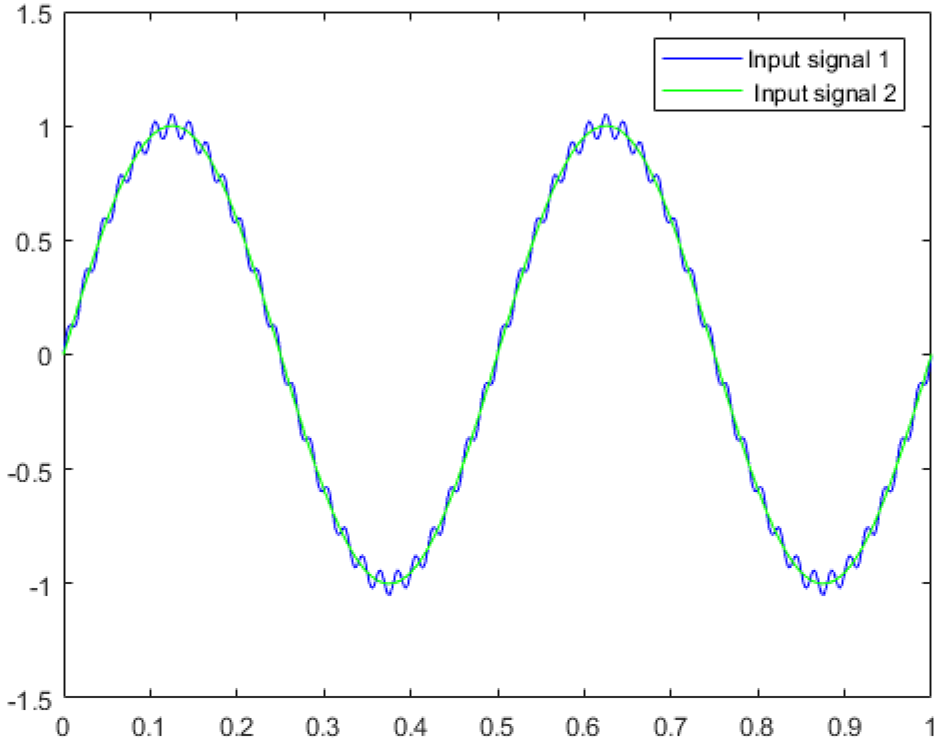
Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

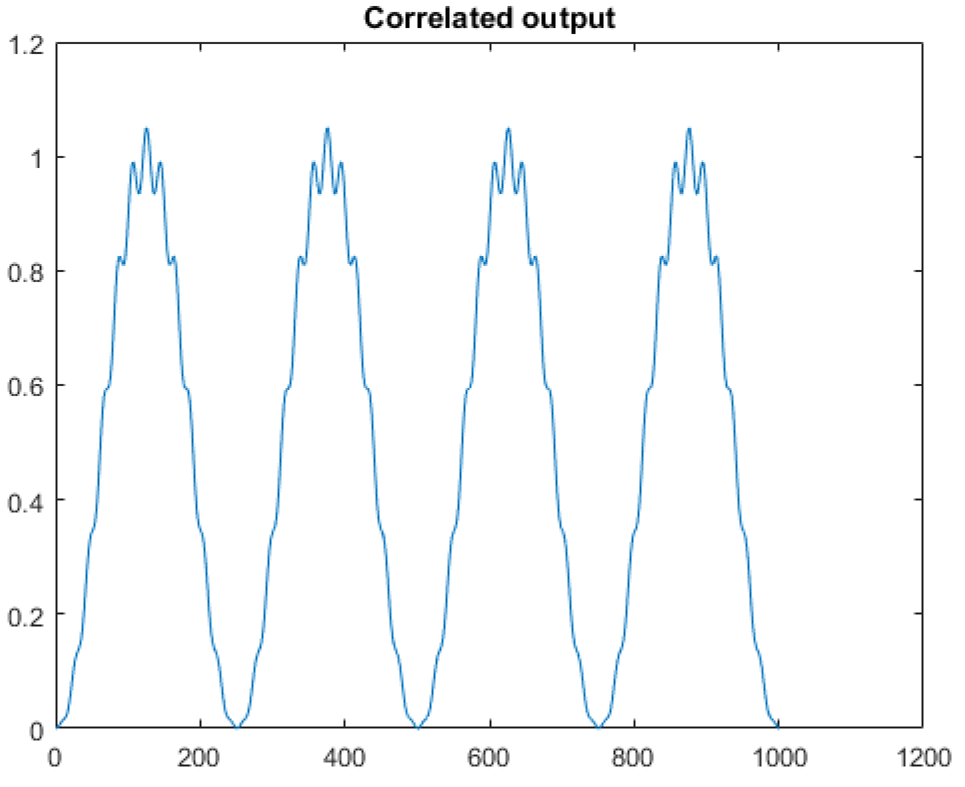
Examples

Compute Correlation Between Two Signals

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
xcorr = dsp.Crosscorrelator;
t = 0:0.001:1;
x1 = sin(2*pi*2*t)+0.05*sin(2*pi*50*t);
x2 = sin(2*pi*2*t);
y = xcorr(x1,x2); %computes cross-correlation of x1 and x2
figure,plot(t,x1,'b',t, x2, 'g');
legend('Input signal 1',' Input signal 2')
figure, plot(y); title('Correlated output')
```


1.5
1
0.5
0
-0.5
-1
-1.5
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Input signal 1
Input signal 2



Cross-Correlation of Input Noise and Delayed Version

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Use cross-correlation to detect delay in jointly stationary white Gaussian noise inputs.

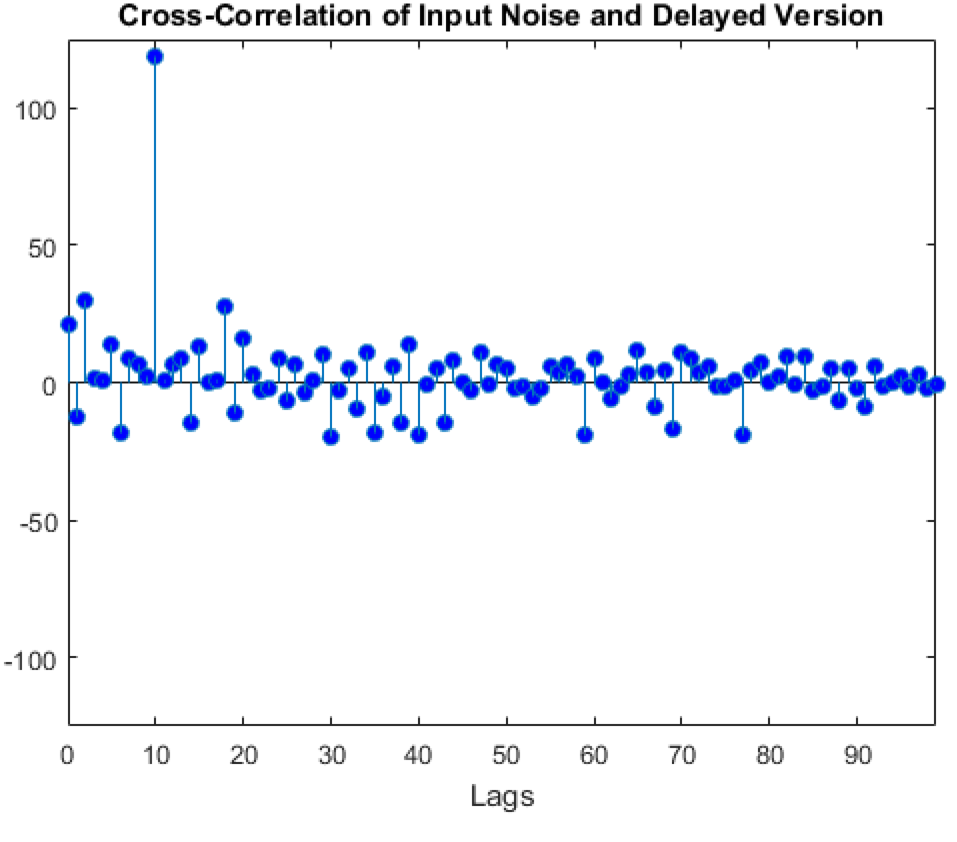
```
S = rng('default');
```

white Gaussian noise input

```
x = randn(100,1);
```

Create copy delayed by 10 samples x1[n] = x[n-10]

```
delay = dsp.Delay(10);
x1 = delay(x);
xcorr = dsp.Crosscorrelator;
y = xcorr(x1,x);
lags = 0:99; %Positive lags
stem(lags,y(100:end),'markerfacecolor',[0 0 1]);
axis([0 99 -125 125]);
xlabel('Lags');
title('Cross-Correlation of Input Noise and Delayed Version');
```



The theoretical cross-correlation sequence is identically zero except at lag 10. Note this is not true in the sample cross-correlation sequence, but the estimate demonstrates a peak at the correct lag.

Definitions

Cross-Correlation

Cross-correlation is the measure of similarity of two discrete-time sequences as a function of the lag of one relative to the other.

For two length N deterministic inputs, or realizations of jointly WSS random processes, x and y, the cross-correlation is computed using the following relationship:

$$r_{xy}(h) = \begin{cases} \sum_{n=0}^{N-h-1} x(n+h)y^*(n) & 0 \le h \le N-1 \\ r_{yx}^*(h) & -(N-1) \le h \le 0 \end{cases}$$

where h is the lag and * denotes the complex conjugate. If the inputs are realizations of jointly WSS stationary random processes, $r_{xy}(h)$ is an unnormalized estimate of the theoretical cross-correlation:

$$\rho_{xy}(h) = E\{x(n+h)y^*(n)\}$$

where E{ } is the expectation operator.

Algorithms

Time domain Computation

When you set the computation domain to time, the algorithm computes the cross-correlation of two signals in the time domain. The input signals can be fixed-point signals in this domain.

Correlate Two 2-D Arrays

When the inputs are two 2-D arrays, the jth column of the output, $y_{u,v}$, has these elements:

$$y_{uv(i,j)} = \sum_{k=0}^{\max(M_u,M_v)-1} u^{*}_{k,j} v_{(k+i),j} \qquad 0 \le i < M_v$$

$$y_{uv(i,j)} = y^{*}_{vu(-i,j)} \qquad -M_u < i < 0$$

- * denotes the complex conjugate.
- u is an M_u-by-N input matrix.
- v is an M_v-by-N input matrix.
- $y_{u,v}$ is an $(M_u + M_v - 1)$-by-N matrix.

Inputs u and v are zero when indexed outside their valid ranges.

Correlate a Column Vector with a 2-D Array

When one input is a column vector and the other input is a 2-D array, the algorithm independently cross-correlates the input vector with each column of the 2-D array. The jth column of the output, $y_{u,v}$, has these elements:

$$y_{uv(i,j)} = \sum_{k=0}^{\max(M_u,M_v)-1} u^{*}_{k} v_{(k+i),j} \qquad 0 \le i < M_v$$

$$y_{uv(i,j)} = y^{*}_{vu(-i,j)} \qquad -M_u < i < 0$$

- * denotes the complex conjugate.
- u is an M_u-by-1 column vector.
- v is an M_v-by-N matrix.
- $y_{u,v}$ is an $(M_u + M_v - 1)$-by-N matrix.

Inputs u and v are zero when indexed outside their valid ranges.

Correlate Two Column Vectors

When the inputs are two column vectors, the jth column of the output, $y_{u,v}$, has these elements:

$$y_{uv(i)} = \sum_{k=0}^{\max(M_u,M_v)-1} u_k^* v_{(k+i)} \qquad 0 \le i < M_v$$

$$y_{uv(i)} = y_{vu(-i)}^* \qquad -M_u < i < 0$$

- * denotes the complex conjugate.
- u is an M_u-by-1 column vector.
- v is an M_v-by-1 column vector.
- $y_{u,v}$ is an $(M_u + M_v - 1)$-by-1 column vector.

Inputs u and v are zero when indexed outside their valid ranges.

Frequency Domain Computation

When you set the computation domain to frequency, the algorithm computes the cross-correlation in the frequency domain.

To compute the cross-correlation, the algorithm:

1. Takes the Fourier transform of both input signals, U and V.
2. Multiplies U and V^*, where * denotes the complex conjugate.
3. Computes the inverse Fourier transform of the product.

In this domain, depending on the input length, the algorithm can require fewer computations.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.Convolver | dsp.Autocorrelator

Introduced in R2012a

step

System object: dsp.Crosscorrelator
Package: dsp

Cross-correlation sequence

Syntax

```
Y = step(xcorr,A,B)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(xcorr,A,B)` computes the cross-correlation of `A` and `B` and returns the result in `Y`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.CumulativeProduct System object

Package: dsp

Cumulative product of channel, column, or row elements

Description

The `CumulativeProduct` object computes the cumulative product of channel, column, or row elements.

To compute the cumulative product of channel, column, or row elements:

1. Define and set up your cumulative product object. See "Construction" on page 4-506.
2. Call step to compute the cumulative product according to the properties of `dsp.CumulativeProduct`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`cprod = dsp.CumulativeProduct` returns a cumulative product object, `cprod`, that computes the cumulative product of input matrix or input vector elements along the default Dimension on page 4-507.

`cprod = dsp.CumulativeProduct('PropertyName',PropertyValue,...)` returns a cumulative product object, `cprod`, with each specified property set to the specified value.

Properties

Dimension

Computation dimension for cumulative product

Specify the computation dimension as one of | `Channels (running product)` | `Rows` | `Columns` |. The default is `Channels (running product)`.

ResetInputPort

Enable resetting cumulative product via input port

Set this property to `true` to enable resetting the cumulative product. When you set this property to `true`, specify a reset signal to the `step` method to reset the cumulative product. You can access this property when the `Dimension on page 4-507` property is set to `Channels (running product)`. The default is `false`.

ResetCondition

Reset condition for cumulative product

Specify the event on the reset input port that causes resetting the cumulative product to one of | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero` |. This property applies when you set the `ResetInputPort on page 4-507` property to `true` and the `Dimension on page 4-507` property to `Channels (running product)`. The default is `Rising edge`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent` | `Floor` | `Nearest`, | `Round` | `Simplest` | `Zero` |. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | `Wrap` | `Saturate` |. The default is `Wrap`.

IntermediateProductDataType

Intermediate product word and fraction lengths

Specify the intermediate product fixed-point data type as one of `| Same as input | Custom |`. The default is `Same as input`.

CustomIntermediateProductDataType

Intermediate product word and fraction lengths

Specify the intermediate product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `IntermediateProductDataType on page 4-508` property to `Custom`. The default is `numerictype([], 16, 15)`.

ProductDataType

Product output word and fraction lengths

Specify the product output fixed-point data type as one of `| Same as input | Custom |`. The default is `Same as input`.

CustomProductDataType

Custom product output word and fraction lengths

Specify the product output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType on page 4-508` property to `Custom`. The default is `numerictype([], 32, 30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `| Same as product output | Same as input | Custom |`. The default is `Same as input`.

CustomAccumulatorDataType

Custom accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType on page 4-508` property to `Custom`. The default is `numerictype([], 32, 30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of | Same as product output | Same as input | Custom |. The default is Same as input.

CustomOutputDataType

Custom output word and fraction lengths

Specify the output fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the OutputDataType on page 4-509 property to Custom. The default is numerictype([], 16, 15).

Methods

reset	Reset running cumulative product
step	Cumulative product of input along specified dimension for input

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Cumulative Product of a Matrix

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent step syntax. For example, myObject(x) becomes step(myObject,x).

Use the dsp.CumulativeProduct object to compute the cumulative product of a matrix.

```
cprod = dsp.CumulativeProduct;
x = magic(2)
y = cprod(x)
```

```
x =

     1     3
     4     2


y =

     1     3
     4     6
```

The cumulative product is computed column-wise along each channel.

Algorithms

This object implements the algorithm, inputs, and outputs described on the Cumulative Product block reference page. The object properties correspond to the block parameters, except:
The **Reset port** block parameter corresponds to both the ResetCondition and ResetInputPort object properties.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

dsp.CumulativeSum

Introduced in R2012a

reset

System object: dsp.CumulativeProduct
Package: dsp

Reset running cumulative product

Syntax

```
reset(cprod)
```

Description

`reset(cprod)` resets the running cumulative product for object `cprod` to zero.

step

System object: dsp.CumulativeProduct
Package: dsp

Cumulative product of input along specified dimension for input

Syntax

```
Y = step(cprod,X)
Y = step(cprod,X,R)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(cprod,X)` computes the cumulative product along the specified dimension for the input `X`.

`Y = step(cprod,X,R)` resets the cumulative product object's state based on the `ResetCondition` property value and the value of the reset signal, `R` when the `ResetInputPort` property is `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.CumulativeSum System object

Package: dsp

Cumulative sum of channel, column, or row elements

Description

The `CumulativeSum` object computes the cumulative sum of channel, column, or row elements.

To compute the cumulative sum of channel, column, or row elements:

1. Define and set up your cumulative sum object. See "Construction" on page 4-514.
2. Call step to compute the cumulative sum according to the properties of `dsp.CumulativeSum`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`csum = dsp.CumulativeSum` returns a cumulative sum System object, `csum`, which computes the running cumulative sum for each channel in the input.

`csum = dsp.CumulativeSum('PropertyName',PropertyValue,...)` returns a cumulative sum object, `csum`, with each specified property set to the specified value.

Properties

`Dimension`

Computation dimension for cumulative sum

Specify the computation dimension as one of | `Channels (running sum)` | `Rows` | `Columns` |. The default is `Channels (running sum)`.

ResetInputPort

Enable resetting cumulative sum via input port.

Set this property to `true` to enable resetting the cumulative sum. When you set this property to `true`, you also specify a reset input to the `step` method to reset the cumulative sum. The default is `false`.

ResetCondition

Reset condition for cumulative sum

Specify the event on the reset input port that resets the cumulative sum as one of | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero` |. This property applies when you set the `ResetInputPort on page 4-515` property to `true`. The default is `Rising edge`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent` | `Floor`, `Nearest`, `Round`, `Simplest` | `Zero` |. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | `Wrap` | `Saturate` |. The default is `Wrap`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of | `Same as input` | `Custom` |. The default is `Same as input`.

CustomAccumulatorDataType

Custom accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType on page 4-515` property to `Custom`. The default is `numerictype([], 32, 30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of one of `| Same as accumulator | Same as input | Custom |`. The default is `Same as accumulator`.

CustomOutputDataType

Custom output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType on page 4-516` property to `Custom`. The default is `numerictype([], 16, 15)`.

Methods

reset	Reset internal states of cumulative sum object to zero
step	Cumulative sum along specified dimension for input

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Cumulative Sum of a Matrix

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Use the `dsp.CumulativeSum` object to compute the cumulative sum of a matrix.

```
csum = dsp.CumulativeSum;
x = magic(2)
y = csum(x)
```

```
x =

     1     3
     4     2


y =

     1     3
     5     5
```

The cumulative sum is computed column-wise along each channel.

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Cumulative Sum` block reference page. The object properties correspond to the block properties, except:
The **Reset port** block parameter corresponds to both the `ResetCondition` and the `ResetInputPort` object properties

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.CumulativeProduct

Introduced in R2012a

reset

System object: dsp.CumulativeSum
Package: dsp

Reset internal states of cumulative sum object to zero

Syntax

```
reset(csum)
```

Description

`reset(csum)` sets the states for the running cumulative sum object, `csum`, to zero when the `Dimension` property is set to `Channels (running sum)`.

step

System object: dsp.CumulativeSum
Package: dsp

Cumulative sum along specified dimension for input

Syntax

```
Y = step(csum,X)
Y = step(csum,X,R)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(csum,X)` computes the cumulative sum along the specified dimension for the input `X`.

`Y = step(csum,X,R)` resets the System object state based on the `ResetCondition` property value and the value of the reset signal, `R`. You can only reset the state if the `ResetInputPort` property is `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.DCT System object

Package: dsp

Discrete cosine transform (DCT)

Description

The `DCT` object computes the discrete cosine transform (DCT) of input.

To compute the DCT of input:

1. Define and set up your DCT object. See “Construction” on page 4-521.
2. Call step to compute the DCT according to the properties of `dsp.DCT`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`dct = dsp.DCT` returns a discrete cosine transform (DCT) object, `dct`, used to compute the DCT of a real or complex input signal.

`dct = dsp.DCT('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a DCT object, `dct`, with each property set to the specified value.

Properties

`SineComputation`

Method to compute sines and cosines

Specify how the DCT object computes the trigonometric values as `Trigonometric function` or `Table lookup`. This property must be set to `Table lookup` for fixed-point inputs. The default is `Table lookup`.

Fixed-Point Properties

`RoundingMethod`

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling`, `Convergent`, `Floor` , `Nearest`, `Round`, `Simplest`, or `Zero`. This property applies when you set the `SineComputation` on page 4-521 property to `Table lookup`.

`OverflowAction`

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` or `Saturate`. This property applies when you set the `SineComputation` on page 4-521 property to `Table lookup`.

`SineTableDataType`

Sine table word-length designation

Specify the sine table fixed-point data type as one of `Same word length as input` or `Custom`. This property applies when you set the `SineComputation` on page 4-521 property to `Table lookup`.

`CustomSineTableDataType`

Sine table word length

Specify the sine table fixed-point type as an unscaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SineComputation` on page 4-521 property to `Table lookup` and the `SineTableDataType` on page 4-522 property to `Custom`. The default is `numerictype([],16)`.

`ProductDataType`

Product word and fraction lengths

Specify the product fixed-point data type as one of `Full precision` , `Same as input`, `Custom`. This property applies when you set the `SineComputation` on page 4-521 property to `Table lookup`.

`CustomProductDataType`

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SineComputation` on page 4-521 property to `Table lookup` and the `ProductDataType` on page 4-522 property to `Custom`. The default is `numerictype([],32,30)`.

`AccumulatorDataType`

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Full precision`, `Same as input`, `Same as product`, or `Custom`. This property applies when you set the `SineComputation` on page 4-521 property to `Table lookup`.

`CustomAccumulatorDataType`

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SineComputation` on page 4-521 property to `Table lookup` and the `AccumulatorDataType` on page 4-523 property to `Custom`. The default is `numerictype([],32,30)`.

`OutputDataType`

Output word and fraction lengths

Specify the output fixed-point data type as one of `Full precision` , `Same as input`, `Custom`. This property applies when you set the `SineComputation` on page 4-521 property to `Table lookup`. The default is `Full precision`.

`CustomOutputDataType`

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SineComputation` on page 4-521 property to `Table lookup` and the `OutputDataType` on page 4-523 property to `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Discrete cosine transform (DCT) of input

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Analyze the Energy Content in a Sequence

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Use DCT to analyze the energy content in a sequence:

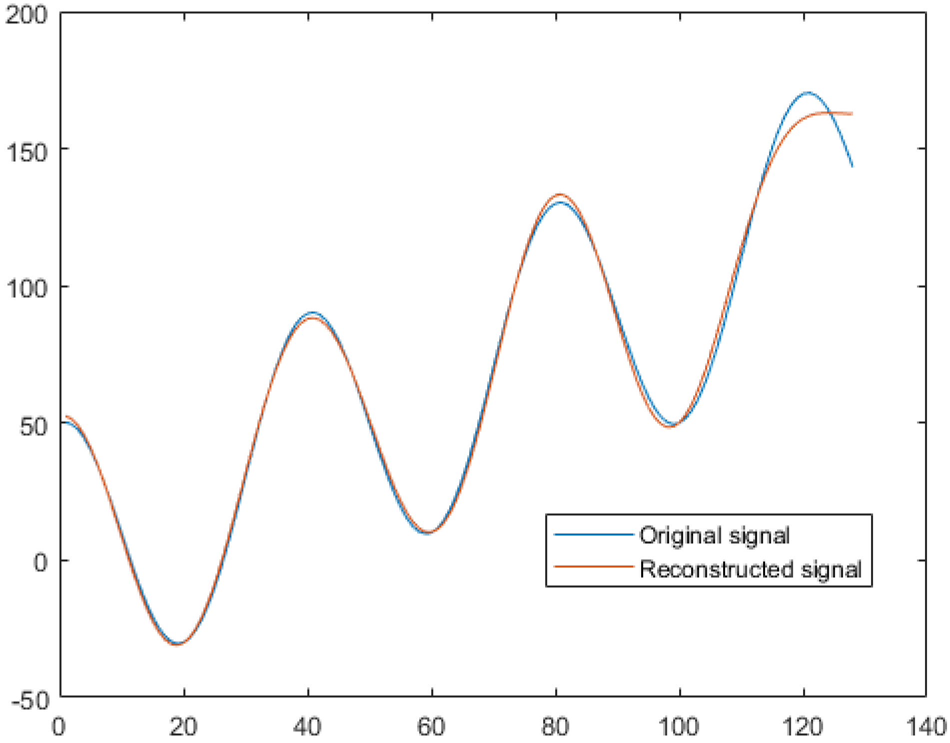
```
x = (1:128).' + 50*cos((1:128).'*2*pi/40);
dct = dsp.DCT;
X = dct(x);
```

Set the DCT coefficients which represent less than 0.1% of the total energy to 0 and reconstruct the sequence using IDCT.

```
[XX, ind] = sort(abs(X),1,'descend');
ii = 1;
while (norm([XX(1:ii);zeros(128-ii,1)]) <= 0.999*norm(XX))
    ii = ii+1;
end
disp(['Number of DCT coefficients that represent 99.9%',...
    'of the total energy in the sequence: ',num2str(ii)]);
XXt = zeros(128,1);
```

```
XXt(ind(1:ii)) = X(ind(1:ii));
idct = dsp.IDCT;
xt = idct(XXt);
plot(1:128,[x xt]);
legend('Original signal','Reconstructed signal',...
    'location','best');
```

```
Number of DCT coefficients that represent 99.9%of the total energy in the sequence: 10
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `DCT` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.FFT | dsp.IDCT | dsp.IFFT

Introduced in R2012a

step

System object: dsp.DCT
Package: dsp

Discrete cosine transform (DCT) of input

Syntax

`Y = step(dct,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(dct,X)` computes the `DCT` of the input `X`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.Delay System object

Package: dsp

Delay input by specified number of samples or frames

Description

Note: Certain features of this object will be removed in future releases. See "Functionality being removed or replaced for blocks and System objects".

The `Delay` object delays an input by a specified number of samples or frames.

To delay an input by a specified number of samples or frames:

1. Define and set up your delay object. See "Construction" on page 4-528.
2. Call step to delay the input according to the properties of `dsp.Delay`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`delay = dsp.Delay` returns a delay object, `delay`, to delay the input by one sample.

`delay = dsp.Delay('`*`PropertyName`*`',` *`PropertyValue`*`, ...)` returns a delay object, `delay`, with each property set to the specified value.

`delay = dsp.Delay(LEN,'`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a delay object, `delay`, with the `Length` on page 4-529 property set to `LEN` and other specified properties set to the specified values.

Properties

`Units`

Delay units as samples or frames

Specify the delay units as one of | `Samples` | `Frames` . The default is `Samples`.

`Length`

Amount of delay

Specify amount of delay to apply to the input signal. The value of this property can be one of the following:

- A scalar non-negative integer by which to delay all channels equally.
- A vector of non-negative integers whose length is equal to the number of input columns (channels).

The default is 1.

`InitialConditionsPerChannel`

Enable different initial conditions per channel

Set this property to `true` to specify different initial conditions per channel. The default value is `false`.

`InitialConditionsPerSample`

Enable different initial conditions per sample

Set this property to `true` to specify different initial conditions per sample. The default value is `false`.

`InitialConditions`

Initial output of delay object

Specify the initial output(s) of the delay object. You can set this property to a scalar, vector, matrix, or a cell array depending on the values of the `InitialConditionsPerChannel` on page 4-529, `InitialConditionsPerSample` on page 4-529, and `Units` on page 4-529 properties. The default is `0`.

If the input is an *M*-by-*N* matrix, the dimensions of this property value must be as follows:

- If the `InitialConditionsPerChannel` and `InitialConditionsPerSample` properties are both `false`, the property value must be a scalar.
- If the `InitialConditionsPerChannel` property is `true` and the `InitialConditionsPerSample` property is `false`, the value must be a vector of length *N*.
- If the `InitialConditionsPerChannel` property is `false`, the `InitialConditionsPerSample` property is `true`, and the `Units` property is `Frames`, the value must be a vector of length equal to the product of *M* and the `Length` property value.
- If the `InitialConditionsPerChannel` property is `false`, the `InitialConditionsPerSample` property is `true`, and the `Units` property is `Samples`, the value must be a vector of length equal to the `Length` property value.
- If the `InitialConditionsPerChannel` and `InitialConditionsPerChannel` properties are both `true`, the property value must be a cell array of size *N*. Each cell of the cell array contains the delay values for one channel.
- If the `Units` property is `Frames`, each cell must be a vector of length equal to the product of *M* and the `Length` property value.
- If the `Units` property is `Samples`, each cell must be a vector of length equal to the `Length` property value.

ResetInputPort

Enable resetting delay states

Specify when the delay object should reset the delay states. By default, the value of this property is `false`, and the object does not reset the delay states. When this property is set to `true`, a reset control input is provided to the `step` method, and the `ResetCondition` on page 4-530 property applies. The object resets the delay states based on the values of the `ResetCondition` property and the reset input to the `step` method.

ResetCondition

Reset trigger setting for delay

Specify the event to reset the delay as one of | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero` |. The delay object resets the delay based on the values of this property and the reset input to the `step` method. This property applies when you set the `ResetInputPort` on page 4-530 property to `true`. The default is `Non-zero`.

Methods

reset	Reset delay states
step	Apply delay to input

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Delay Input

Delay input by five samples using the `dsp.Delay` System object™.

```
delay = dsp.Delay(5);
y = delay((1:10)');
display(y);
```

```
y =

     0
     0
     0
     0
     0
     1
     2
     3
     4
```

```
5
```

The output `y` is delayed by 5 samples.

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Delay` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.VariableIntegerDelay | dsp.VariableFractionalDelay. | Delay

Introduced in R2012a

reset

System object: dsp.Delay
Package: dsp

Reset delay states

Syntax

```
reset(delay)
```

Description

`reset(delay)` resets the states of the delay object `delay` to the values specified in the `InitialConditions` property.

After you invoke the `step` method for a nonzero input, the delay object states may change and invoking the `step` method again without invoking the `reset` method may produce different outputs for identical inputs.

Reset the States of the Delay Object

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Delay input by 5 samples.

```
delay = dsp.Delay(5);
y = delay((1:10)')
```

```
y =

     0
     0
     0
     0
     0
```

```
     1
     2
     3
     4
     5
```

Delay the input without calling reset.

```
y1 = delay((1:10)')
```

```
y1 =

     6
     7
     8
     9
    10
     1
     2
     3
     4
     5
```

Reset states.

```
reset(delay);
y2 = delay((1:10)')
```

```
y2 =

     0
     0
     0
     0
     0
     1
     2
     3
     4
     5
```

step

System object: dsp.Delay
Package: dsp

Apply delay to input

Syntax

```
Y = step(delay,X)
Y = step(delay,X,R)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(delay,X)` adds delay to input `X` to return output `Y`.

`Y = step(delay,X,R)` adds delay to `X` , and selectively resets the delay object's state based on the value of reset input `R` and the value of the `ResetCondition` property. You can use this option only when you set the `ResetInputPort` property to `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.DelayLine System object

Package: dsp

Rebuffer sequence of inputs with one-sample shift

Description

The `DelayLine` object rebuffers a sequence of inputs with one-sample shift.

To rebuffer a sequence of inputs with one-sample shift:

1. Define and set up your delay line object. See "Construction" on page 4-536.
2. Call step to rebuffer the sequence of inputs according to the properties of `dsp.DelayLine`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`dline = dsp.DelayLine` returns a delay line System object, `dline`, that buffers the input samples into a sequence of overlapping or underlapping matrix outputs.

`dline = dsp.DelayLine('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a delay line System object, `dline`, with each specified property set to the specified value.

`dline = dsp.DelayLine(DELAYSIZE,INITIAL,'`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a delay line System object, `dline`, with the `Length` property set to `DELAYSIZE`, `InitialConditions` property set to `INITIAL` and other specified properties set to the specified values.

Properties

`Length`

Number of rows in output matrix

Specify the number of rows in the output matrix as a scalar positive integer. The default is `64`.

`InitialConditions`

Initial delay line output

Set the value of the object's initial output as one of `| scalar | vector | matrix |`.

For vector outputs, the following selections apply for the `InitialConditions` property:

- A vector of the same size
- A scalar value that you want repeated across all elements of the initial output

For matrix outputs, the following selections apply for the `InitialConditions` property:

- A matrix of the same size
- A vector (equal to the length of the number of matrix rows) that repeats across all columns of the initial output
- A scalar that repeats across all elements of the initial output

The default is `0`.

`DirectFeedthrough`

Enable passing input data to output without extra frame delay

When you set this property to `true`, there is no input data delay by an extra frame before it is available at the output buffer. Instead, the input data is available immediately at the output. When you set this property to `false`, there is one frame delay on the output. The default is `false`.

`EnableOutputInputPort`

Enable selective output linearization

The object internally uses a circular buffer, even though the output is linear. To obtain a valid output, the object must linearize the circular buffer. When this property is `true`, the object uses an additional Boolean input to determine if a valid output calculation is needed. If Boolean the input value is `1`, the object's output is linearized and thus valid. If Boolean the input value is `0`, the output is not linearized and is invalid. This allows the object to be more efficient when each step does not require the tapped delay line output. When you set this property to `false`, the output is always linearized and valid. The default is `false`.

`HoldPreviousValue`

Hold previous valid value for invalid output

If you set this property to `true`, the most recent, valid value is held on the output. If you set this property to `false`, the signal on the output is invalid data. This property applies only when you set the `EnableOutputInputPort on page 4-537` property to `true`. The default is `false`.

Methods

reset	Reset internal states of delay line object
step	Delayed version of input

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Delay Line

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Use a delay line object with a delay line size of 4 samples.

```
delayline = dsp.DelayLine( ...
    'Length', 4, ...
    'DirectFeedthrough', true, ...
    'InitialConditions', -2, ...
    'EnableOutputInputPort', true, ...
    'HoldPreviousValue', true);
en = logical([1 1 0 1 0]);
y = zeros(4,5);
for ii = 1:5
    y(:,ii) = delayline(ii, en(ii));
end
display(y);
```

```
y =

    -2    -2    -2     1     1
    -2    -2    -2     2     2
    -2     1     1     3     3
     1     2     2     4     4
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Delay Line` block reference page. The object properties correspond to the block properties, except as noted.

This object processes inputs as separate channels (frames). The corresponding block has a temporary **Treat Mx1 and unoriented sample-based signals as** parameter with a `One channel` option for frame-based behavior and an `M channels` option for sample-based behavior. See the Delay Line block reference page for more information.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.Delay

Introduced in R2012a

reset

System object: dsp.DelayLine
Package: dsp

Reset internal states of delay line object

Syntax

```
reset(dline)
```

Description

`reset(dline)` sets the internal delay buffers of the `DelayLine` object `dline` to their initial conditions.

step

System object: dsp.DelayLine
Package: dsp

Delayed version of input

Syntax

```
Y = step(dline,X)
Y = step(dline,X,EN)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(dline,X)` returns the delayed version of input `X`. `Y` is an output matrix with the same number of rows as the delay line size. Each column of `X` is treated as a separate channel.

The System object rebuffers a sequence of *Mi*-by-*N* matrix inputs into a sequence of *Mo*-by-*N* matrix outputs, where *Mo* is the output frame size specified by the `Length` property. Depending on whether *Mo* is greater than, less than, or equal to the input frame size, *Mi*, the output frames can be underlapped or overlapped. Each of the *N* input channels is rebuffered independently:

- When*Mo* > *Mi*, the output frame overlap is the difference between the output and input frame size, *Mo*–*Mi*.
- When *Mo* < *Mi*, the output is underlapped; the object discards the first *Mi*–*Mo* samples of each input frame so that only the last *Mo* samples are buffered into the corresponding output frame.
- When *Mo* = *Mi*, the output data is identical to the input data, but is delayed by the latency of the object.

`Y = step(dline,X,EN)` selectively outputs the delayed version of input `X` depending on the Boolean input `EN`. This occurs only when you set the `EnableOutputInputPort` property to `true`. If `EN` is `false`, use the `HoldPreviousValue` property to specify if the object should hold the previous output value(s).

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.Differentiator System object

Package: dsp

Direct form FIR fullband differentiator filter

Description

`dsp.Differentiator` applies a fullband differentiator filter on the input signal to differentiate all its frequency components. This object uses an FIR equiripple filter design to design the differentiator filter. The ideal frequency response of the differentiator is $D(\omega) = j\omega$ for $-\pi \leq \omega \leq \pi$. You can design the filter with minimum order with a specified order. This object supports fixed-point operations.

To filter each channel of your input:

1. Define and set up your differentiator. See "Construction" on page 4-544.
2. Call step to filter each channel of the input signal according to the properties of `dsp.Differentiator`. The input signal can be a real-valued or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The number of rows in the input signal denotes the channel length. The data type of the input can be double, single, or fixed-point data type. The number of channels cannot change between calls to the `step` method. The data type characteristics (double, single, or fixed-point) and the real-complex characteristics (real or complex valued) must be the same for the input data and output data.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`DF = dsp.Differentiator` returns a differentiator, `DF`, with the default settings. Calling `step` with the default property settings filters the input data with a minimum-order filter that has a passband ripple of 0.1 dB.

`DF = dsp.Differentiator(Name,Value)` sets each property Name to the specified Value. Unspecified properties have default values.

Properties

`DesignForMinimumOrder` — Design minimum order filter

`true` (default) | `false`

Option to design a minimum-order filter, specified as a logical scalar. The filter has 2 degrees of freedom. When you set this property to

- `true` — The object designs the filter with the minimum order that meets the `PassbandRipple` value.
- `false` — The object designs the filter with order that you specify in the `FilterOrder` property.

This property is not tunable.

`FilterOrder` — Order of the filter

31 (default) | odd positive integer

Order of the filter, specified as an odd positive integer. You can specify the filter order only when `'DesignForMinimumOrder'` is set to `false`.

This property is not tunable.

`PassbandRipple` — Maximum passband ripple

0.1 (default) | positive real scalar

Maximum passband ripple in dB, specified as a positive real scalar. You can specify the passband ripple only when `'DesignForMinimumOrder'` is set to `true`.

This property is not tunable.

`ScaleCoefficients` — Scale filter coefficients

false (default) | true

Option to scale the filter coefficients, specified as a logical scalar. When you set this property to `true`, the object scales the filter coefficients to preserve the input dynamic range.

This property is not tunable.

Fixed-Point Properties

`CoefficientsDataType` — Word and fraction lengths of coefficients

`numerictype(1,16)` (default) | `numerictype` object

Word and fraction lengths of coefficients, specified as a signed or unsigned `numerictype` object. The default, `numerictype(1,16)`, corresponds to a signed numeric type object with 16-bit coefficients. To give the best possible precision, the fraction length is computed based on the coefficient values.

This property is not tunable.

The word length of the output is the same as the word length of the input. The object computes the fraction length of the output such that the entire dynamic range of the output can be represented without overflow. For details on how the object computes the fraction length of the output, see "Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters".

`RoundingMethod` — Rounding method for output fixed-point operations

`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`

Rounding method for output fixed-point operations, specified as a character vector. For more information on the rounding modes, see "Precision and Range".

This property is not tunable.

Examples

Group Delay Estimation

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Estimate the group delay of a linear phase FIR filter using a `dsp.TransferFunctionEstimator` object followed by `dsp.PhaseExtractor` and `dsp.Differentiator` objects. The group delay of a linear phase FIR filter is given by $GD = -(d\theta(\omega)/d\omega) = -\frac{N}{2}$, where $\theta(\omega)$ is the phase information of the filter, ω is the frequency vector, and N is the order of the filter.

Set Up the Objects

Create a linear phase FIR lowpass filter. Set the order to 200, the passband frequency to 255 Hz, the passband ripple to 0.1 dB, and the stopband attenuation to 80 dB. Specify a sample rate of 512 Hz.

```
Fs = 512;
LPF = dsp.LowpassFilter('SampleRate',Fs,'PassbandFrequency',255,...
    'DesignForMinimumOrder',false,'FilterOrder',200);
```

To estimate the transfer function of the lowpass filter, create a transfer function estimator. Specify the window to be `Hann`. Set the FFT length to 1024 and the number of spectral averages to 200.

```
TFE = dsp.TransferFunctionEstimator('FrequencyRange','twosided',...
    'SpectralAverages',200,'FFTLengthSource','Property',...
    'FFTLength',1024);
```

To extract the unwrapped phase from the frequency response of the filter, create a phase extractor.

```
PE = dsp.PhaseExtractor;
```

To differentiate the phase θ, create a differentiator filter. This value is used in computing the group delay.

```
DF = dsp.Differentiator;
```

To smoothen the inputm create a variable bandwidth FIR filter.

```
Gain1 = 512/pi;
Gain2 = -1;
VBFilter = dsp.VariableBandwidthFIRFilter('CutoffFrequency',10,...
    'SampleRate',Fs);
```

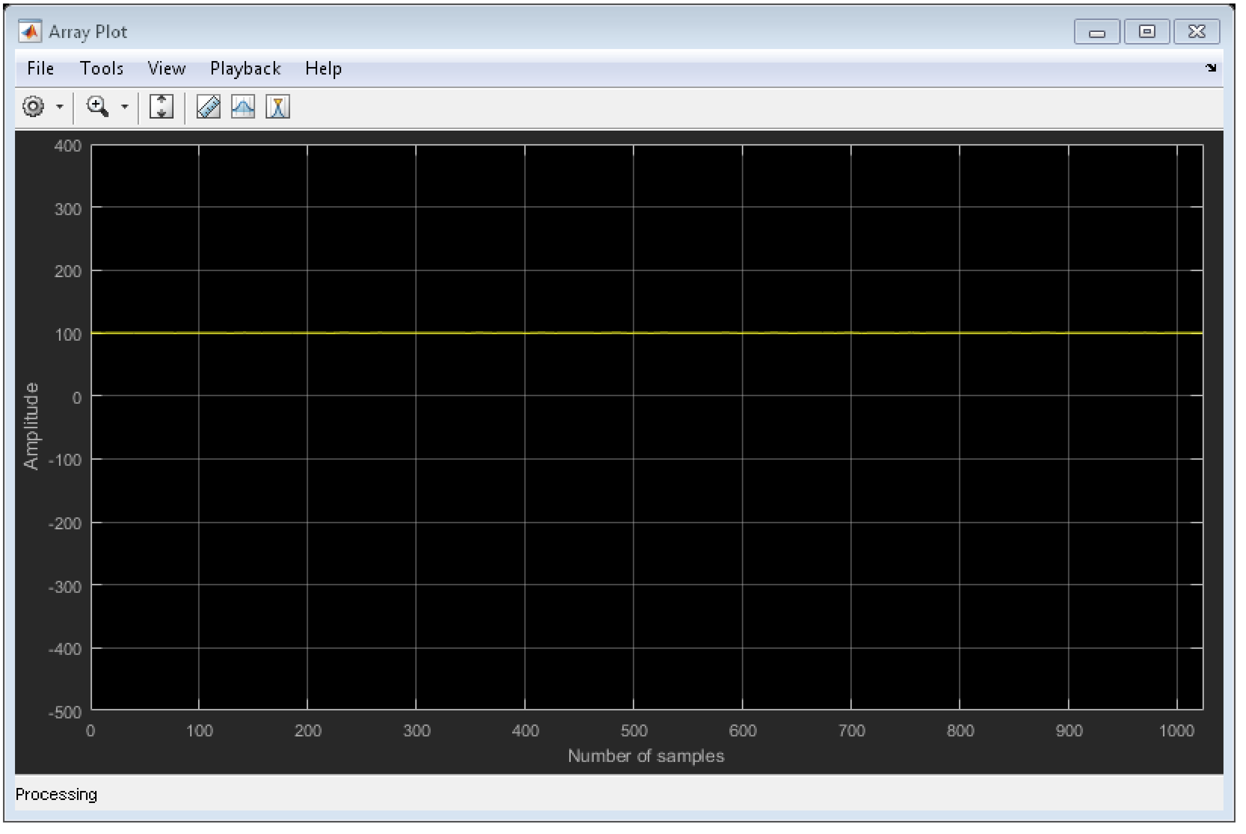
To view the group delay of the filter, create an array plot object.

```
AP = dsp.ArrayPlot('PlotType','Line','YLimits',[-500 400],...
    'YLabel','Amplitude','XLabel','Number of samples');
```

Run the Algorithm

The `for`-loop is the streaming loop that estimates the group delay of the filter. In the loop, the algorithm filters the input signal, estimates the transfer function of the filter, and differentiates the phase of the filter to compute the group delay.

```
Niter = 1000; % Number of iterations
for k = 1:Niter
        x = randn(512,1);  % Input signal = white Gaussian noise
        y = LPF(x);   % Filter noise with Lowpass FIR filter
        H = TFE(x,y); % Compute transfer function estimate
        Phase = PE(H); % Extract the Unwrapped phase
        phaseaftergain1 = Gain1*Phase;
        DiffOut = DF(phaseaftergain1); % Differentiate the phase
        phaseaftergain2 = Gain2 * DiffOut;
        VBFOut = VBFilter(phaseaftergain2); % Smooth the group delay
        AP(VBFOut); % Display the group delay
end
```



As you can see, the group delay of the lowpass filter is 100.

Methods

step	Differentiate input signal with respect to time
getFilter	Get underlying FIR filter
reset	Reset internal states of differentiator

For additional methods, see "Analysis Methods for Filter System Objects" on page 3-2.

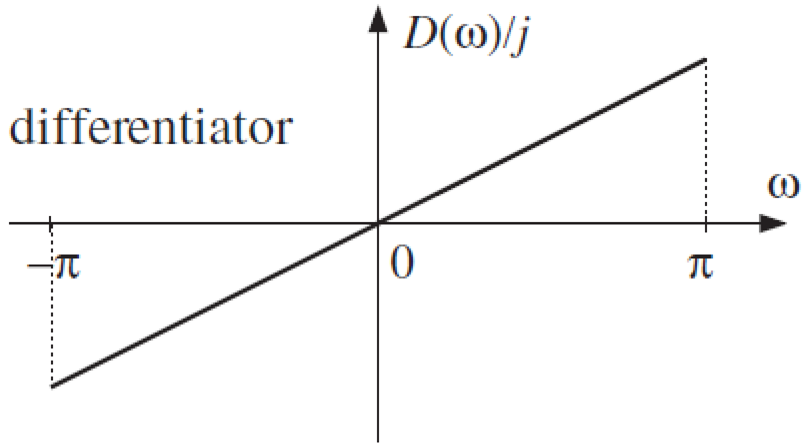
For a complete list of analysis methods supported for the `dsp.Differentiator` object, enter `dsp.Differentiator.helpFilterAnalysis` at the MATLAB command prompt.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Algorithms

Differentiator Filter

Differentiator computes the derivative of a signal. The frequency response of an ideal differentiator filter is given by $D(\omega) = j\omega$, defined over the Nyquist interval $-\pi \leq \omega \leq \pi$.



The frequency response is antisymmetric and is linearly proportional to the frequency.

`dsp.Differentiator` object acts as a differentiator filter. This object condenses the two-step process into one. For the minimum order design, the object uses generalized Remez FIR filter design algorithm. For the specified order design, the object uses the Parks-McClellan optimal equiripple FIR filter design algorithm. The filter is designed as a linear phase Type-IV FIR filter with a Direct form structure.

The ideal differentiator has an antisymmetric impulse response given by $d(n) = -d(-n)$. Hence $d(0) = 0$. The differentiator must have zero response at zero frequency.

Linear-Phase FIR Differentiator Filter

The impulse response of an antisymmetric linear-phase FIR filter is given by $h(n) = -h(M-1-n)$, where M is the length of the filter. Because the filter is antisymmetric, you can use this type of FIR filter to design the linear-phase FIR differentiators.

Consider the design of linear-phase FIR differentiators based on the Chebyshev approximation criterion.

If M is odd, the real-valued frequency response of the FIR filter, $H_r(\omega)$, has the characteristics that $H_r(0) = 0$ and $H_r(\pi) = 0$. This filter satisfies the condition of zero response at zero frequency. However, it is not fullband because $H_r(\pi) = 0$. This differentiator has a linear response over the limited frequency range $[0\ 2\pi f_p]$, where f_p is the bandwidth of the differentiator. The absolute error between the desired response and the Chebyshev approximation increases as ω increases from 0 to $2\pi f_p$.

If M is even, the real-valued frequency response of the FIR filter, $H_r(\omega)$, has the characteristics that $H_r(0) = 0$ and $H_r(\pi) \neq 0$. This filter satisfies the condition of zero response at zero frequency. It is fullband and this design results in a significantly smaller approximation error than comparable odd-length differentiators. Hence, even-length (odd order) differentiators are preferred in practical systems.

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

`Differentiator Filter` | dsp.BiquadFilter | dsp.FIRFilter | dsp.HighpassFilter | dsp.VariableBandwidthFIRFilter | dsp.VariableBandwidthIIRFilter

Introduced in R2016a

step

System object: dsp.Differentiator
Package: dsp

Differentiate input signal with respect to time

Syntax

`Y = step(DF,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(DF,X)` filters the real or complex input signal, `X`, using the differentiator filter, `DF`. `Y` is a differentiated version of `X`. The input `X` must be a column vector or matrix. If `X` is a matrix, each column is treated as an independent channel. `X` can be a floating point or a fixed-point input.

Input Arguments

DF — Differentiator filter

`dsp.Differentiator` System object

Differentiator filter, specified as a `dsp.Differentiator` System object.

X — Input signal

vector | matrix

Input signal, specified as a vector or matrix with floating point or fixed-point precision. The row length of `X` is the frame size, that is, the channel length. Each column of `X` is treated as a separate channel. The column length of `X` is the number of channels.

Output Arguments

Y — Differentiated signal

vector | matrix

Differentiated signal, returned as a vector or matrix of the same size, data type, and complexity as the input signal, X.

Examples

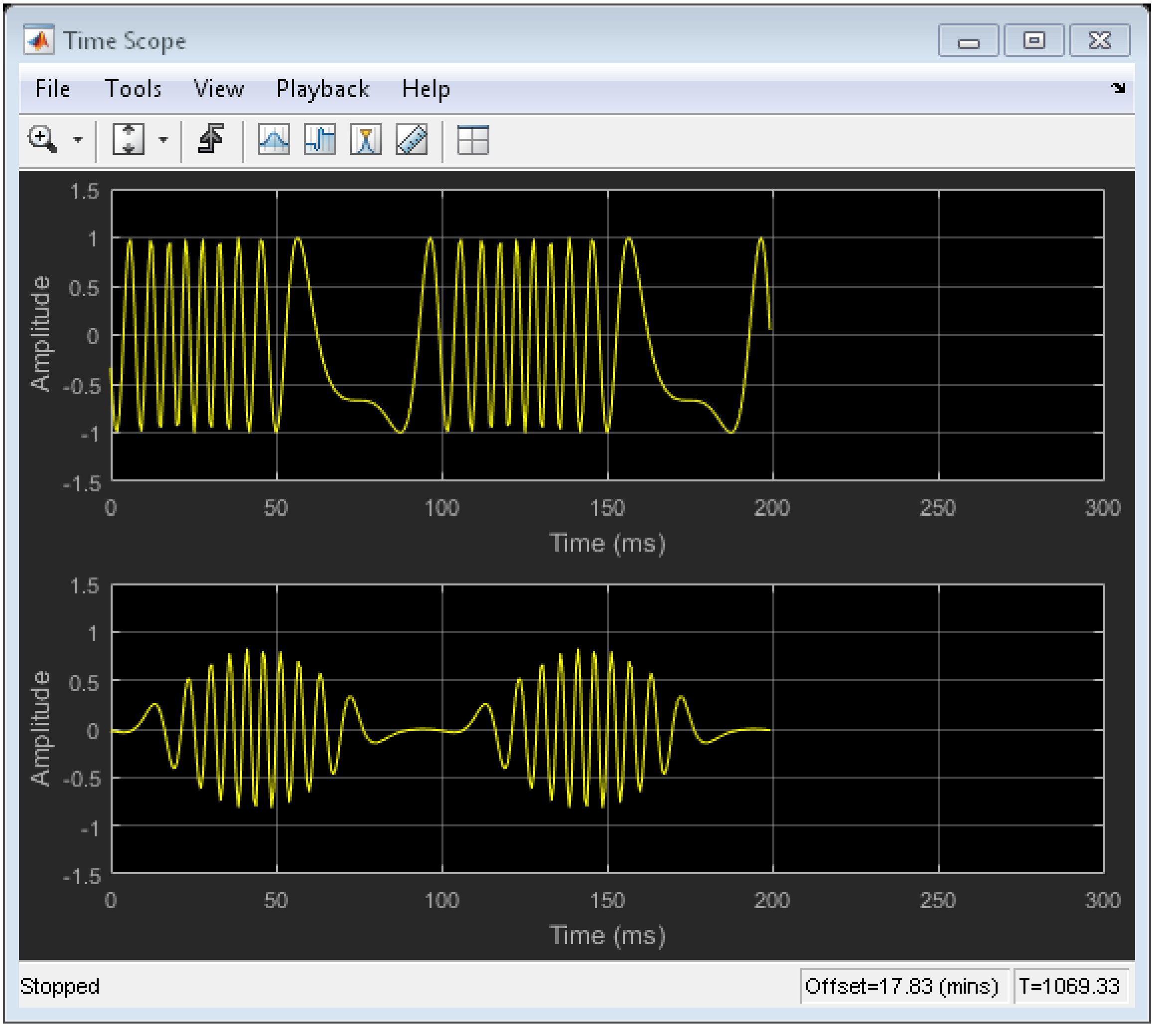
Convert FM Signal to AM Signal

Create an FM wave on a 100 Hz carrier signal sampled at 1.5 kHz.

```
Fc = 1e2;   % Carrier
Fs = 1.5e3; % Sample rate
sinewave = dsp.SineWave('Frequency',10,...
                            'SamplesPerFrame',1e3,...
                            'SampleRate',Fs);
```

Convert the FM signal to an AM signal.

```
ts = dsp.TimeScope(2,...
                      'TimeSpan',0.3,...
                      'BufferLength',10*Fs,...
                      'SampleRate',Fs,...
                      'ShowGrid',true,...
                      'YLimits',[-1.5 1.5],...
                      'LayoutDimensions',[2 1]);

df = dsp.Differentiator;

tic
while toc<2.2
    x = step(sinewave);
    fm_y = modulate(x,Fc,Fs,'fm');
    am_y = step(df,fm_y);
    step(ts,fm_y,am_y);
end

release(df);
release(ts);
```



Introduced in R2016a

getFilter

System object: dsp.Differentiator
Package: dsp

Get underlying FIR filter

Syntax

```
filter = getFilter(DF)
```

Description

`filter = getFilter(DF)` returns the underlying FIR filter, `filter`, used to implement the differentiator, `DF`.

Input Arguments

DF — Differentiator filter

`dsp.Differentiator` System object

Differentiator filter, specified as a `dsp.Differentiator` System object.

Examples

Get the Underlying FIR Filter of the Differentiator

Create a `dsp.Differentiator` System object with default properties.

```
DF = dsp.Differentiator;
```

Use `getFilter` to get the underlying FIR filter, `filter`, which implements the differentiator, `DF`.

```
filter = getFilter(DF)
```

```
filter =

  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: [1×60 double]
    InitialConditions: 0

  Use get to show all properties
```

Introduced in R2016a

reset

System object: dsp.Differentiator
Package: dsp

Reset internal states of differentiator

Syntax

```
reset(DF)
```

Description

`reset(DF)` resets the filter states of the differentiator, `DF`, to their initial values. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the differentiator to nonzero input data, the filter might have nonzero states. If you call `step` again without first calling `reset`, the object might produce different outputs for an identical input.

Input Arguments

DF — Differentiator filter

`dsp.Differentiator` System object

Differentiator filter, specified as a `dsp.Differentiator` System object.

Examples

Reset Differentiator

Create a differentiator with default properties.

```
DF = dsp.Differentiator;
```

Create a two-channel random signal. Call the `step` method twice on the signal.

```
x = randn(10,2);

y1 = step(DF,x);
y2 = step(DF,x);

no = all(y2==y1)
```

```
no =

  1×2 logical array

   0   0
```

The output is different because the internal states of `DF` have changed. Use `reset` to reset the differentiator. Call `step` again, and verify that the output is unchanged.

```
reset(DF)

y3 = step(DF,x);

yes = all(y3==y1)
```

```
yes =

  1×2 logical array

   1   1
```

Introduced in R2016a

dsp.DigitalDownConverter System object

Package: dsp

Translate digital signal from Intermediate Frequency (IF) band to baseband and decimate it

Description

The `DigitalDownConverter` object translates digital signal from Intermediate Frequency (IF) band to baseband, and decimates it.

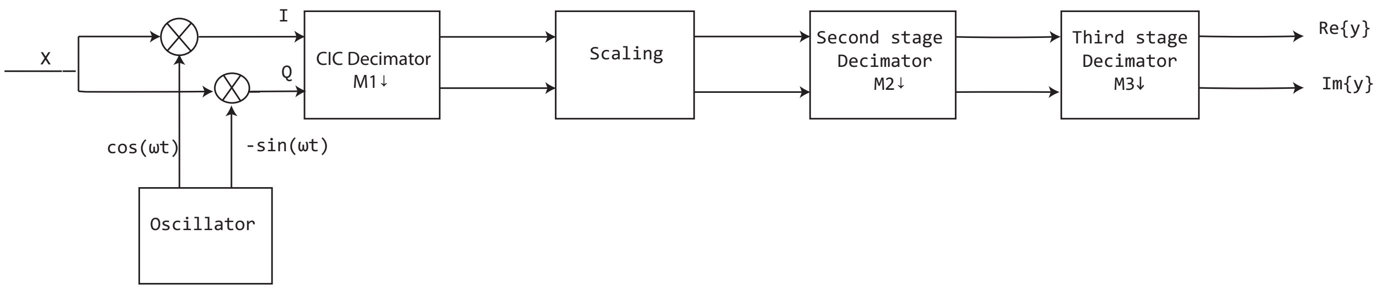
To digitally downconvert the input signal:

1. Define and set up your digital down converter. See “Construction” on page 4-559.
2. Call step to downconvert the input according to the properties of `dsp.DigitalDownConverter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

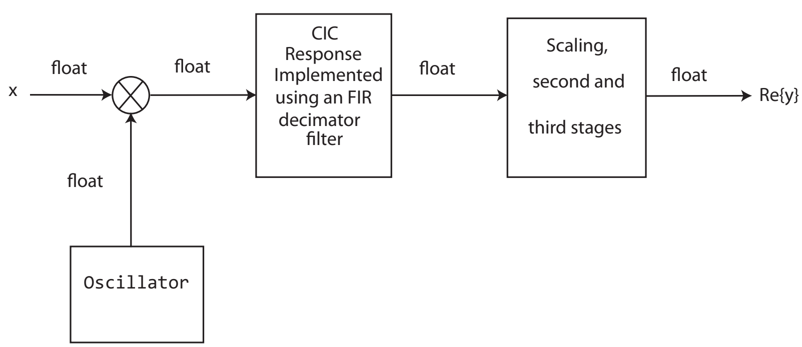
Construction

`dwnConv = dsp.DigitalDownConverter` returns a digital down-converter (DDC) System object, `dwnConv`. The object downconverts the input signal by multiplying it with a complex exponential with center frequency equal to the value in the `CenterFrequency` property. The object downsamples the frequency down-converted signal using a cascade of three decimation filters. In this case, the filter cascade consists of a CIC decimator, a CIC compensator, and a third FIR decimation stage. The following block diagram shows the architecture of the digital down converter.

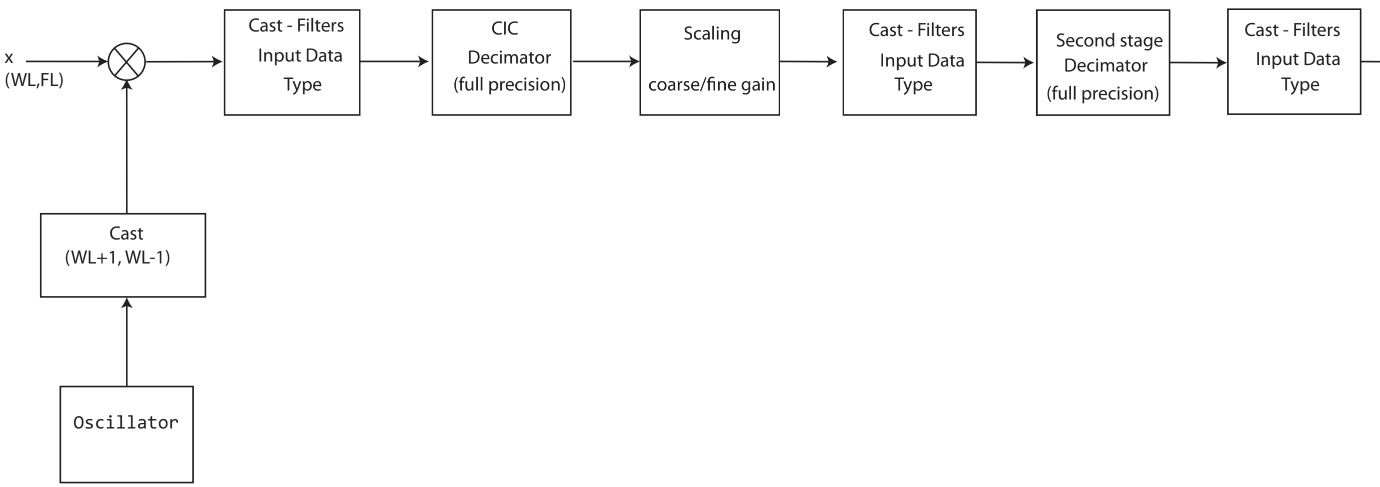


The scaling section normalizes the CIC gain and the oscillator power. It may also contain a correction factor to achieve the desired ripple specification. When you set the **Oscillator** property to `InputPort`, the normalization factor does not include the oscillator power factor. Depending on the setting of the **DecimationFactor** property, you may be able to bypass the third filter stage. When the input data type is double or single, the object implements an *N*-section CIC decimation filter as an FIR filter with a response that corresponds to a cascade of *N* boxcar filters. A true CIC filter with actual comb and integrator sections is implemented when the input data is of a fixed-point type. The CIC filter is emulated with an FIR filter so that you can run simulations with floating-point data.

The following block diagram represents the DDC arithmetic with single or double-precision, floating-point inputs.

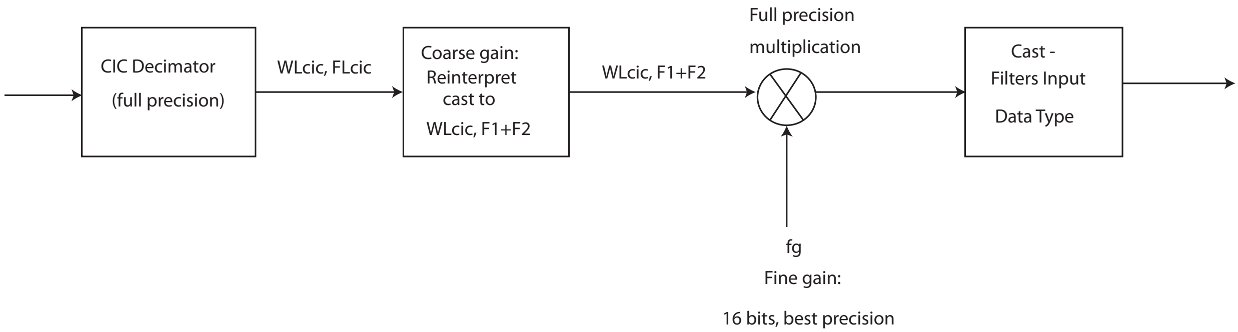


The following block diagram represents the DDC arithmetic with signed fixed-point inputs.



- `WL` is the word length of the input, and `FL` is the fraction length of the input.
- The input of each filter is cast to the data type specified in the **FiltersInputDataType** and **CustomFiltersInputDataType** properties.
- The oscillator output is cast to a word length equal to the input word length plus one. The fraction length is equal to the input word length minus one.
- The scaling at the output of the CIC decimator consists of coarse- and fine-gain adjustments. The coarse gain is achieved using the `reinterpretcast` function on the CIC decimator output. The fine gain is achieved using full-precision multiplication.

The following figure depicts the coarse- and fine-gain operations.



If the normalization gain is G (where `0<G≦1`), then:

- `WLcic` is the word length of the CIC decimator output and `FLcic` is the fraction length of the CIC decimator output
- `F1 = abs(nextpow2(G))`, indicating the part of G achieved using bit shifts (coarse gain)
- `F2` = fraction length specified by the **FiltersInputDataType** and **CustomFiltersInputDataType** properties
- `fg = fi((2^F1)*G, true, 16)`, indicating that the remaining gain cannot be achieved with a bit shift (fine gain)

`dwnConv = dsp.DigitalDownConverter(Name,Value)` returns a DDC object, `dwnConv`, with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`SampleRate`

Sample rate of input signal

Set this property to a positive scalar value, greater than or equal to twice the value of the `CenterFrequency` property. The default is `30` MHz.

DecimationFactor

Decimation factor

Set this property to a positive integer scalar, or to a 1-by-2 or 1-by-3 vector of positive integers.

When you set this property to a scalar, the object automatically chooses the decimation factors for each of the three filtering stages.

When you set this property to a 1-by-2 vector, the object bypasses the third filter stage and sets the decimation factor of the first and second filtering stages to the values in the first and second vector elements respectively. Both elements of the `DecimationFactor` vector must be greater than one.

When you set this property to a 1-by-3 vector, the *i* th element of the vector specifies the decimation factor for the *i*th filtering stage. The first and second elements of the `DecimationFactor` vector must be greater than one, and the third element must be 1 or 2. The default is 100.

MinimumOrderDesign

Minimum order filter design

When you set this property to `true`, the object designs filters with the minimum order that meets the passband ripple, stopband attenuation, passband frequency, and stopband frequency specifications that you set using the `PassbandRipple`, `StopbandAttenuation`, `Bandwidth`, `StopbandFrequencySource`, and `StopbandFrequency` properties.

When you set this property to `false`, the object designs filters with orders that you specify in the `NumCICSections`, `SecondFilterOrder`, and `ThirdFilterOrder` properties. The filter designs meet the passband and stopband frequency specifications that you set using the `Bandwidth`, `StopbandFrequencySource`, and `StopbandFrequency` properties. The default is `true`.

NumCICSections

Number of sections of CIC decimator

Set this property to a positive integer scalar. This property applies when you set the `MinimumOrderDesign` property to `false`. The default is `3`.

SecondFilterOrder

Order of CIC compensation filter stage

Set this property to a positive integer scalar. This property applies when you set the MinimumOrderDesign property to false. The default is 12.

ThirdFilterOrder

Order of third filter stage

Set this property to an even positive integer scalar. When you set the DecimationFactor property to a 1-by-2 vector, the object ignores the ThirdFilterOrder property because the third filter stage is bypassed. This property applies when you set the MinimumOrderDesign property to false. The default is 10.

Bandwidth

Two-sided bandwidth of input signal in Hz

Set this property to a positive integer scalar. The object sets the passband frequency of the cascade of filters to one-half of the value that you specify in the Bandwidth property. Set the value of this property to less than SampleRate/DecimationFactor. The default is 200 kHz.

StopbandFrequencySource

Source of stopband frequency

Specify the source of the stopband frequency as one of Auto | Property. The default is Auto. When you set this property to Auto, the object places the cutoff frequency of the cascade filter response at approximately F_c = SampleRate/M/2 Hz, where M is the total decimation factor that you specify in the DecimationFactor property. The object computes the stopband frequency as $F_{stop} = F_c + TW/2$. TW is the transition bandwidth of the cascade response computed as $2\times(F_c–F_p)$, and the passband frequency, F_p, equals Bandwidth/2.

StopbandFrequency

Stopband frequency in Hz

Set this property to a double-precision positive scalar. This property applies when you set the StopbandFrequencySource property to Property. The default is 150 kHz.

PassbandRipple

Passband ripple of cascade response in dB.

Set this property to a double-precision positive scalar. When you set the `MinimumOrderDesign` property to `true`, the object designs the filters so that the cascade response meets the passband ripple that you specify in the `PassbandRipple` property. This property applies when you set the `MinimumOrderDesign` property to `true`. The default is `0.1` dB.

StopbandAttenuation

Stopband attenuation of cascade response in dB

Set this property to a double-precision positive scalar. When you set the `MinimumOrderDesign` property to `true`, the object designs the filters so that the cascade response meets the stopband attenuation that you specify in the `StopbandAttenuation` property. This property applies when you set the `MinimumOrderDesign` property to `true`. The default is `60` dB.

Oscillator

Type of oscillator

Specify the oscillator as one of `Sine wave` | `NCO` | `Input port` | `None`. The default is `Sine wave`. When you set this property to `Sine wave`, the object frequency down converts the input signal using a complex exponential obtained from samples of a sinusoidal trigonometric function. When you set this property to `NCO`, the object performs frequency down conversion with a complex exponential obtained using a numerically controlled oscillator (NCO). When you set this property to `Input port`, the object performs frequency down conversion using the complex signal that you set as an input to the step method. When you set this property to `None`, the mixer stage in the object is not present and the object acts as three stage cascaded decimator.

CenterFrequency

Center frequency of input signal in Hz

Specify this property as a double-precision positive scalar that is less than or equal to half the value of the `SampleRate` property. The object down converts the input signal from the passband center frequency you specify in the `CenterFrequency` property, to `0`

Hz. This property applies when you set the `Oscillator` property to `Sine wave` or `NCO`. The default is `14` MHz.

`NumAccumulatorBits`

Number of NCO accumulator bits

Specify this property as an integer scalar in the range `[1 128]`. This property applies when you set the `Oscillator` property to `NCO`. The default is `16`.

`NumQuantizedAccumulatorBits`

Number of NCO quantized accumulator bits

Specify this property as an integer scalar in the range `[1 128]`. The value you specify in this property must be less than the value you specify in the `NumAccumulatorBits` property. This property applies when you set the Oscillator property to `NCO`. The default is `12`.

`Dither`

Dither control for NCO

When you set this property to `true`, a number of dither bits specified in the `NumDitherBits` property will be used to apply dither to the NCO signal. This property applies when you set the `Oscillator` property to `NCO`. The default is `true`.

`NumDitherBits`

Number of NCO dither bits

Specify this property as an integer scalar smaller than the number of accumulator bits that you specify in the `NumAccumulatorBits` property. This property applies when you set the `Oscillator` property to `NCO` and the Dither property to `true`. The default is `4`.

Fixed-Point Properties

`FiltersInputDataType`

Data type of input of each filter stage

Specify the data type at the input of the first, second, and third (if it has not been bypassed) filter stages as one of `Same as input | Custom`. The default is `Same as` input. The object casts the data at the input of each filter stage according to the value you set in this property.

CustomFiltersInputDataType

Fixed-point data type of input of each filter stage

Specify the filters input fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the FiltersInputDataType property to Custom. The default is numerictype([],16,15).

OutputDataType

Data type of output

Specify the data type of output as Same as input | Custom. The default is Same as input.

CustomOutputDataType

Fixed-point data type of output

Specify the output fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the OutputDataType property to Custom. The default is numerictype([],16,15).

Methods

fvtool	Visualize response of filter cascade
getDecimationFactors	Get decimation factors of each filter stage
getFilters	Get handles to decimation filter objects
getFilterOrders	Get orders of decimation filters
groupDelay	Group delay of filter cascade
step	Digitally down convert input signal
visualizeFilterStages	Display response of filter stages

Common to All System Objects	
clone	Create System object with same property values

Common to All System Objects	
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Upconvert and Downconvert a Sine Wave Signal

Create a digital up converter object that up samples a 1 KHz sinusoidal signal by a factor of 20 and up converts it to 50 KHz. Create a digital down converter object that down converts the signal to 0 Hz and down samples it by a factor of 20.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create a sine wave generator to obtain the 1 KHz sinusoidal signal with a sample rate of 6 KHz.

```
Fs = 6e3; % Sample rate
sine = dsp.SineWave('Frequency',1000,'SampleRate',...
Fs,'SamplesPerFrame',1024);
x = sine(); % generate signal
```

Create a `DigitalUpConverter` object. Use minimum order filter designs and set passband ripple to 0.2 dB and the stopband attenuation to 55 dB. Set the double sided signal bandwidth to 2 KHz.

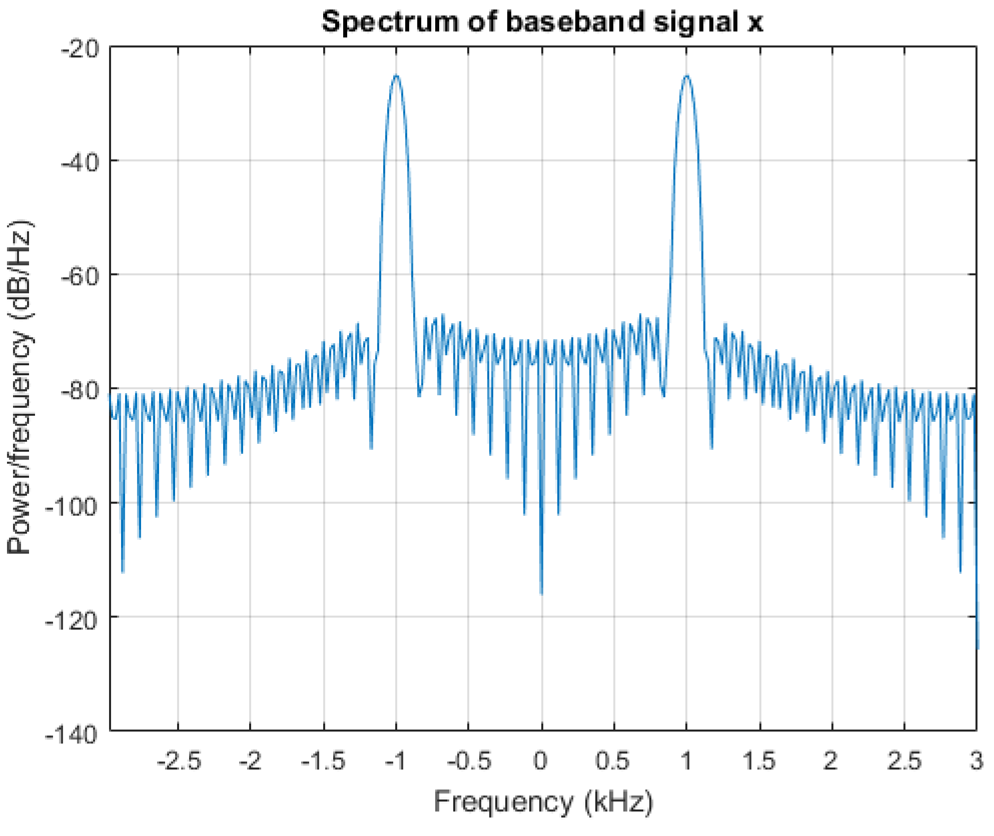
```
upConv = dsp.DigitalUpConverter(...
 'InterpolationFactor', 20,...
 'SampleRate', Fs,...
 'Bandwidth', 2e3,...
 'StopbandAttenuation', 55,...
 'PassbandRipple',0.2,...
 'CenterFrequency',50e3);
```

Create a `DigitalDownConverter` object. Use minimum order filter designs and set the passband ripple to 0.2 dB and the stopband attenuation to 55 dB.

```
dwnConv = dsp.DigitalDownConverter(...
  'DecimationFactor',20,...
  'SampleRate', Fs*20,...
  'Bandwidth', 3e3,...
  'StopbandAttenuation', 55,...
  'PassbandRipple',0.2,...
  'CenterFrequency',50e3);
```

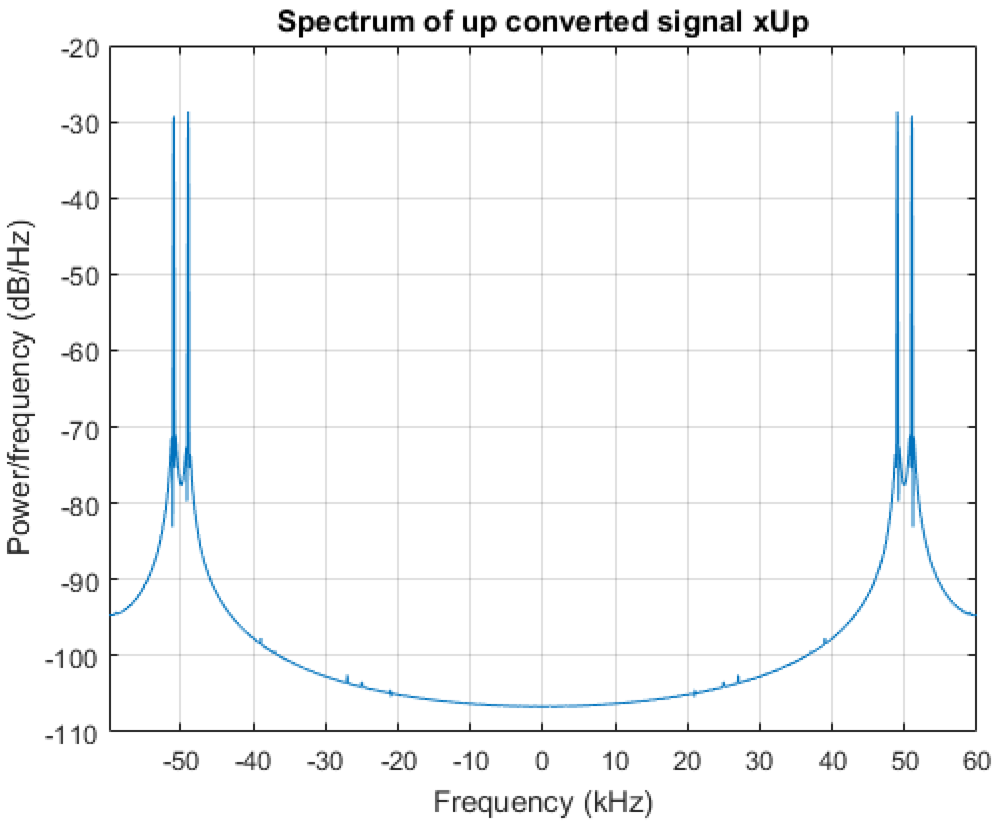
Create a spectrum estimator to visualize the signal spectrum before up converting, after up converting, and after down converting.

```
window = hamming(floor(length(x)/10));
figure; pwelch(x,window,[],[],Fs,'centered')
title('Spectrum of baseband signal x')
```



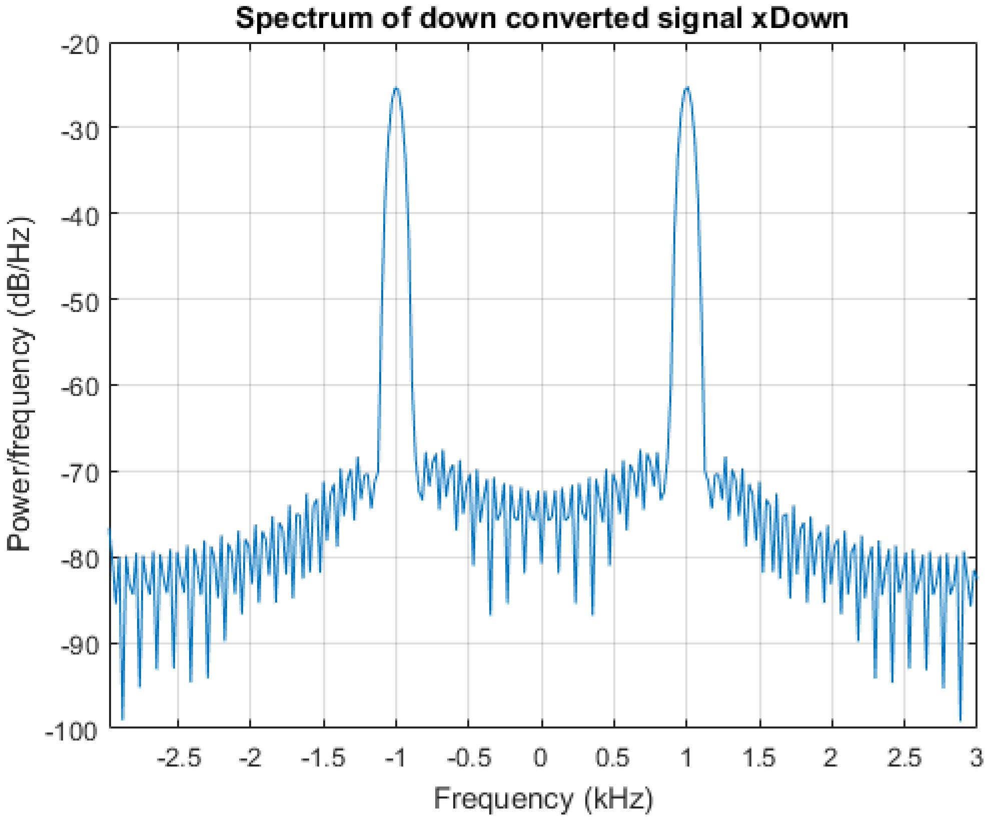
Up convert the signal and visualize the spectrum

```
xUp = upConv(x); % up convert
window = hamming(floor(length(xUp)/10));
figure; pwelch(xUp,window,[],[],20*Fs,'centered');
title('Spectrum of up converted signal xUp')
```



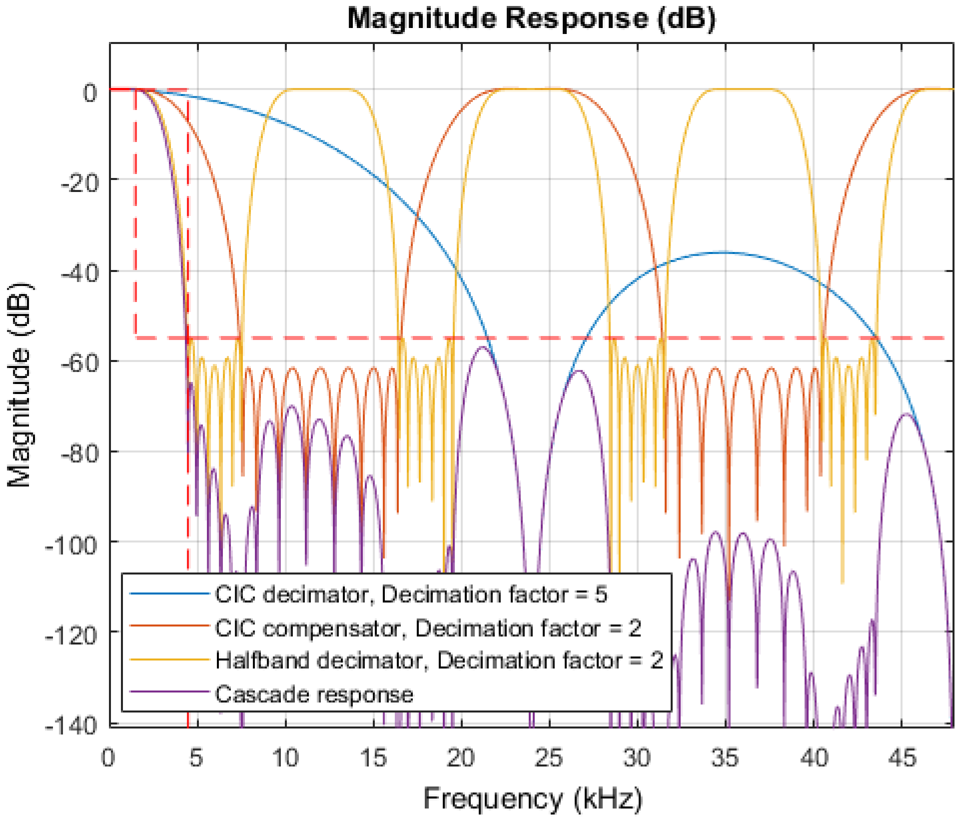
Down convert the signal and visualize the spectrum

```
xDown = dwnConv(xUp); % down convert
window = hamming(floor(length(xDown)/10));
figure; pwelch(xDown,window,[],[],Fs,'centered')
title('Spectrum of down converted signal xDown')
```



Visualize the response of the decimation filters

```
visualizeFilterStages(dwnConv)
```



- “Digital Up and Down Conversion for Family Radio Service”
- “Design and Analysis of a Digital Down Converter”

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

System Objects
dsp.DigitalUpConverter

Blocks
`Digital Down-Converter` | `Digital Up-Converter`

Topics
“Digital Up and Down Conversion for Family Radio Service”
“Design and Analysis of a Digital Down Converter”

Introduced in R2012a

fvtool

System object: dsp.DigitalDownConverter
Package: dsp

Visualize response of filter cascade

Syntax

```
fvtool(dwnConv)
fvtool(dwnConv,..,'Arithmetic',ARITH,...)
fvtool(dwnConv,..., PROP1, VALUE1,PROP2,VALUE2,...)
```

Description

`fvtool(dwnConv)` plots the magnitude response of the cascade of filters. By default, the object plots the cascade response up to the second CIC null frequency (or to the first when only one CIC null exists). When you set the `FilterSpecification` property to `Design parameters` the method plots a mask based on the filter specifications.

`fvtool(dwnConv,..,'Arithmetic',ARITH,...)` specifies the arithmetic of the filter cascade. You set input `ARITH` to `double`, `single`, or `fixed-point`. When object `dwnConv` is in an unlocked state you must specify the arithmetic. When object `dwnConv` is in a locked state the arithmetic input is ignored.

`fvtool(dwnConv,..., PROP1, VALUE1,PROP2,VALUE2,...)` launches FVTool and sets the specified FVTool properties to the specified values.

getDecimationFactors

System object: dsp.DigitalDownConverter
Package: dsp

Get decimation factors of each filter stage

Syntax

`M = getDecimationFactors(dwnConv)`

Description

`M = getDecimationFactors(dwnConv)`) returns a vector, `M`, with the decimation factors of each filter stage. If the third filter stage is bypassed, then `M` is a 1-by-2 vector containing the decimation factors of the first and second filter stages in the first and second elements respectively. If the third filter stage is not bypassed then `M` is a 1-by-3 vector containing the decimation factors of the first, second and third filter stages.

getFilters

System object: dsp.DigitalDownConverter
Package: dsp

Get handles to decimation filter objects

Syntax

```
S = getFilters(dwnConv)
getFilters(dwnConv,'Arithmetic',ARITH)
```

Description

`S = getFilters(dwnConv)` returns a structure, `S`, with copies of the filter System objects and the CIC normalization factor that form the decimation filter cascade. The `ThirdFilterStage` structure field is empty if the third filter stage has been bypassed. The CIC normalization factor equals the inverse of the CIC filter gain. In some cases, this gain includes a correction factor to ensure that the cascade response meets the ripple specifications.

`getFilters(dwnConv,'Arithmetic',ARITH)` specifies the arithmetic of the filter stages. You can set `ARITH` to `double`, `single`, or `fixed-point`. When object `dwnConv` is in an unlocked state, you must specify the arithmetic input. When object `dwnConv` is in a locked state, the arithmetic input is ignored.

When `dwnConv` is in an unlocked state, and you specify the arithmetic as `fixed-point`, the `getFilters` method returns filter System objects. The custom coefficient data type properties of these System objects are set to the values that the `dsp.DigitalDownConverter` System object uses to process data when you call the `step` method. All other fixed-point properties are set to their default values.

When `dwnConv` is in a locked state, and the input to the step method is of a fixed-point data type, the `getFilters` method returns filter System objects. All fixed-point properties of these System objects are set to the exact values that the `dsp.DigitalDownConverter` System object uses to process the data.

getFilterOrders

System object: dsp.DigitalDownConverter
Package: dsp

Get orders of decimation filters

Syntax

```
S = getFilterOrders(dwnConv)
```

Description

`S = getFilterOrders(dwnConv)` returns a structure, `S`, that contains the orders of the interpolation filter stages. The `ThirdFilterOrder` structure field will be empty if the third filter stage has been bypassed.

groupDelay

System object: dsp.DigitalDownConverter
Package: dsp

Group delay of filter cascade

Syntax

```
D = groupDelay(dwnConv,N)
[D,F] = groupDelay(dwnConv,N)
```

Description

`D = groupDelay(dwnConv,N)` returns a vector of group delays, `D`, evaluated at `N` frequency points equally spaced around the upper half of the unit circle. If you don't specify `N`, it defaults to `8192`.

`[D,F] = groupDelay(dwnConv,N)` returns a vector of frequencies, `F`, at which the group delay has been computed.

step

System object: dsp.DigitalDownConverter
Package: dsp

Digitally down convert input signal

Syntax

```
Y = step(dwnConv,X)
Y = step(dwnConv,X,Z)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(dwnConv,X)` takes a real or complex input column vector `X` and outputs a frequency down converted and down sampled signal `Y`. The length of input `X` must be a multiple of the decimation factor. `X` can be of data type double, single, signed integer, or signed fixed point (fi objects). The length of Y is equal to the length of `X` divided by the `DecimationFactor`. When the data type of `X` is double or single precision, the data type of `Y` is the same as that of `X`. When the data type of `X` is of a fixed point type, the data type of `Y` is defined by the `OutputDataType` on page 4-605 property.

`Y = step(dwnConv,X,Z)` uses the complex input, `Z`, as the oscillator signal used to frequency down convert input `X`, when you set the `Oscillator` on page 4-603 property to `Input port`. The length of `Z` must be equal to the length of `X`. `Z` can be of data type double, single, signed integer, or signed fixed point (fi objects).

visualizeFilterStages

System object: dsp.DigitalDownConverter
Package: dsp

Display response of filter stages

Syntax

```
visualizeFilterStages(dwnConv)
visualizeFilterStages(dwnConv,'Arithmetic',ARITH)
fvt = visualizeFilterStages(dwnConv)
```

Description

`visualizeFilterStages(dwnConv)` plots the magnitude response of the filter stages and of the cascade response. When you set the `FilterSpecification` property to `Design parameters` the method plots a mask based on the filter specifications. By default, the object plots the response of the filters up to the second CIC null frequency (or to the first when only one CIC null exists).

`visualizeFilterStages(dwnConv,'Arithmetic',ARITH)` specifies the arithmetic of the filter stages. You set input `ARITH` to `double`, `single`, or `fixed-point`. When object `dwnConv` is in an unlocked state you must specify the arithmetic. When object `dwnConv` is in a locked state the arithmetic input is ignored.

`fvt = visualizeFilterStages(dwnConv)` returns the handle to the FVTool object.

dsp.DigitalFilter System object

Package: dsp

Static or time-varying digital filter

Description

Note: `dsp.DigitalFilter` has been removed. For FIR filter structures, use dsp.FIRFilter. For Biquad (SOS) structures, use dsp.BiquadFilter. For non-SOS IIR filter structures, use dsp.IIRFilter. For All pole structures, use dsp.AllpoleFilter. For more details, see “Functionality being removed or replaced for blocks and System objects”.

The `DigitalFilter` object filters each channel of the input using static or time-varying digital filter implementations.

To filter each channel of the input using digital filter implementation:

1. Define and set up your IIR digital filter. See “Construction” on page 4-581.
2. Call step to filter each channel according to the properties of `dsp.DigitalFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = dsp.DigitalFilter` returns a default IIR digital filter object, `H`, which independently filters each channel of the input over successive calls to the `step` method, using a specified digital filter implementation. The default numerator coefficients are `[1 2]` and denominator coefficients are `[1 0.1]`.

`H = dsp.DigitalFilter('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a digital filter object, `H`, with each property set to the specified value.

Properties

TransferFunction

Type of filter transfer function

Specify the type of digital filter transfer function as one of | IIR (poles & zeros) | IIR (all poles) | FIR (all zeros) |. The default is IIR (poles & zeros).

Structure

Filter structure

Specify the filter structure.
When you set the TransferFunction on page 4-582 property to FIR (all zeros), you can specify the filter structure as one of | Direct form | Direct form symmetric | Direct form antisymmetric | Direct form transposed | Lattice MA |. The default is Direct form.
When you set the TransferFunction property to IIR (all poles), you can specify the filter structure as Direct form, Direct form transposed, or Lattice AR. The default is Direct form.
When you set the TransferFunction to IIR (poles & zeros), you can specify the filter structure as Direct form I, Direct form I transposed, Direct form II, Direct form II transposed, Biquad direct form I (SOS), Biquad direct form I transposed (SOS), Biquad direct form II (SOS), or Biquad direct form II transposed (SOS). The biquad filter structure does not apply when you set the CoefficientsSource on page 4-582 property to Input port. The default is Direct form II transposed.

CoefficientsSource

Source of filter coefficients

Specify the source of the filter coefficients as one of | Property | Input port |. The default is Property. When you specify Input port, the digital filter object updates the time-varying filter once every frame.

Numerator

Numerator coefficients

Specify the filter numerator coefficients as a real or complex numeric vector. This property applies when you set the TransferFunction on page 4-582 property to FIR (all zeros), the CoefficientsSource on page 4-582 property to Property, and the Structure on page 4-582 property is not set to Lattice MA. This property also applies when you set the TransferFunction property to IIR (poles & zeros), the CoefficientsSource on page 4-582 to Property, and the TransferFunction property to Direct form, Direct form symmetric, Direct form antisymmetric, or Direct form transposed. The default is [1 2]. This property is tunable.

Denominator

Denominator coefficients

Specify the filter denominator coefficients as a real or complex numeric vector. This property applies when you set the TransferFunction on page 4-582 property to IIR (all poles), the CoefficientsSource on page 4-582 property to Property and the Structure on page 4-582 property is not set to Lattice AR. This property also applies when you set the TransferFunction on page 4-582 property to IIR (poles & zeros), the CoefficientsSource on page 4-582 to Property, and the TransferFunction on page 4-582 property to Direct form, Direct form symmetric, Direct form antisymmetric, or Direct form transposed. When the TransferFunction on page 4-582 property is IIR (poles & zeros), the numerator and denominator must have the same complexity. The default is [1 0.1]. This property is tunable.

ReflectionCoefficients

Reflection coefficients of lattice filter structure

Specify the reflection coefficients of a lattice filter as a real or complex numeric vector. This property applies when you set the TransferFunction on page 4-582 property to FIR (all zeros), the Structure on page 4-582 property to Lattice MA, and the CoefficientsSource on page 4-582 property to Property. This property also applies when you set the TransferFunction on page 4-582 property to IIR (all poles), the Structure property to Lattice AR, and the CoefficientsSource property to Property. The default is [0.2 0.4]. This property is tunable.

SOSMatrix

SOS matrix of biquad filter structure

Specify the second-order section (SOS) matrix of a biquad filter as an M-by-6 matrix, where M is the number of sections in the filter. Each row of the SOS matrix contains the numerator and denominator coefficients of the corresponding filter section. The first three elements of each row are the numerator coefficients and the last three elements are the denominator coefficients. You can use real or complex coefficients. This property applies when you set the `TransferFunction` on page 4-582 property to `IIR (poles and zeros)`, the `CoefficientsSource` on page 4-582 property to `Property`, and the `Structure` on page 4-582 property to `Biquad direct form I (SOS)`, `Biquad direct form I transposed (SOS)`, `Biquad direct form II (SOS)`, or `Biquad direct form II transposed (SOS)`. The default is `[1 0.3 0.4 1 0.1 0.2]`. This property is tunable.

ScaleValues

Scale values of biquad filter structure

Specify the scale values to apply before and after each section of a biquad filter. `ScaleValues` must be a scalar or a vector of length M+1, where$M$ is the number of sections. If you specify a scalar value, that value is used as the gain value before the first section of the second-order filter and the rest of the gain values are set to 1. If you specify a vector of M+1 values, each value is used for a separate section of the filter. This property applies when you set the `TransferFunction` on page 4-582 property to `IIR (poles & zeros)`, the `CoefficientsSource` on page 4-582 property to `Property`, and the `Structure` on page 4-582 property to `Biquad direct form I (SOS)`, `Biquad direct form I transposed (SOS)`, `Biquad direct form II (SOS)`, or `Biquad direct form II transposed (SOS)`. The default is `1`. This property is tunable.

IgnoreFirstDenominatorCoefficient

Assume first denominator coefficient is 1

Setting this Boolean property to `true` reduces the number of computations to produce the digital filter output by omitting the first denominator term, *a0*, (poles side) in the filter structure. The object output is invalid when you set this property to `true` when the first denominator coefficient is not always 1 for your time-varying filter. The object ignores this property for fixed-point inputs, because this object does not support nonunity *a0* coefficients for fixed-point inputs.

This property applies when you set the `CoefficientsSource` on page 4-582 property to `Input port`, the `TransferFunction` on page 4-582 property to `IIR (all poles)`, and the `Structure` on page 4-582 property is not set to `Lattice AR`. This property also applies when you set the `CoefficientsSource` on page 4-582 property to `Input port`, the `TransferFunction` on page 4-582 property to `IIR (poles & zeros)`, and the `Structure` property to `Direct form`, `Direct form symmetric`, `Direct form antisymmetric`, or `Direct form transposed`. The default is `true`.

InitialConditions

Initial conditions for all poles or all zeros filter

Specify the initial conditions of the filter states. When the `TransferFunction` on page 4-582 property is `FIR (all zeros)` or `IIR (all poles)`, the number of states or delay elements equals the number of reflection coefficients for the lattice structure, or the number of filter coefficients–1 for the other direct form structures. When the `TransferFunction` property is `IIR (poles & zeros)`, the number of states for direct form II and direct form II transposed structures equals the `max(N,M)`–1, where `N` and `M` are the number of poles and zeros, respectively.

You can specify the initial conditions as a scalar, vector, or matrix. If you specify a scalar value, the digital filter object initializes all delay elements in the filter to that value. If you specify a vector whose length equals the number of delay elements in the filter, each vector element specifies a unique initial condition for the corresponding delay element. The object applies the same vector of initial conditions to each channel of the input signal.

If you specify a vector whose length equals the product of the number of input channels and the number of delay elements in the filter, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel.

If you specify a matrix with the same number of rows as the number of delay elements in the filter, and one column for each channel of the input signal, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel. This property applies when you do not set the `Structure` on page 4-582 property to `Direct Form I`, `Direct Form I transposed`, `Biquad direct form I (SOS)`, or `Biquad direct form I transposed (SOS)`. The default is 0.

NumeratorInitialConditions

Initial conditions on zeros side

Specify the initial conditions of the filter states on the side of the filter structure with the zeros. This property applies when you set the `TransferFunction` on page 4-582 property to `IIR (poles & zeros)` and the `Structure` on page 4-582 property is `Direct form I`, `Direct form I transposed`, `Biquad direct form`

I (SOS), or Biquad direct form I transposed (SOS). The default is 0. See the InitialConditions on page 4-585 property for information on setting initial conditions.

DenominatorInitialConditions

Initial conditions on poles side

Specify the initial conditions of the filter states on the side of the direct form I (noncanonic) filter structure with the poles. This property applies when you set the TransferFunction on page 4-582 property to IIR (poles & zeros) and the Structure on page 4-582 property to Direct form I, Direct form I transposed, Biquad direct form I (SOS), or Biquad direct form I transposed (SOS). The default is 0. See the InitialConditions on page 4-585 property for information on setting initial conditions.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap.

TapSumDataType

Tap sum word and fraction lengths

Specify the tap sum fixed-point data type as one of | Same as input | Custom |. This property applies when you set the TransferFunction on page 4-582 property to FIR (all zeros) and the Structure on page 4-582 property to either Direct form symmetric or Direct form antisymmetric. The default is Same as input.

CustomTapSumDataType

Custom tap sum word and fraction lengths

Specify the tap sum fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `TapSumDataType` on page 4-586 property to `Custom`. The default is `numerictype([],32,30)`.

`MultiplicandDataType`

Multiplicand word and fraction lengths

Specify the multiplicand fixed-point data type as one of `| Same as output | Custom |`. This property applies when you set the `Structure` on page 4-582 property to either `Direct form I transposed` or `Biquad direct form I transposed (SOS)`. The default is `Same as output`.

`CustomMultiplicandDataType`

Custom multiplicand word and fraction lengths

Specify the multiplicand fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `MultiplicandDataType` property to `Custom`. The default to `numerictype([],32,30)`.

`SectionInputDataType`

Section input word and fraction lengths

Specify the section input fixed-point data type as one of `| Same as input | Custom |`. Setting this property also sets the `SectionOutputDataType` on page 4-587 property to the same value. This property applies when you set the `TransferFunction` on page 4-582 property to `IIR (poles & zeros)` and the `Structure` on page 4-582 property to one of the biquad filter structures. The default is `Same as input`.

`CustomSectionInputDataType`

Custom section input word and fraction lengths

Specify the section input fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SectionInputDataType` on page 4-587 property to `Custom`. The `CustomSectionInputDataType` on page 4-587 and `CustomSectionOutputDataType` on page 4-588 property values must have equal word lengths. The default is `numerictype([],16,15)`.

`SectionOutputDataType`

Section output word and fraction lengths

Specify the section output fixed-point data type as one of | `Same as input` | `Custom` |. Setting this property also sets the `SectionInputDataType` on page 4-587 property to the same value. This property applies when you set the `TransferFunction` on page 4-582 property to `IIR (poles & zeros)` and the `Structure` on page 4-582 property to one of the biquad filter structures. The default is `Same as input`.

CustomSectionOutputDataType

Custom section output word and fraction lengths

Specify the section output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SectionOutputDataType` on page 4-587 property to `Custom`. The `CustomSectionInputDataType` on page 4-587 and `CustomSectionOutputDataType` on page 4-588 property values must have equal word lengths. The default is `numerictype([],16,15)`.

NumeratorCoefficientsDataType

Numerator coefficients word and fraction lengths

Specify the numerator coefficients fixed-point data type as one of | `Same word length as input` | `Custom` |. Setting this property also sets the `DenominatorCoefficientsDataType` on page 4-588 and the `ScaleValuesDataType` on page 4-589 properties, if applicable, to the same value. This property applies when you set the `CoefficientsSource` on page 4-582 property to `Property` and the `TransferFunction` on page 4-582 property is not set to `IIR (all poles)`. The default is `Same word length as input`.

CustomNumeratorCoefficientsDataType

Custom numerator coefficients word and fraction lengths

Specify the numerator coefficients fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `NumeratorCoefficientsDataType` on page 4-588 property to `Custom`. The `CustomNumeratorCoefficientsDataType` on page 4-588, `CustomDenominatorCoefficientsDataType` on page 4-589 and `ScaleValuesDataType` on page 4-589 properties, if applicable, must have equal word lengths. The default is `numerictype([],16,15)`.

DenominatorCoefficientsDataType

Denominator coefficients word and fraction lengths

Specify the denominator coefficients fixed-point data type as one of | `Same word length as input` | `Custom` |. Setting this property also sets the `NumeratorCoefficientsDataType` on page 4-588 and `ScaleValuesDataType` on page 4-589 properties, if applicable, to the same value. This property applies when you set the `CoefficientsSource` on page 4-582 property to `Property` and the `TransferFunction` on page 4-582 property is not set to `FIR (all zeros)`. The default is `Same word length as input`.

CustomDenominatorCoefficientsDataType

Custom denominator coefficients word and fraction lengths

Specify the denominator coefficients fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `DenominatorCoefficientsDataType` property to `Custom`. The `CustomNumeratorCoefficientsDataType` on page 4-588, `CustomDenominatorCoefficientsDataType` and `CustomScaleValuesDataType` on page 4-589 properties, if they apply, must have equal word lengths. The default is `numerictype([],16,15)`.

ScaleValuesDataType

Scale values word and fraction lengths

Specify the scale values fixed-point data type as one of | `Same word length as input` | `Custom` |. Setting this property also sets the `NumeratorCoefficientsDataType` on page 4-588 and the `DenominatorCoefficientsDataType` on page 4-588 properties, if they apply, to the same value. This property applies when you set the `CoefficientsSource` on page 4-582 property to `Property`, the `TransferFunction` on page 4-582 property to `IIR (poles & zeros)`, and the `Structure` on page 4-582 property to one of the biquad filter structures. The default is `Same word length as input`.

CustomScaleValuesDataType

Custom scale values word and fraction lengths

Specify the scale values fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `CoefficientsSource` on page 4-582 property to `Property` and the `ScaleValuesDataType` on page 4-589 property to `Custom`. The `CustomNumeratorCoefficientsDataType` on page 4-588, `CustomDenominatorCoefficientsDataType` on page 4-589, and the

CustomScaleValuesDataType on page 4-589 properties, if applicable, must have equal word lengths. The default is numerictype([],16,15).

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of | Same as input | Custom |. The default is Same as input.

CustomProductDataType

Custom product word and fraction lengths

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the ProductDataType on page 4-590 property to Custom. The default is numerictype([],32,30).

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type to one of | Same as input | Same as product | Custom |. The default is Same as product.

CustomAccumulatorDataType

Custom accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the AccumulatorDataType on page 4-590 property to Custom. The default is numerictype([],32,30).

StateDataType

State word and fraction lengths

Specify the state fixed-point data type as one of | Same as input | Same as accumulator | Custom |. This property does not apply to any of the direct form or direct form I filter structures. The default is Same as accumulator.

CustomStateDataType

Custom state word and fraction lengths

Specify the state fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `StateDataType` on page 4-590 property to `Custom`. The default is `numerictype([],16,15)`.

`OutputDataType`

Output word and fraction lengths

Specify the output fixed-point data type as one of | `Same as input` | `Same as accumulator` | `Custom` |. The default is `Same as accumulator`.

`CustomOutputDataType`

Custom output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` on page 4-591 property to `Custom`. The default is `numerictype([],16,15)`.

Methods

reset	Reset internal states of digital filter
step	Filter input with digital filter object

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Use an FIR filter to apply a low pass filter to a waveform with two sinusoidal components:

```
 t = [0:63]./32e3;
 xin = (sin(2*pi*4e3*t)+sin(2*pi*12e3*t)) / 2;

hSR = dsp.SignalSource(xin', 4);
 hLog = dsp.SignalSink;
 hFilt = dsp.DigitalFilter;
 hFilt.TransferFunction = 'FIR (all zeros)';
 hFilt.Numerator = fir1(10,0.5);

 while ~isDone(hSR)
      input = step(hSR);
      filteredOutput = step(hFilt,input);
      step(hLog,filteredOutput);
 end

 filteredResult = hLog.Buffer;
 periodogram(filteredResult,[],[],32e3)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Digital Filter` block reference page. The object properties correspond to the block parameters.

This object processes inputs only as frames of data. The block lets you specify whether to process inputs as individual samples or as frames of data . You use the **Input processing** parameter on the block.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `SOSMatrix` and `Scalevalues` properties are not supported for code generation.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.FIRFilter | dsp.IIRFilter | dsp.BiquadFilter | dsp.AllpoleFilter

Introduced in R2012a

reset

System object: dsp.DigitalFilter
Package: dsp

Reset internal states of digital filter

Syntax

```
reset(H)
```

Description

`reset(H)` resets the filter states of the digital filter object,`H`, to their initial values of 0. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the digital filter object to nonzero input data, the states may be nonzero. Invoking the `step` method again without first invoking the`reset` method may produce different outputs for an identical input.

For example:

```
H = dsp.DigitalFilter;
H.TransferFunction = 'FIR (all zeros)';
H.Numerator = fir1(20,0.25);
n = 0:100;
x = cos(0.2*pi*n)+sin(0.8*pi*n);
y = step(H,x);
% Filter states are nonzero
% Invoke step method again without resetting states
y1 = step(H,x);
isequal(y,y1) % returns 0
% Now reset filter states to 0
reset(H)
% Invoke step method
y2 = step(H,x);
isequal(y,y2) % returns a 1
```

step

System object: dsp.DigitalFilter
Package: dsp

Filter input with digital filter object

Syntax

```
Y = step(HFILT,X)
Y = step(HFILT,X,COEFF)
Y = step(HFILT,X,NUM,DEN)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(HFILT,X)` filters the real or complex input signal `X` using the digital filter, `H`, to produce the output `Y` . The digital filter object operates on each channel of the input signal independently over successive calls to step method.

`Y = step(HFILT,X,COEFF)` uses the time-varying numerator or denominator coefficients, `COEFF` , to filter the input signal `X` and produce the output `Y` . You can use this option when you set the `TransferFunction` property to either `FIR (all zeros)` or `IIR (all poles)` and the `CoefficientsSource` property to `Input port`.

`Y = step(HFILT,X,NUM,DEN)` uses the time-varying numerator coefficients `NUM` and the time-varying denominator coefficients `DEN` to filter the input signal `X` and produce the output `Y` . You can use this option when you set the `TransferFunction` property to `IIR (poles and zeros)` and the `CoefficientsSource` property to `Input port`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.DigitalUpConverter System object

Package: dsp

Interpolate digital signal and translate it from baseband to Intermediate Frequency (IF) band

Description

The `DigitalUpConverter` object interpolates digital signal, and translates it from baseband to Intermediate Frequency (IF) band.

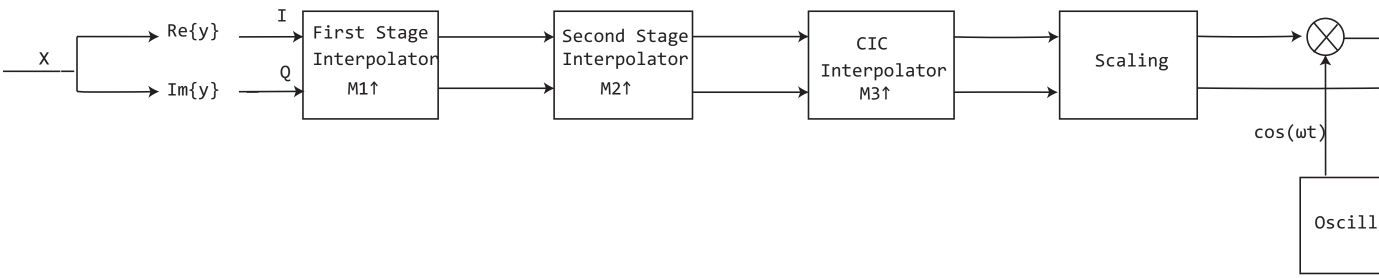
To digitally up convert the input signal:

1. Define and set up your digital up converter. See "Construction" on page 4-597.
2. Call step to upconvert the input according to the properties of `dsp.DigitalUpConverter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

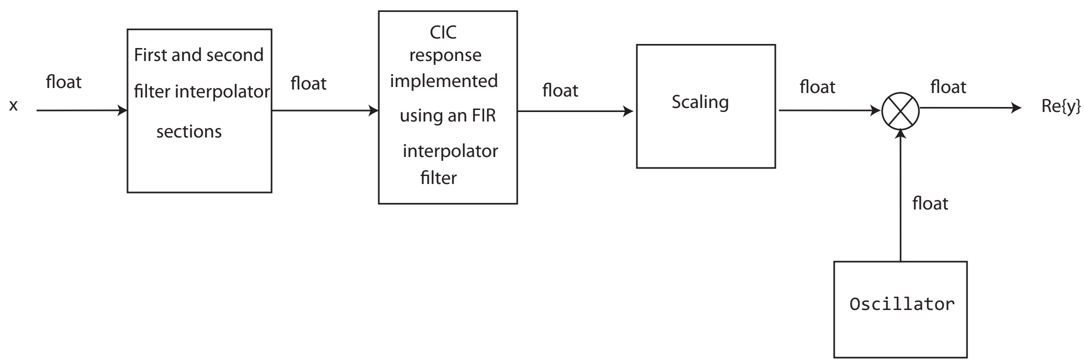
Construction

`upConv = dsp.DigitalUpConverter` returns a digital up-converter (DUC) System object, `upConv`. The object up samples the input signal using a cascade of three interpolation filters. This object frequency upconverts the up sampled signal by multiplying it with a complex exponential with center frequency equal to the value in the `CenterFrequency` property. In this case, the filter cascade consists of a first FIR interpolation stage, a CIC compensator, and a CIC interpolator. The following block diagram shows the architecture of the digital up converter.

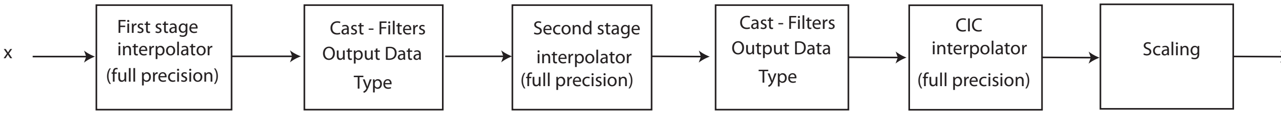


The scaling section normalizes the CIC gain and the oscillator power. It may also contain a correction factor to achieve the desired ripple specification. Depending on the setting of the **InterpolationFactor** property, you may be able to bypass the first filter stage. When the input data type is double or single, the object implements an N-section CIC interpolation filter as an FIR filter with a response that corresponds to a cascade of *N* boxcar filters. A true CIC filter with actual comb and integrator sections is implemented when the input data is of a fixed-point type. The CIC filter is emulated with an FIR filter so that you can run simulations with floating-point data.

The following diagram represents the DUC arithmetic with single or double-precision, floating-point inputs.

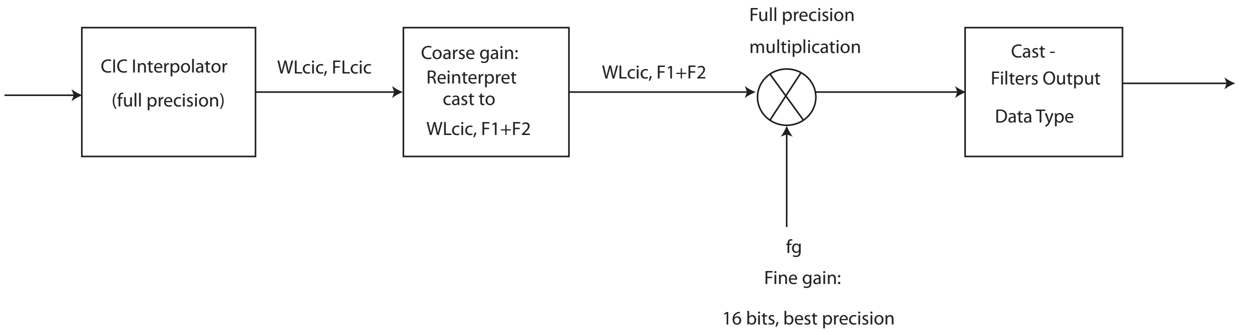


The following block diagram represents the DUC arithmetic with signed fixed-point inputs.



- `WL` is the word length of the input, and `FL` is the fraction length of the input.
- The output of each filter is cast to the data type specified in the **FiltersOutputDataType** and **CustomFiltersOutputDataType** properties. The casting of the CIC output occurs after the scaling factor is applied.
- The oscillator output is cast to a word length equal to the **FiltersOutputDataType** word length plus one. The fraction length is equal to the **FiltersOutputDataType** word length minus one.
- The scaling at the output of the CIC interpolator consists of coarse- and fine-gain adjustments. The coarse gain is achieved using the `reinterpretcast` function on the CIC interpolator output. The fine gain is achieved using full-precision multiplication.

The following figure depicts the coarse- and fine-gain operations.



If the normalization gain is G (where $0<G\leqq 1$), then:

- `WLcic` is the word length of the CIC interpolator output and `FLcic` is the fraction length of the CIC interpolator output
- `F1 = abs(nextpow2(G))`, indicating the part of G achieved using bit shifts (coarse gain)
- `F2` = fraction length specified by the **FiltersOutputDataType** and **CustomFiltersOutputDataType** properties
- `fg = fi((2^F1)*G, true, 16)`, indicating that the remaining gain cannot be achieved with a bit shift (fine gain)

`upConv = dsp.DigitalUpConverter('PropertyName','PropertyValue')` returns a DUC object, `upConv`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`SampleRate`

Sample rate of input signal

Set this property to a positive scalar. The value of this property multiplied by the total interpolation factor must be greater than or equal to twice the value of the `CenterFrequency` property. The default is `30` MHz.

InterpolationFactor

Interpolation factor

Set this property to a positive, integer scalar, or to a 1-by-2 or 1-by-3 vector of positive integers.

When you set this property to a scalar the object automatically chooses the interpolation factors for each of the three filtering stages.

When you set this property to a 1-by-2 vector, the object bypasses the first filter stage and sets the interpolation factor of the second and third filtering stages to the values in the first and second vector elements respectively. Both elements of the `InterpolationFactor` vector must be greater than one.

When you set this property to a 1-by-3 vector, the *i*th element of the vector specifies the interpolation factor for the *i*th filtering stage. The second and third elements of the `InterpolationFactor` vector must be greater than one and the first element must be `1` or `2`. The default is `100`.

MinimumOrderDesign

Minimum order filter design.

When you set this property to `true`, the object designs filters with the minimum order that meets the passband ripple, stopband attenuation, passband frequency, and stopband frequency specifications that you set using the `PassbandRipple`, `StopbandAttenuation`, `Bandwidth`, `StopbandFrequencySource`, and `StopbandFrequency` properties. When you set this property to`false`, the object designs filters with orders that you specify in the `FirstFilterOrder`, `SecondFilterOrder`, and `NumCICSections` properties. The filter designs meet the passband and stopband frequency specifications that you set using the `Bandwidth`, `StopbandFrequencySource`, and `StopbandFrequency` properties. The default is `true`.

SecondFilterOrder

Order of CIC compensation filter stage

Set this property to a positive, integer scalar. This property applies when you set the `MinimumOrderDesign` property to `false`. The default is `12`.

FirstFilterOrder

Order of first filter stage

Set this property to a positive, integer, even scalar. When you set the `InterpolationFactor` property to a 1-by-2 vector, the object ignores the `FirstFilterOrder` property because the first filter stage is bypassed. This property applies when you set the `MinimumOrderDesign` property to `false`. The default is `10`.

NumCICSections

Number of sections of CIC interpolator

Set this property to a positive, integer scalar. This property applies when you set the `MinimumOrderDesign` property to `false`. The default is `3`.

Bandwidth

Two-sided bandwidth of input signal in Hz .

Set this property to a positive, integer scalar. The object sets the passband frequency of the cascade of filters to one-half of the value that you specify in the `Bandwidth` property. The default is 200 kHz.

StopbandFrequencySource

Source of stopband frequency.

Specify the source of the stopband frequency as one of `Auto | Property`. The default is `Auto`. When you set this property to `Auto`, the object places the cutoff frequency of the cascade filter response at approximately F_c = `SampleRate`/2 Hz, and computes the stopband frequency as $F_{stop} = F_c + TW/2$. TW is the transition bandwidth of the cascade response, computed as $2\times(F_c–F_p)$, and the passband frequency, F_p, equals `Bandwidth`/2.

StopbandFrequency

Stopband frequency in Hz.

Set this property to a double precision positive scalar. This property applies when you set the `StopbandFrequencySource` property to `Property`. The default is `150` kHz.

PassbandRipple

Passband ripple of cascade response in dB.

Set this property to a double precision, positive scalar. When you set the MinimumOrderDesign property to true, the object designs the filters so that the cascade response meets the passband ripple that you specify in the PassbandRipple property. This property applies when you set the MinimumOrderDesign property to true. The default is 0.1 dB.

StopbandAttenuation

Stopband attenuation of cascade response in dB.

Set this property to a double precision, positive scalar. When you set the MinimumOrderDesign property to true, the object designs the filters so that the cascade response meets the stopband attenuation that you specify in the StopbandAttenuation property. This property applies when you set the MinimumOrderDesign property to true. The default is 60 dB.

Oscillator

Type of oscillator.

Specify the oscillator as one of Sine wave | NCO. The default is Sine wave. When you set this property to Sine wave, the object frequency up converts the output of the interpolation filter cascade using a complex exponential signal obtained from samples of a sinusoidal trigonometric function. When you set this property to NCO the object performs frequency up conversion with a complex exponential obtained using a numerically controlled oscillator (NCO).

CenterFrequency

Center frequency of output signal in Hz.

Specify this property as a double precision, positive scalar. The value of this property must be less than or equal to half the product of the SampleRate property times the total interpolation factor. The object up converts the input signal so that the output spectrum centers at the frequency you specify in the CenterFrequency property. The default is 14 MHz.

NumAccumulatorBits

Number of NCO accumulator bits

Specify this property as an integer scalar in the range [1 128]. This property applies when you set the Oscillator property to NCO. The default is 16.

See also dsp.NCO.

NumQuantizedAccumulatorBits

Number of NCO quantized accumulator bits.

Specify this property as an integer scalar in the range`[1 128]`. The value you specify for this property must be less than the value you specify in the `NumAccumulatorBits` property. This property applies when you set the `Oscillator` property to `NCO`. The default is `12`.

See also dsp.NCO.

Dither

Dither control for NCO.

When you set this property to `true`, the object uses the number of dither bits specified in the `NumDitherBits` property when applying dither to the NCO signal. This property applies when you set the `Oscillator` property to `NCO`. The default is `true`.

See also dsp.NCO.

NumDitherBits

Number of NCO dither bits.

Specify this property as an integer scalar smaller than the number of accumulator bits that you specify in the `NumAccumulatorBits` property. This property applies when you set the `Oscillator` property to `NCO` and the Dither property to `true`. The default is `4`.

See also dsp.NCO.

Fixed-Point Properties

FiltersOutputDataType

Data type of output of each filter stage

Specify the data type at the output of the first (if it has not been bypassed), second, and third filter stages as one of `Same as input` | `Custom`. The default is `Same as input`.

The object casts the data at the output of each filter stage according to the value you set in this property. For the CIC stage, the casting is done after the signal has been scaled by the normalization factor.

CustomFiltersOutputDataType

Fixed-point data type of output of each filter stage

Specify the filters output fixed-point type as a scaled numerictype object with a Signedness of `Auto`. This property applies when you set the `FiltersOutputDataType` property to `Custom`. The default is `numerictype([],16,15)`.

OutputDataType

Data type of output

Specify the data type of output as `Same as input` | `Custom`. The default is `Same as input`.

CustomOutputDataType

Fixed-point data type of output

Specify the output fixed-point type as a scaled numerictype object with a Signedness of `Auto`. This property applies when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],16,15)`.

Methods

fvtool	Visualize response of filter cascade
getFilters	Get handles to interpolation filter objects
getFilterOrders	Get orders of interpolation filters
getInterpolationFactors	Get interpolation factors of each filter stage
groupDelay	Group delay of filter cascade
step	Process inputs using the digital up converter
visualizeFilterStages	Display response of filter stages

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Upconvert a Sine Wave Signal

Create a digital up converter object that upsamples a 1 KHz sinusoidal signal by a factor of 20, and up converts it to 50 KHz.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Initialization

Create a sine wave generator to obtain the 1 KHz sinusoidal signal with a sample rate of 6 KHz.

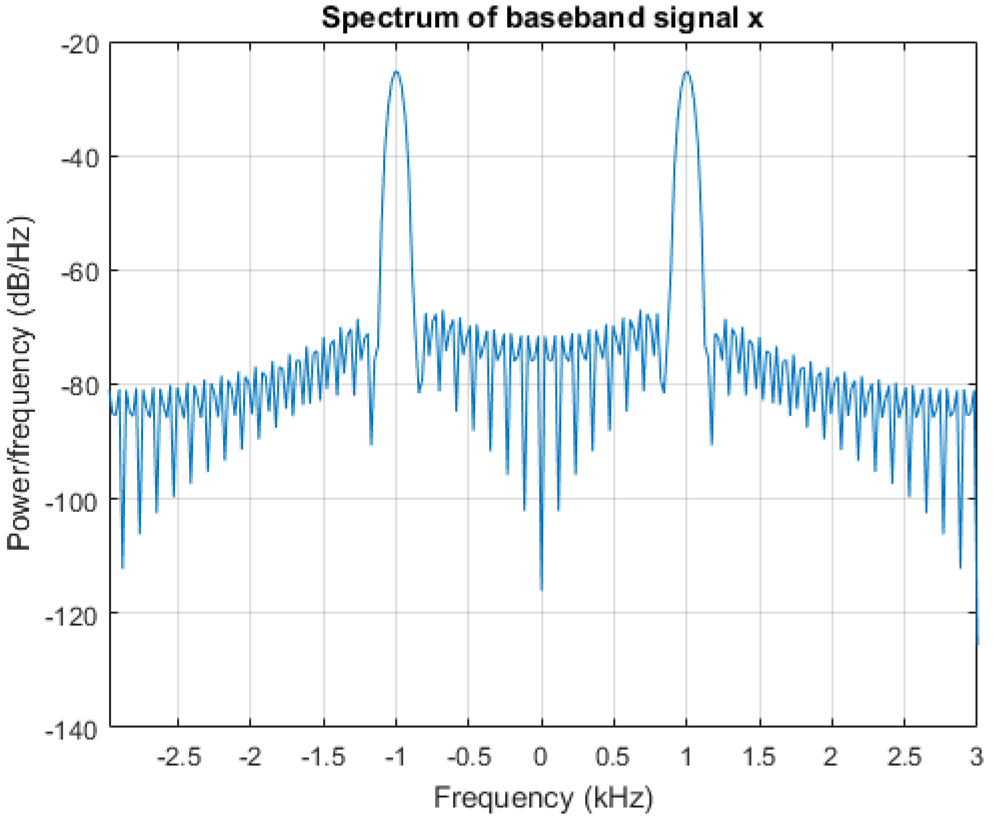
```
Fs = 6e3; % Sample rate
sine = dsp.SineWave('Frequency',1000,'SampleRate', Fs,'SamplesPerFrame',1024);
x = sine(); % generate signal
```

Create a DUC object. Use minimum order filter designs and set the passband ripple to 0.2 dB and the stopband attenuation to 55 dB. Set the double sided signal bandwidth to 2 KHz.

```
upConv = dsp.DigitalUpConverter(...
     'InterpolationFactor', 20,...
     'SampleRate', Fs,...
     'Bandwidth', 2e3,...
     'StopbandAttenuation', 55,...
     'PassbandRipple',0.2,...
     'CenterFrequency',50e3);
```

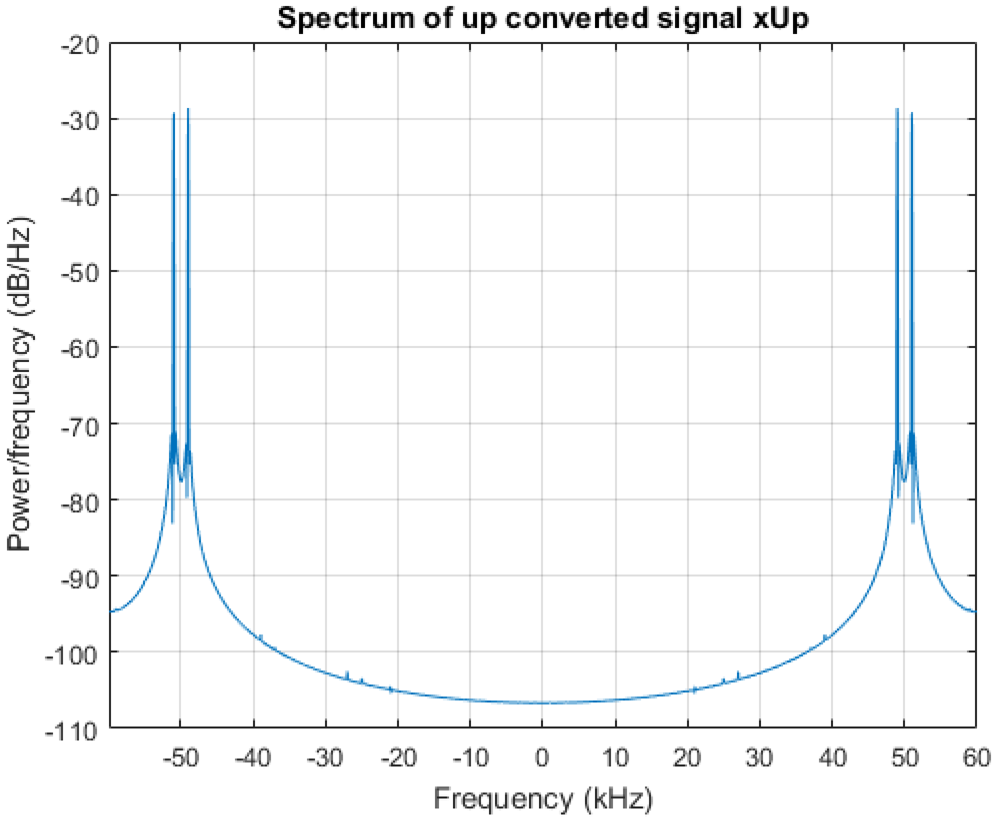
Create a spectrum estimator to visualize the signal spectrum before and after up converting.

```
window = hamming(floor(length(x)/10));
figure; pwelch(x,window,[],[],Fs,'centered')
title('Spectrum of baseband signal x')
```



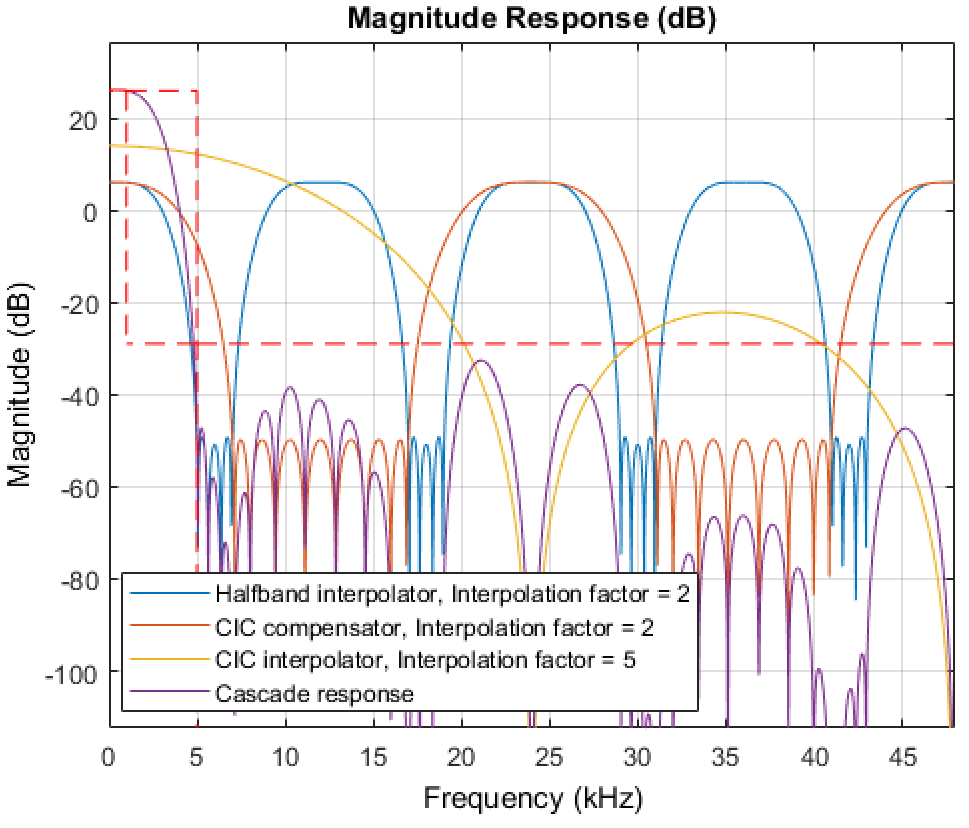
Upconvert the Signal and Visualize the Spectrum

```
xUp = upConv(x); % up convert
window = hamming(floor(length(xUp)/10));
figure; pwelch(xUp,window,[],[],20*Fs,'centered')
title('Spectrum of up converted signal xUp')
```



Visualize the response of the interpolation filters

```
visualizeFilterStages(upConv)
```



Related Examples

- “Digital Up and Down Conversion for Family Radio Service”
- “Design and Analysis of a Digital Down Converter”

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

dsp.DigitalDownConverter | `Digital Up-Converter` | `Digital Down-Converter`

Introduced in R2012a

fvtool

System object: dsp.DigitalUpConverter
Package: dsp

Visualize response of filter cascade

Syntax

```
fvtool(upConv)
fvtool(upConv,..,'Arithmetic',ARITH,...)
fvtool(upConv,..., PROP1, VALUE1,PROP2,VALUE2)
```

Description

`fvtool(upConv)` plots the magnitude response of the cascade of filters. By default, the object plots the cascade response up to the second CIC null frequency (or to the first when only one CIC null exists). When you set the `FilterSpecification` property to `Design parameters` the method plots a mask based on the filter specifications.

`fvtool(upConv,..,'Arithmetic',ARITH,...)` specifies the arithmetic of the filter cascade. You set input `ARITH` to `double`, `single`, or `fixed-point`. When object `upConv` is in an unlocked state you must specify the arithmetic. When object `upConv` is in a locked state the arithmetic input is ignored.

`fvtool(upConv,..., PROP1, VALUE1,PROP2,VALUE2)` launches FVTool and sets the specified FVTool properties to the specified values.

getFilters

System object: dsp.DigitalUpConverter
Package: dsp

Get handles to interpolation filter objects

Syntax

```
S = getFilters(upConv)
getFilters(upConv,'Arithmetic',ARITH)
```

Description

`S = getFilters(upConv)` returns a structure, `S`, with copies of the filter System objects and the CIC normalization factor that form the interpolation filter cascade. The `FirstFilterStage` structure field is empty if the first filter stage has been bypassed. The CIC normalization factor equals the inverse of the CIC filter gain. In some cases, this gain includes a correction factor to ensure that the cascade response meets the ripple specifications.

`getFilters(upConv,'Arithmetic',ARITH)` specifies the arithmetic of the filter stages. You set `ARITH` to `double`, `single`, or `fixed-point`. When object `upConv` is in an unlocked state, you must specify the arithmetic input. When object `upConv` is in a locked state, the arithmetic input is ignored.

When `upConv` is in an unlocked state, and you specify the arithmetic as `fixed-point`, the `getFilters` method returns filter System objects. The custom coefficient data type properties of these System objects are set to the values that the `dsp.DigitalUpConverter` System object uses to process data when you call the `step` method. All other fixed-point properties are set to their default values.

When `upConv` is in a locked state, and the input to the step method is of a fixed-point data type, the `getFilters` method returns filter System objects. All fixed-point properties of these System objects are set to the exact values that the `dsp.DigitalUpConverter` System object uses to process the data.

getFilterOrders

System object: dsp.DigitalUpConverter
Package: dsp

Get orders of interpolation filters

Syntax

```
S = getFilterOrders(upConv)
```

Description

`S = getFilterOrders(upConv)` returns a structure, `S`, that contains the orders of the interpolation filter stages. The `FirstFilterOrder` structure field will be empty if the first filter stage has been bypassed.

getInterpolationFactors

System object: dsp.DigitalUpConverter
Package: dsp

Get interpolation factors of each filter stage

Syntax

```
M = getInterpolationFactors(upConv)
```

Description

`M = getInterpolationFactors(upConv)` returns a vector, `M`, with the interpolation factors of each filter stage. If the first filter stage is bypassed, then `M` is a 1-by-2 vector containing the interpolation factors of the second and third stages in the first and second elements respectively. If the first filter stage is not bypassed then `M` is a 1-by-3 vector containing the interpolation factors of the first, second and third filter stages.

groupDelay

System object: dsp.DigitalUpConverter
Package: dsp

Group delay of filter cascade

Syntax

```
D = groupDelay(upConv,N)
[D,F] = groupDelay(upConv,N)
```

Description

`D = groupDelay(upConv,N)` returns a vector of group delays, `D`, evaluated at `N` frequency points equally spaced around the upper half of the unit circle. If you don't specify `N`, it defaults to `8192`.

`[D,F] = groupDelay(upConv,N)` returns a vector, `F`, of frequencies at which the group delay has been computed.

step

System object: dsp.DigitalUpConverter
Package: dsp

Process inputs using the digital up converter

Syntax

`Y = step(upConv,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(upConv,X)` takes a real or complex input column vector `X` and outputs an up sampled and frequency up converted signal `Y`. `X` can be of data type double, single, signed integer, or signed fixed point (fi objects). The length of `Y` is equal to the length of `X` multiplied by the `InterpolationFactor` on page 4-601. When the data type of `X` is double or single precision, the data type of `Y` is the same as that of `X`. When the data type of `X` is of a fixed point type, the data type of Y is defined by the `OutputDataType` on page 4-605 property.

visualizeFilterStages

System object: dsp.DigitalUpConverter
Package: dsp

Display response of filter stages

Syntax

```
visualizeFilterStages(upConv)
visualizeFilterStages(upConv,'Arithmetic',ARITH)
fvt = visualizeFilterStages(upConv)
```

Description

`visualizeFilterStages(upConv)` plots the magnitude response of the filter stages and of the cascade response. When the `FilterSpecification` property is set to `Design parameters` the method plots a mask based on the filter specifications. By default, the object plots the response of the filters up to the second CIC null frequency (or to the first when only one CIC null exists).

`visualizeFilterStages(upConv,'Arithmetic',ARITH)` specifies the arithmetic of the filter stages. You set input ARITH to 'double', 'single', or 'fixed-point'. When object `upConv` is in an unlocked state you must specify the arithmetic. When object `upConv` is in a locked state the arithmetic input is ignored.

`fvt = visualizeFilterStages(upConv)` returns the handle to the FVTool object.

dsp.DyadicAnalysisFilterBank System object

Package: dsp

Dyadic analysis filter bank

Description

The `DyadicAnalysisFilterBank` object uses a series of highpass and lowpass FIR filters to provide approximate octave band frequency decompositions of the input. Each filter output is downsampled by a factor of two. With the appropriate analysis filters and tree structure, the dyadic analysis filter bank is a discrete wavelet transform (DWT) or discrete wavelet packet transform (DWPT).

To obtain approximate octave band frequency decompositions of the input:

1. Define and set up your dyadic analysis filter bank. See "Construction" on page 4-618.
2. Call step to get the octave half band frequency decompositions of the input according to the properties of `dsp.DyadicAnalysisFilterBank`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`dydan = dsp.DyadicAnalysisFilterBank` constructs a dyadic analysis filter bank object, `dydan`, that computes the level-two discrete wavelet transform (DWT) of a column vector input. For a 2-D matrix input, the object transforms the columns using the Daubechies third-order extremal phase wavelet. The length of the input along the first dimension must be a multiple of 4.

```
dydan =
dsp.DyadicAnalysisFilterBank('PropertyName',PropertyValue, ...)
```

returns a dyadic analysis filter bank object, `dydan`, with each property set to the specified value.

Properties

`Filter`

Type of filter used in subband decomposition

Specify the type of filter used to determine the high and lowpass FIR filters in the dyadic analysis filter bank as `Custom` , `Haar`, `Daubechies`, `Symlets`, `Coiflets`, `Biorthogonal`, `Reverse Biorthogonal`, or `Discrete Meyer`. All property values except `Custom` require Wavelet Toolbox software. If the value of this property is `Custom`, the filter coefficients are specified by the values of the `CustomLowpassFilter` on page 4-619 and `CustomHighpassFilter` on page 4-620 properties. Otherwise, the dyadic analysis filter bank object uses the Wavelet Toolbox function `wfilters` to construct the filters. The following table lists supported wavelet filters and example syntax to construct the filters:

Filter	Example Setting	Syntax for Analysis Filters
Haar	N/A	`[Lo_D,Hi_D]=wfilters('haar');`
Daubechies extremal phase	`WaveletOrder=3;`	`[Lo_D,Hi_D]=wfilters('db3');`
Symlets (Daubechies least-asymmetric)	`WaveletOrder=4;`	`[Lo_D,Hi_D]=wfilters('sym4');`
Coiflets	`WaveletOrder=1;`	`[Lo_D,Hi_D]=wfilters('coif1');`
Biorthogonal	`FilterOrder='[3/1]';`	`[Lo_D,Hi_D,Lo_R,Hi_R]=...` `wfilters('bior3.1');`
Reverse biorthogonal	`FilterOrder='[3/1]';`	`[Lo_D,Hi_D,Lo_R,Hi_R]=...` `wfilters('rbior3.1');`
Discrete Meyer	N/A	`[Lo_D,Hi_D]=wfilters('dmey');`

`CustomLowpassFilter`

Lowpass FIR filter coefficients

Specify a vector of lowpass FIR filter coefficients, in powers of z^{-1}. Use a half-band filter that passes the frequency band stopped by the filter specified in the `CustomHighpassFilter` on page 4-620 property. This property applies when you set the `Filter` on page 4-619 property to `Custom`. The default specifies a Daubechies third-order extremal phase scaling (lowpass) filter.

`CustomHighpassFilter`

Highpass FIR filter coefficients

Specify a vector of highpass FIR filter coefficients, in powers of z^{-1}. Use a half-band filter that passes the frequency band stopped by the filter specified in the `CustomLowpassFilter` on page 4-619 property. This property applies when you set the `Filter` on page 4-619 property to `Custom`. The default specifies a Daubechies 3rd-order extremal phase wavelet (highpass) filter.

`WaveletOrder`

Order for orthogonal wavelets

Specify the order of the wavelet selected in the `Filter` on page 4-619 property. This property applies when you set the `Filter` property to an orthogonal wavelet: `Daubechies` (Daubechies extremal phase), `Symlets` (Daubechies least-asymmetric), or `Coiflets`. The default is `2`.

`FilterOrder`

Analysis and synthesis filter orders for biorthogonal filters

Specify the order of the analysis and synthesis filter orders for biorthogonal filter banks as `1 / 1`, `1 / 3`, `1 / 5`, `2 / 2`, `2 / 4`, `2 / 6`, `2 / 8`, `3 / 1`, `3 / 3`, `3 / 5`, `3 / 7`, `3 / 9`, `4 / 4`, or `5 / 5`, `6 / 8`. Unlike orthogonal wavelets, biorthogonal wavelets require different filters for the analysis (decomposition) and synthesis (reconstruction) of an input. The first number indicates the order of the synthesis (reconstruction) filter. The second number indicates the order of the analysis (decomposition) filter. This property applies when you set the `Filter` on page 4-619 property to `Biorthogonal` or `Reverse Biorthogonal`. The default is `1 / 1`.

`NumLevels`

Number of filter bank levels used in analysis (decomposition)

Specify the number of filter bank analysis levels a positive integer. A level-N asymmetric structure produces N+1 output subbands. A level-N symmetric structure produces 2^N output subbands. The default is 2. The size of the input along the first dimension must be a multiple of 2^N, where N is the number of levels.

TreeStructure

Structure of filter bank

Specify the structure of the filter bank as `Asymmetric` or `Symmetric`. The asymmetric structure decomposes only the lowpass filter output from each level. The symmetric structure decomposes the highpass and lowpass filter outputs from each level. If the analysis filters are scaling (lowpass) and wavelet (highpass) filters, the asymmetric structure is the discrete wavelet transform, while the symmetric structure is the discrete wavelet packet transform.

When this property is `Symmetric`, the output has 2^N subbands each of size $M/2^N$. In this case, M is the length of the input along the first dimension and N is the value of the `NumLevels` on page 4-620 property. When this property is `Asymmetric`, the output has N+1 subbands. The following equation gives the length of the output in the kth subband in the asymmetric case:

$$M_k = \begin{cases} \dfrac{M}{2^k} & 1 \le k \le N \\ \dfrac{M}{2^N} & k = N+1 \end{cases}$$

The default is `Asymmetric`.

Methods

reset	Reset filter states
step	Decompose input with dyadic filter bank

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object

Common to All System Objects	
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Filter Square Wave Using Dyadic Filter Banks

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Denoise square wave input using dyadic analysis and synthesis filter banks.

```
t = 0:.0001:.0511;
x= square(2*pi*30*t);
xn = x' + 0.08*randn(length(x),1);
dydanl = dsp.DyadicAnalysisFilterBank;
```

The filter coefficients correspond to a `haar` wavelet.

```
dydanl.CustomLowpassFilter = [1/sqrt(2) 1/sqrt(2)];
dydanl.CustomHighpassFilter = [-1/sqrt(2) 1/sqrt(2)];
dydsyn = dsp.DyadicSynthesisFilterBank;
dydsyn.CustomLowpassFilter = [1/sqrt(2) 1/sqrt(2)];
dydsyn.CustomHighpassFilter = [1/sqrt(2) -1/sqrt(2)];
C = dydanl(xn);
```

Subband outputs.

```
C1 = C(1:256); C2 = C(257:384); C3 = C(385:512);
```

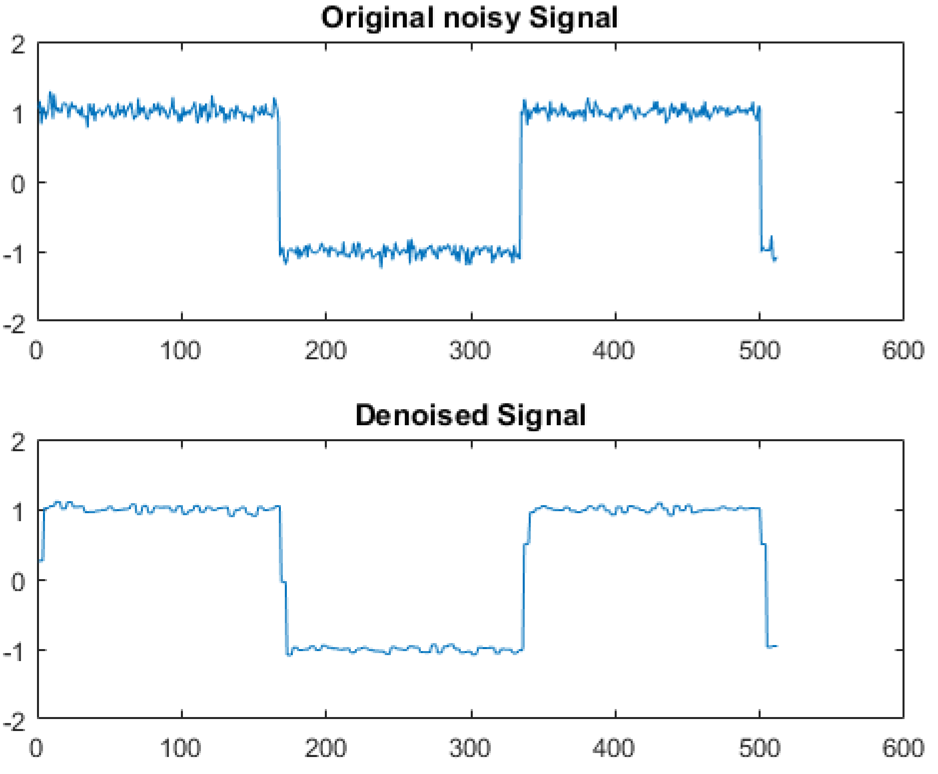
Set higher frequency coefficients to zero to remove the noise.

```
x_den = dydsyn([zeros(length(C1),1);...
    zeros(length(C2),1);C3]);
```

Plot the original and denoised signals.

```
subplot(2,1,1), plot(xn); title('Original noisy Signal');
```

```
subplot(2,1,2), plot(x_den); title('Denoised Signal');
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Dyadic Analysis Filter Bank` block reference page. The object properties correspond to the block parameters, except:
The dyadic analysis filter bank object always concatenates the subbands into a single column vector for a column vector input, or into the columns of a matrix for a matrix input. This behavior corresponds to the block's behavior when you set the **Output** parameter to `Single port`.

See Also

dsp.DyadicSynthesisFilterBank | dsp.SubbandAnalysisFilter

Introduced in R2012a

reset

System object: dsp.DyadicAnalysisFilterBank
Package: dsp

Reset filter states

Syntax

```
reset(dydan)
```

Description

`reset(dydan)` resets the filter states of the dyadic analysis filter bank object, `dydan`, to their initial values of zero. After the `step` method applies the dyadic analysis filter bank to nonzero input data, the filter states may change. Invoking the `step` method again without first invoking the `reset` method may produce different outputs for an identical input.

Reset a Dyadic Analysis Filter Bank

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
X = [1 1 7 9 2 8 8 6]';
dydan = dsp.DyadicAnalysisFilterBank('NumLevels',1);
```

Filter states are zero.

```
y = dydan(X);
```

Invoke step method again without resetting states.

```
y1 = dydan(X);
isequal(y,y1)
```

```
ans =
```

```
  logical

   0
```

Reset the filter states.

```
reset(dydan); % Reset filter states to zero
y2 = dydan(X);
isequal(y,y2)
```

```
ans =

  logical

   1
```

step

System object: dsp.DyadicAnalysisFilterBank
Package: dsp

Decompose input with dyadic filter bank

Syntax

`Y = step(dydan,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(dydan,X)` computes the subband decomposition of the input `X` and outputs the dyadic subband decomposition in `Y` as a single concatenated column vector or matrix of coefficients. Each column of `X` is treated as an independent input, and the number of rows of `X` must be a multiple of 2^N, where N is the value of the `NumLevels` property. The elements of `Y` are ordered with the highest frequency subband first followed by subbands in decreasing frequency

Examples

Subband Ordering For Asymmetric Tree Structure Using Dyadic Analysis Filter Bank

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Sampling frequency 1 kHz input length 1024

```
t = 0:.001:1.023;
x = square(2*pi*30*t);
xn = x' + 0.08*randn(length(x),1);
```

Default asymmetric structure with Daubechies order 3 extremal phase wavelet

```
dydan = dsp.DyadicAnalysisFilterBank;
Y = dydan(xn);
```

Level 2 yields 3 subbands (two detail-one approximation) Nyquist frequency is 500 Hz

```
D1 = Y(1:512); % subband approx. [250, 500] Hz
D2 = Y(513:768); % subband approx. [125, 250] Hz
Approx = Y(769:1024); % subband approx. [0,125] Hz
```

Subband Ordering For Symmetric Tree Structure Using Dyadic Analysis Filter Bank

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Sampling frequency 1 kHz input length 1024.

```
t = 0:.001:1.023;
x = square(2*pi*30*t);
xn = x' + 0.08*randn(length(x),1);
```

symmetric structure with Daubechies order 3 extremal phase wavelets.

```
dydan = dsp.DyadicAnalysisFilterBank('TreeStructure',...
'Symmetric');
Y = dydan(xn);
D1 = Y(1:256); % subband approx. [375,500] Hz
D2 = Y(257:512); % subband approx. [250,375] Hz
D3 = Y(513:768); % subband approx. [125,250] Hz
Approx = Y(769:1024); % subband approx. [0, 125] Hz
```

dsp.DyadicSynthesisFilterBank System object

Package: dsp

Reconstruct signals from subbands

Description

The `DyadicSynthesisFilterBank` object reconstructs signals from subbands with smaller bandwidths and lower sample rates. The filter bank uses a series of highpass and lowpass FIR filters to repeatedly reconstruct the signal.

To reconstruct signals from subbands with smaller bandwidths and lower sample rates:

1 Define and set up your synthesis filter bank. See "Construction" on page 4-629.
2 Call step to reconstruct the signal according to the properties of `dsp.DyadicSynthesisFilterBank`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`dydsyn = dsp.DyadicSynthesisFilterBank` returns a synthesis filter bank, `dydsyn`, that reconstructs a signal from its subbands with smaller bandwidths and smaller sample rates.

`dydsyn =`
`dsp.DyadicSynthesisFilterBank('`*`Propertyname`*`',`*`PropertyValue`*`,...)` returns a synthesis filter bank, `dydsyn`, with each specified property set to the specified value.

Properties

`Filter`

Type of filter used in filter bank

Specify the type of filter used to determine the highpass and lowpass FIR filters in the filter bank as one of `Custom`, `Haar`, `Daubechies`, `Symlets`, `Coiflets`, `Biorthogonal`, `Reverse Biorthogonal`, or `Discrete Meyer`. If you set this property to `Custom`, the `CustomLowpassFilter` on page 4-630 and `CustomHighpassFilter` on page 4-631 properties specify the filter coefficients. Otherwise, the object uses the Wavelet Toolbox `wfilters` function to construct the filters. Depending on the filter, the `WaveletOrder` on page 4-631 or `FilterOrder` on page 4-631 property might apply. For a list of the supported wavelets, see the following table.

Filter	Sample Setting for Related Filter Specification Properties	Corresponding Wavelet Toolbox Function Syntax
`Haar`	None	`wfilters('haar')`
`Daubechies`	`H.WaveletOrder = 4`	`wfilters('db4')`
`Symlets`	`H.WaveletOrder = 3`	`wfilters('sym3')`
`Coiflets`	`H.WaveletOrder = 1`	`wfilters('coif1')`
`Biorthogonal`	`H.FilterOrder = '[3/1]'`	`wfilters('bior3.1')`
`Reverse Biorthogonal`	`H.FilterOrder = '[3/1]'`	`wfilters('rbior3.1')`
`Discrete Meyer`	None	`wfilters('dmey')`

In order to automatically design wavelet-based filters, install the Wavelet Toolbox product. Otherwise, use the `CustomLowpassFilter` on page 4-630 and `CustomHighpassFilter` on page 4-631 properties to specify lowpass and highpass FIR filters.

`CustomLowpassFilter`

Lowpass FIR filter coefficients

Specify a vector of lowpass FIR filter coefficients, in descending powers of z. Use a half-band filter that passes the frequency band stopped by the filter specified in the

`CustomHighpassFilter` on page 4-631 property. This property applies only when you set the `Filter` on page 4-630 property to `Custom`. To perfectly reconstruct a signal decomposed by the `DyadicAnalysisFilterBank` on page 4-618, design the filters in the synthesis filter bank to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction is imperfect. The default values of this property specify a perfect reconstruction filter for the default settings of the analysis filter bank (based on a third-order Daubechies wavelet).

CustomHighpassFilter

Highpass FIR filter coefficients

Specify a vector of highpass FIR filter coefficients, in descending powers of z. Use a half-band filter that passes the frequency band stopped by the filter specified in the `CustomLowpassFilter` on page 4-630 property. This property applies only when you set the `Filter` on page 4-630 property to `Custom`. To perfectly reconstruct a signal decomposed by the `DyadicAnalysisFilterBank` on page 4-618, design the filters in the synthesis filter bank to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction is imperfect. The default values of this property specify a perfect reconstruction filter for the default settings of the analysis filter bank (based on a third-order Daubechies wavelet).

WaveletOrder

Wavelet order

Specify the order of the wavelet selected in the `Filter` on page 4-630 property. This property applies only when you set the `Filter` on page 4-630 property to `Daubechies`, `Symlets` or `Coiflets`. The default is `2`.

FilterOrder

Wavelet order for synthesis filter stage

Specify the order of the wavelet for the synthesis filter stage as:

- First order: `'[1/1]'`, `'[1/3]'`, or `'[1/5]'`.
- Second order: `'[2/2]'`, `'[2/4]'`, `'[2/6]'`, or `'[2/8]'`.
- Third order: `'[3/1]'`, `'[3/3]'`, `'[3/5]'`, `'[3/7]'`, or `'[3/9]'`.
- Fourth order: `'[4/4]'`.

- Fifth order: `'[5/5]'`.
- Sixth order: `'[6/8]'`.

This property applies only when you set the `Filter` on page 4-630 property to `Biorthogonal` or `Reverse Biorthogonal`. The default is `'[1/1]'`.

`NumLevels`

Number of filter bank levels

Specify the number of filter bank levels as a scalar integer. An N-level asymmetric structure has $N + 1$ input subbands, and an N-level symmetric structure has 2^N input subbands. The default is `2`.

rg

`TreeStructure`

Structure of filter bank

Specify the structure of the filter bank as `Asymmetric` or `Symmetric`. In the asymmetric structure, the low-frequency subband input to each level is the output of the previous level, while the high-frequency subband input to each level is an input to the filter bank. In the symmetric structure, both the low- and high-frequency subband inputs to each level are outputs from the previous level. The default is `Asymmetric`.

Methods

step	Reconstruct signal from high- and low-frequency subbands

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Filter Square Wave Using Dyadic Filter Banks

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Denoise square wave input using dyadic analysis and synthesis filter banks.

```
t = 0:.0001:.0511;
x= square(2*pi*30*t);
xn = x' + 0.08*randn(length(x),1);
dydanl = dsp.DyadicAnalysisFilterBank;
```

The filter coefficients correspond to a `haar` wavelet.

```
dydanl.CustomLowpassFilter = [1/sqrt(2) 1/sqrt(2)];
dydanl.CustomHighpassFilter = [-1/sqrt(2) 1/sqrt(2)];
dydsyn = dsp.DyadicSynthesisFilterBank;
dydsyn.CustomLowpassFilter = [1/sqrt(2) 1/sqrt(2)];
dydsyn.CustomHighpassFilter = [1/sqrt(2) -1/sqrt(2)];
C = dydanl(xn);
```

Subband outputs.

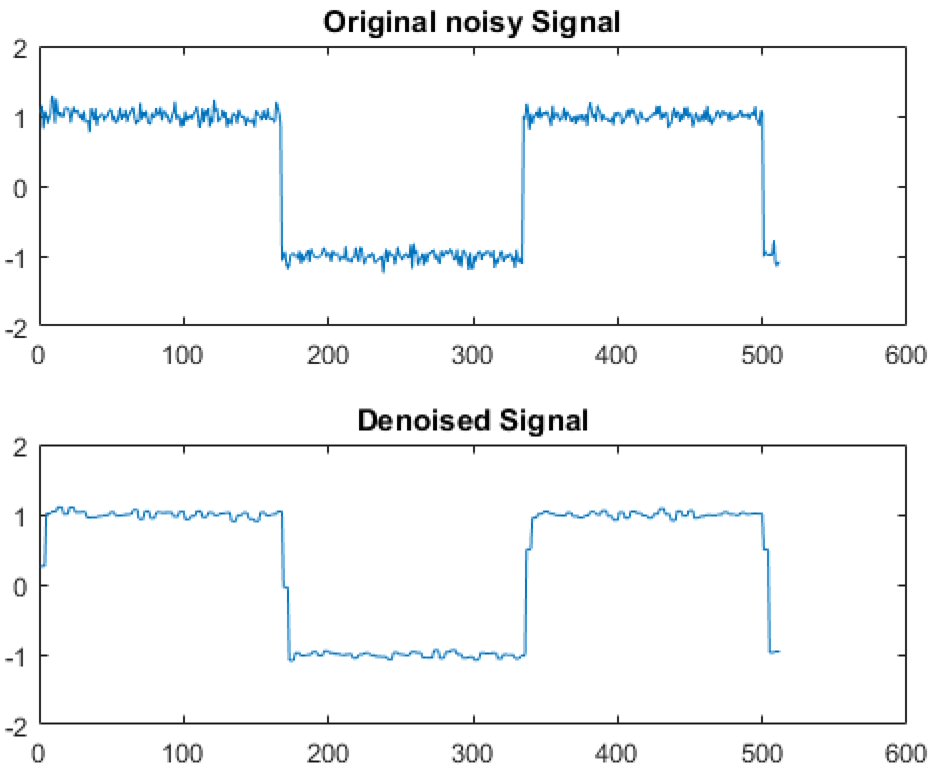
```
C1 = C(1:256); C2 = C(257:384); C3 = C(385:512);
```

Set higher frequency coefficients to zero to remove the noise.

```
x_den = dydsyn([zeros(length(C1),1);...
    zeros(length(C2),1);C3]);
```

Plot the original and denoised signals.

```
subplot(2,1,1), plot(xn); title('Original noisy Signal');
subplot(2,1,2), plot(x_den); title('Denoised Signal');
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Dyadic Synthesis Filter Bank` block reference page. The object properties correspond to the block parameters, except:
The object only receives data as a vector or matrix of concatenated subbands, as specified using the `step` method.

See Also

dsp.DyadicAnalysisFilterBank | dsp.SubbandSynthesisFilter

Introduced in R2012a

step

System object: dsp.DyadicSynthesisFilterBank
Package: dsp

Reconstruct signal from high- and low-frequency subbands

Syntax

`X = step(dydsyn,S)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`X = step(dydsyn,S)` reconstructs the concatenated subband input `S` to output `X`. Each column of input `S` contains the subbands for an independent signal. Upper rows contain the high-frequency subbands, and lower rows contain the low-frequency subbands. The number of rows of `S` must be a multiple of 2^N, where N is the value of the `NumLevels` property.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.FarrowRateConverter System object

Package: dsp

Polynomial sample rate converter with arbitrary conversion factor

Description

The `FarrowRateConverter` System object implements an efficient polynomial-fit sample rate conversion filter using a Farrow structure. You can use this object to convert the sample rate of a signal up or down by an arbitrary factor. This object supports fixed-point operations.

To convert the sample rate of a signal:

1. Define and set up your sample rate converter. See "Construction" on page 4-637.
2. Call step to convert the sample rate according to the properties of `dsp.FarrowRateConverter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`src = dsp.FarrowRateConverter` creates a polynomial filter-based sample rate converter System object, `src`. For each channel of an input signal, `src` converts the input sample rate to the output sampling rate.

`src = dsp.FarrowRateConverter(Name,Value)` uses additional rate and filter properties, specified by one or more `Name,Value` pair arguments.

`src = dsp.FarrowRateConverter(FSIN,FSOUT,TOL,NP)` returns a sample rate converter object, `src`, with `InputSampleRate` property set to `FSIN`, OutputSampleRate property set to FSOUT, OutputRateTolerance property set to TOL, and PolynomialOrder property set to NP.

Properties

InputSampleRate — Sample rate of input signal

44100 Hz (default) | positive scalar in Hz

Sample rate of the input signal, specified as a positive scalar in Hz. The input sample rate must be greater than the bandwidth of interest. The default is 44.1 kHz.

OutputSampleRate — Sample rate of output signal

48000 Hz (default) | positive scalar in Hz

Sample rate of the output signal, specified as a positive scalar in Hz. The output sample rate can be higher or lower than the input sample rate. The default is 48 kHz.

OutputRateTolerance — Maximum allowed tolerance for output sample rate

0 (default) | positive scalar

Maximum allowed tolerance for the sample rate of the output signal, specified as a positive scalar from 0 through 0.5, inclusive. The default is 0.

The actual output sample rate varies but is within the specified range. For example, if OutputRateTolerance is specified as 0.01, then the actual output sample rate is in the range given by OutputSampleRate ± 1%. This flexibility allows for a simpler filter design often.

Specification — Polynomial specification

'Polynomial order' (default) | 'Coefficients'

Method used to specify the polynomial interpolator coefficients, specified as one of the following:

- 'Polynomial order' — Specify the order of the Lagrange interpolation filter polynomial through the PolynomialOrder property.
- 'Coefficients' — Specify the polynomial coefficients directly through the Coefficients property.

PolynomialOrder — Order of filter polynomial

3 (default) | positive integer less than or equal to 4

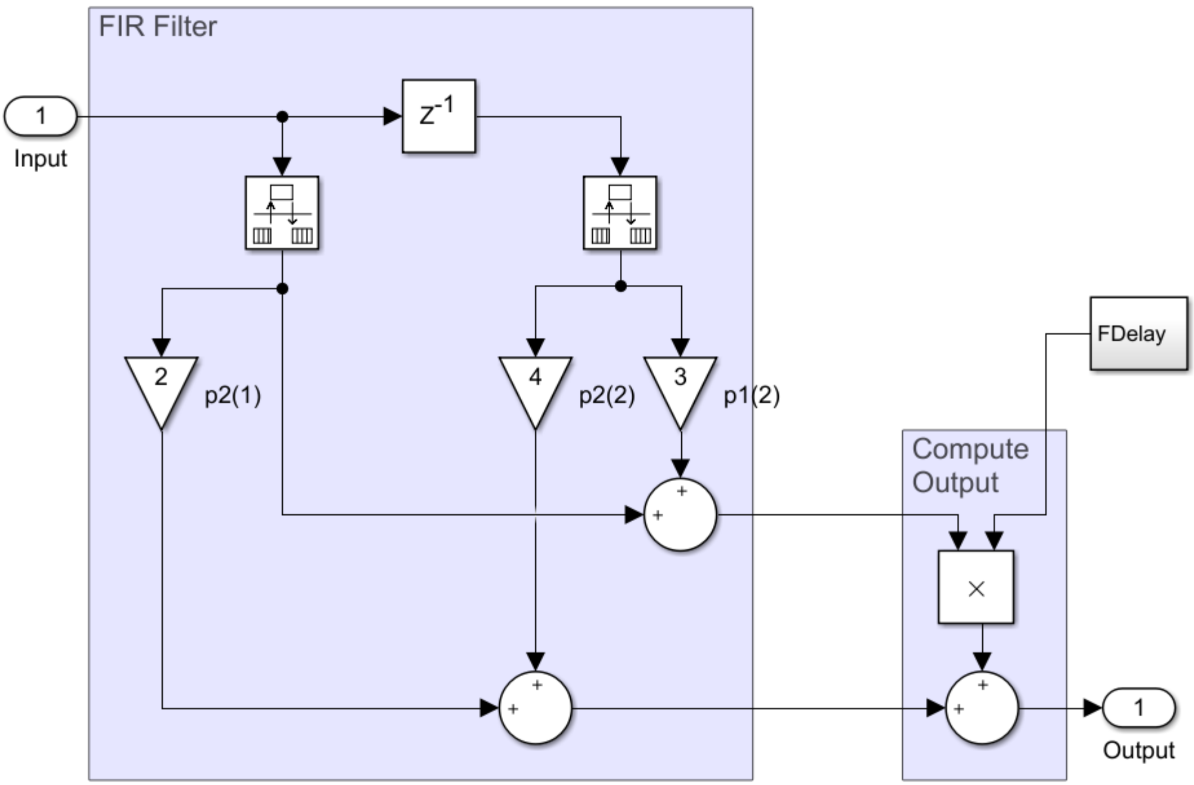
Order of the Lagrange interpolation filter polynomial, specified as a positive integer less than or equal to 4. The default is 3. This property applies only when you set `Specification` to `'Polynomial order'`.

`Coefficients` — Filter polynomial coefficients

`[-1 1; 1 0]` (default) | real-valued square matrix

Filter polynomial coefficients, specified as a real-valued square matrix. The default is `[-1 1; 1 0]`. This property applies only when you set `Specification` to `'Coefficients'`.

Here is the signal flow graph for farrow rate converter with coefficients set to `[1 2; 3 4]`.



Each branch of the FIR filter corresponds to each row of the coefficients matrix.

Fixed-Point Properties

`RoundingMethod` — Rounding method for fixed-point operations

`'Ceiling'` | `'Convergent'` | `'Floor'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding method, specified as a character vector. For more information on the rounding modes, see “Precision and Range”.

This property is not tunable.

`OverflowAction` — Overflow action for fixed-point operations

`'Wrap'` | `'Saturate'`

Overflow action, specified as `Wrap|Saturate`. For more details on the overflow modes, see the 'Overflow Handling' section of “Precision and Range”.

This property is not tunable.

`CoefficientsDataType` — Data type of the filter coefficients

`numerictype(1,16)` | `numerictype(1,16,3)`

Data type of the filter coefficients, specified as a signed `numerictype` object. The default, `numerictype(1,16)` corresponds to a signed numeric type object with 16-bit coefficients. To give the best possible precision, the fraction length of this data type is determined based on the coefficient values.

This property is not tunable.

`FractionalDelayDataType` — Data type of the fractional delay

`numerictype(0,8)` | `numerictype(0,16)`

Data type of the fractional delay, specified as an unsigned `numerictype` object. The default, `numerictype(0,8)` corresponds to an unsigned fixed-point fractional delay data type object with 8-bit word length. To give the best possible precision, the fraction length of this data type is determined based on the fractional delay values.

This property is not tunable.

`MultiplicandDataType` — Data type of the multiplicand

`numerictype(1,16,13)`

Data type of the multiplicand, specified as a signed `numerictype` object. The default, `numerictype(1,16,13)` corresponds to a signed fixed-point multiplicand data type with 16-bit word length and 13-bit fraction length.

This property is not tunable.

`OutputDataType` — Data type of the farrow rate converter output

`'Same word length as input'` | `'Same as accumulator'` | `numerictype(1,16)` | `numerictype(1,16,0)`

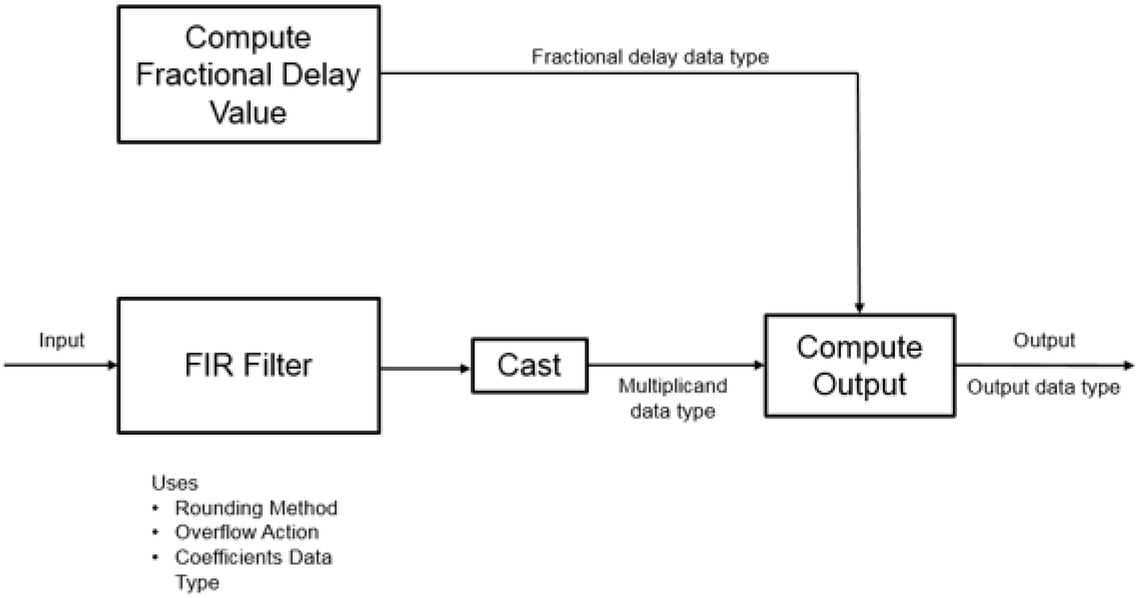
Data type of the output data type, specified as one of the following:

- `Same word length as input` (default) — Output word length and fraction lengths are the same as the input.
- `Same as accumulator` — Output word length and fraction lengths are the same as the accumulator.
- `numerictype(1,16)` — Signed fixed-point output data type with 16-bit word length. To give the best possible precision, the fraction length is computed based on the input range. The dynamic range of the input is preserved.
- `numerictype(1,16,0)` — Signed fixed-point output data type with 16-bit word length and zero fraction length.

This property is not tunable.

Fixed-Point Data Types

The signal flow diagram shows the data types used within the farrow rate converter for fixed-point signals and floating-point signals. You can specify these data types in the Farrow rate converter. If the input is floating point, all the data types with in the signal flow diagram have the same data type as the input, which can be single or double.



If the input is fixed point, the FIR filter uses the rounding mode, overflow mode, and the coefficients data type data you provide to the Farrow rate converter. The accumulator data type and the product data type in the signal flow are full precision.

Methods

cost	Implementation cost
freqz	Frequency response
getPolynomialCoefficients	Polynomial filter coefficients
getActualOutputRate	Actual output rate, accounting for tolerance
getRateChangeFactors	Overall interpolation and decimation factors
info	Filter implementation details
reset	Reset internal states of polynomial sample rate converter
step	Convert sample rate of signal

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Upsample an Audio Signal Using Farrow Sample Rate Converter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create a Farrow rate converter object to convert an audio signal from 44.1 kHz to 96 kHz. Set the polynomial order for the Farrow filter.

```
fs1 = 44.1e3;
fs2 = 96e3;
LagrangeOrder = 2; % 1 = linear interpolation
frc = dsp.FarrowRateConverter('InputSampleRate',fs1,...
                              'OutputSampleRate',fs2,...
                              'PolynomialOrder',LagrangeOrder);
ar = dsp.AudioFileReader('guitar10min.ogg','SamplesPerFrame',14700);
aw = dsp.AudioFileWriter('guitar10min_96kHz.wav','SampleRate',fs2);
```

Check the resulting interpolation (L) and decimation (M) factors.

```
[L,M] = getRateChangeFactors(frc)
```

```
L =

   320


M =

   147
```

Display the polynomial the object uses to fit the input samples.

```
coeffs = getPolynomialCoefficients(frc)
```

```
coeffs =

    0.5000   -0.5000         0
   -1.0000         0    1.0000
    0.5000    0.5000         0
```

Convert 100 frames of the audio signal. Write the result to a file.

```
for n = 1:1:100
   x = ar();
   y = frc(x);
   aw(y);
end
```

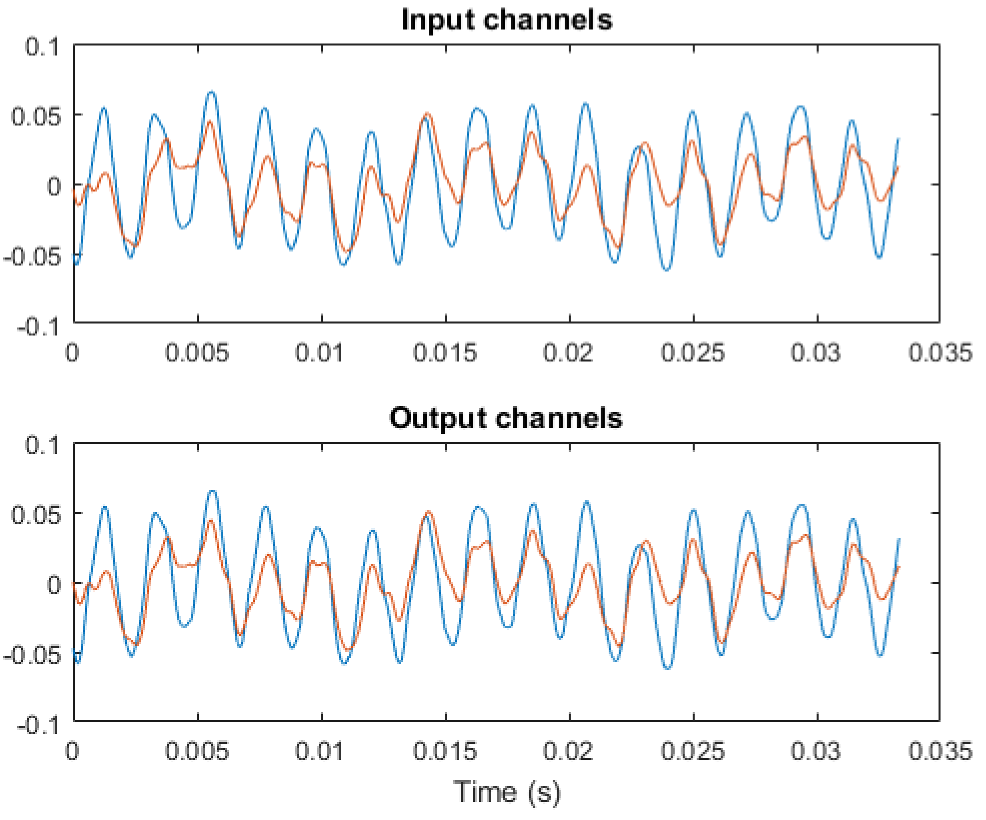
Release the `AudioFileWriter` object to complete creation of the output file.

```
release(aw)
release(ar)
```

Plot the input and output signals of the 100th frame of data. Delay the input to compensate for the latency introduced by the filter.

```
t1 = 0:1/fs1:1/30-1/fs1;
t2 = 0:1/fs2:1/30-1/fs2;

delay = ceil((LagrangeOrder+1)/2)/fs1;
el1 = 1:length(t1)-delay;
el2 = 1:length(t2);
el2(1:delay) = [];

figure

subplot(2,1,1)
plot(t1(1:length(el1)),x(el1,1))
hold on
plot(t1(1:length(el1)),x(el1,2))
title('Input channels')

subplot(2,1,2)
plot(t2(1:length(el2)),y(el2,1))
```

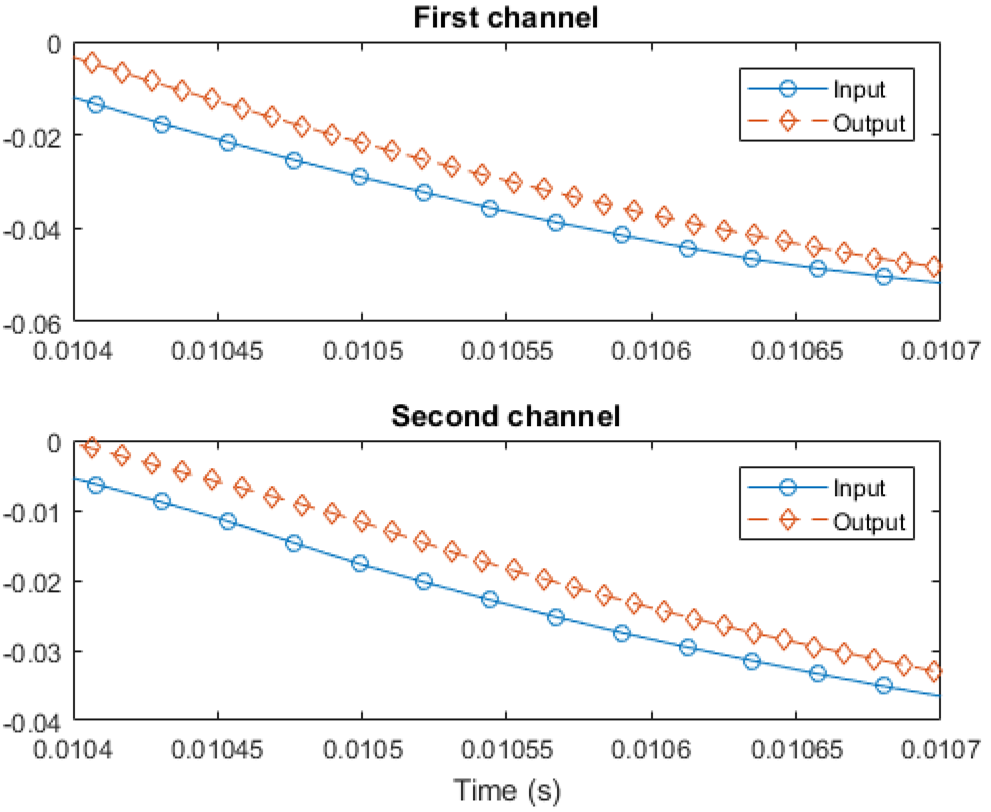
```
hold on
plot(t2(1:length(el2)),y(el2,2))
xlabel('Time (s)')
title('Output channels')
```



Zoom in to see the difference in sample rates.

```
figure

subplot(2,1,1)
plot(t1(1:length(el1)),x(el1,1),'o-')
hold on
plot(t2(1:length(el2)),y(el2,1),'d--')
xlim([0.0104 0.0107])
title('First channel')
```

```
legend('Input','Output')

subplot(2,1,2)
plot(t1(1:length(el1)),x(el1,2),'o-')
hold on
plot(t2(1:length(el2)),y(el2,2),'d--')
xlim([0.0104 0.0107])
xlabel('Time (s)')
title('Second channel')
legend('Input','Output')
```



Using Output Tolerance To Reduce Input Size Restriction

Create a Farrow rate converter with 0% tolerance. The output rate matches `OutputSampleRate` exactly. However, the input size must be a multiple of the decimation factor, M. In this case M is 320.

```
frc = dsp.FarrowRateConverter('InputSampleRate', 96e3,'OutputSampleRate',44.1e3);
FsOut = getActualOutputRate(frc)
[L,M] = getRateChangeFactors(frc)
```

```
FsOut =

       44100
```

```
L =

   147


M =

   320
```

Allow a 1% tolerance on the output rate and observe the difference in decimation factor. M is now only 13. The lower decimation factor allows more flexibility in input size. The output rate is within the range `OutputSampleRate` ± 1%.

```
frc.OutputRateTolerance = 0.01;
FsOut = getActualOutputRate(frc)
[Le,Me] = getRateChangeFactors(frc)
```

```
FsOut =

   4.4308e+04


Le =

     6


Me =

    13
```

- “Efficient Sample Rate Conversion Between Arbitrary Factors”

References

Hentschel, T., and G. Fettweis. “Continuous-Time Digital Filters for Sample-Rate Conversion in Reconfigurable Radio Terminals.” *Frequenz*. Volume 55, Issue 5-6, 2001, pp. 185–188.

Algorithm

Farrow filters implement piecewise polynomial interpolation using Horner's rule to compute samples from the polynomial. The polynomial coefficients used to fit the input samples correspond to the Lagrange interpolation coefficients.

Once a polynomial is fitted to the input data, the value of the polynomial can be calculated at any point. Therefore, a polynomial filter allows for interpolation at arbitrary locations between input samples.

You can use a polynomial of any order to fit to the existing samples. However, since large order polynomials oscillate a lot, polynomials of order 1, 2, 3, or 4 are used in practice.

The block computes interpolated values at the desired locations by varying only the fractional interval, μ. This interval is the distance between the previous input sample and the current output sample. All filter coefficients remain constant.

- The input samples, x, are filtered using $M + 1$ FIR filters, where M is the polynomial order.
- The outputs of these filters are multiplied by the fractional delay, μ.
- The output, y, is the sum of the multiplication results.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

dsp.FIRRateConverter | dsp.SampleRateConverter

Topics

"Efficient Sample Rate Conversion Between Arbitrary Factors"

Introduced in R2014b

cost

System object: dsp.FarrowRateConverter
Package: dsp

Implementation cost

Syntax

`c = cost(src)`

Description

`c = cost(src)` returns a structure, `c`, whose fields contain information about the computational cost of implementing a polynomial sample rate converter, `src`.

Input Arguments

`src` — Polynomial sample rate converter

`FarrowRateConverter` System object

Polynomial sample rate converter, specified as a `FarrowRateConverter` System object.

Output Arguments

`c` — Output structure

structure

Output structure containing information about the computational cost of `src`, including the number of:

- Coefficients
- States
- Multiplications per unit sample

- Additions per unit sample

Examples

Computational Cost of Farrow Rate Converter

Create a polynomial sample rate converter with default values. Determine its computational cost: the number of coefficients, states, multiplications per unit sample, and additions per unit sample.

```
frc = dsp.FarrowRateConverter;
cst = cost(frc)
```

```
cst = 

  struct with fields:

                  NumCoefficients: 16
                        NumStates: 3
    MultiplicationsPerInputSample: 13.0612
          AdditionsPerInputSample: 11.9728
```

Repeat the computation, allowing for a 10% tolerance in the output sample rate.

```
frc.OutputRateTolerance = 0.1;
ctl = cost(frc)
```

```
ctl = 

  struct with fields:

                  NumCoefficients: 16
                        NumStates: 3
    MultiplicationsPerInputSample: 12
          AdditionsPerInputSample: 11
```

freqz

System object: dsp.FarrowRateConverter
Package: dsp

Frequency response

Syntax

```
[h,fout] = freqz(src,n,range)
[h,fout] = freqz(src,fin)
```

Description

`[h,fout] = freqz(src,n,range)` returns the complex frequency response, `h`, of the polynomial sample rate converter, `src`, evaluated at the `n` frequencies returned in `fout`. `range` is the frequency range over which the response is computed. The sample rate is taken to be the largest of `InputSampleRate` and `OutputSampleRate`.

`[h,fout] = freqz(src,fin)` returns the complex frequency response evaluated at the frequency points specified in the vector `fin`. `fin` is assumed to be expressed in hertz.

Input Arguments

`src` — Polynomial sample rate converter

`FarrowRateConverter` System object

Polynomial sample rate converter, specified as a `FarrowRateConverter` System object.

`n` — Number of evaluation points

512 (default) | positive integer

Number of frequencies for response evaluation, specified as a positive integer scalar. If not specified, `n` defaults to 512.

`range` — Range of frequencies

`'half'` (default) | `'whole'`

Range of frequencies considered when computing the frequency response, specified as either `'half'` (from 0 to π) or `'whole'` (from 0 to 2π). If not specified, `range` defaults to `'half'`.

`fin` — Frequencies
vector

Frequencies at which the response is evaluated, specified as a vector.

Output Arguments

`h` — Complex frequency response
vector

Complex frequency response, returned as a vector.

`fout` — Frequencies
vector

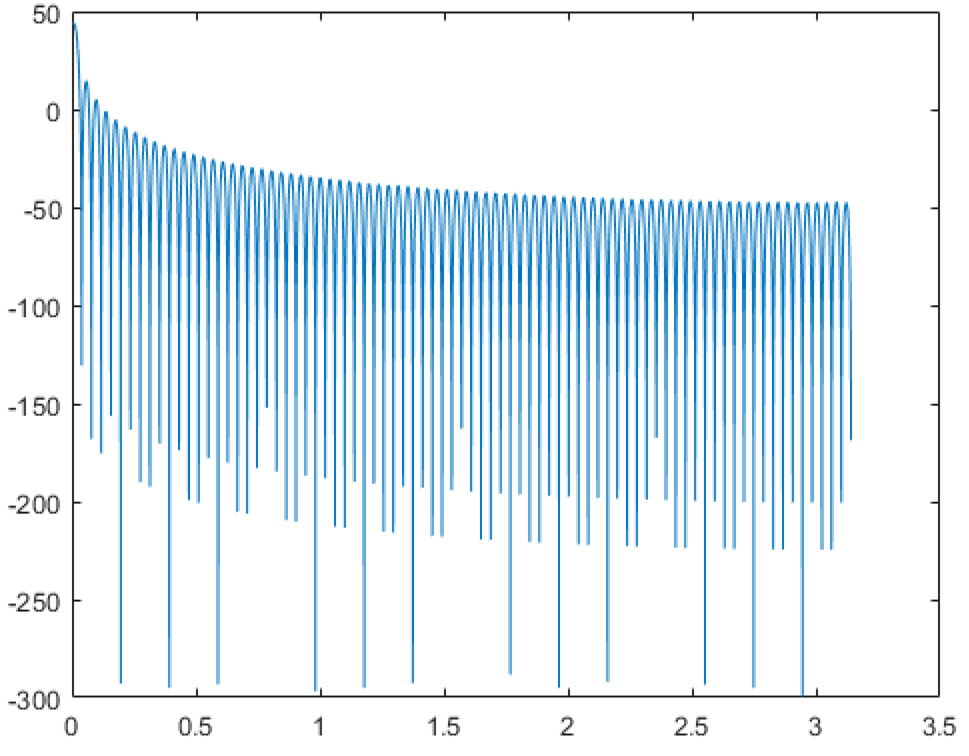
Frequencies at which the response is evaluated, returned as a vector.

Examples

Frequency Response of Farrow Rate Converter

Create a polynomial sample rate converter with default properties. Compute and display the frequency response.

```
frc = dsp.FarrowRateConverter;
[h,f] = freqz(frc);
plot(f,20*log10(abs(h)))
```


50
0
-50
-100
-150
-200
-250
-300
0
0.5
1
1.5
2
2.5
3
3.5

getPolynomialCoefficients

System object: dsp.FarrowRateConverter
Package: dsp

Polynomial filter coefficients

Syntax

```
[c] = getPolynomialCoefficients(src)
```

Description

`[c] = getPolynomialCoefficients(src)` returns the polynomial coefficients that the polynomial filter sample rate converter, `src`, uses to achieve the requested rate conversion.

Input Arguments

`src` — Polynomial filter sample rate converter

`FarrowRateConverter` System object

Polynomial sample rate converter, specified as a `FarrowRateConverter` System object.

Output Arguments

`c` — Polynomial filter coefficients

matrix

Polynomial filter coefficients, returned as a matrix.

Examples

Polynomial Coefficients of the Default Farrow Rate Converter

Create a default Farrow Rate Converter object to convert a signal from 44.1 kHz to 48 kHz.

```
FRC = dsp.FarrowRateConverter()
```

```
FRC =

  dsp.FarrowRateConverter with properties:

   Main
        InputSampleRate: 44100
       OutputSampleRate: 48000
    OutputRateTolerance: 0
          Specification: 'Polynomial order'
        PolynomialOrder: 3

  Use get to show all properties
```

Check the self-designed polynomial.

```
coeffs = getPolynomialCoefficients(FRC)
```

```
coeffs =

   -0.1667    0.5000   -0.3333         0
    0.5000   -1.0000   -0.5000    1.0000
   -0.5000    0.5000    1.0000         0
    0.1667         0   -0.1667         0
```

getActualOutputRate

System object: dsp.FarrowRateConverter
Package: dsp

Actual output rate, accounting for tolerance

Syntax

```
fsout = getActualOutputRate(src)
```

Description

`fsout = getActualOutputRate(src)` returns the actual output sample rate yielded by a `FarrowRateConverter` System object, accounting for the `OutputRateTolerance` parameter.

Input Arguments

`src` — Polynomial filter sample rate converter

`FarrowRateConverter` System object

Polynomial sample rate converter, specified as a `FarrowRateConverter` System object.

Output Arguments

`fsout` — Actual output sample rate

scalar

Actual output sample rate, returned as a scalar expressed in hertz.

Examples

Output Sample Rate of Farrow Rate Converter with Given Tolerance

Get the actual output sample rate for conversion between 44.1 kHz and 48 kHz, given a tolerance of 1%.

```
frc = dsp.FarrowRateConverter();
frc.OutputRateTolerance = 0.01;
FsOut = getActualOutputRate(frc)
```

```
FsOut =

   4.8109e+04
```

getRateChangeFactors

System object: dsp.FarrowRateConverter
Package: dsp

Overall interpolation and decimation factors

Syntax

```
[L,M] = getRateChangeFactors(src)
```

Description

`[L,M] = getRateChangeFactors(src)` returns the overall interpolation factor, `L`, and the overall decimation factor, `M`, corresponding to the polynomial filter sample rate converter, `src`. The overall decimation factor affects the allowable frame size of the input signal, which must be an integer multiple of `M`.

Input Arguments

`src` — Polynomial sample rate converter

`FarrowRateConverter` System object

Polynomial sample rate converter, specified as a `FarrowRateConverter` System object.

Output Arguments

`L` — Overall interpolation factor

scalar

Overall interpolation factor, returned as a scalar.

`M` — Overall decimation factor

scalar

Overall decimation factor, returned as a scalar.

Examples

Resampling Factors of Farrow Rate Converter

Create a default Farrow rate converter object to convert a signal from 44.1 kHz to 48 kHz. Determine its overall interpolation (L) and decimation (M) factors.

```
frc = dsp.FarrowRateConverter
[L,M] = getRateChangeFactors(frc)
```

```
frc = 

  dsp.FarrowRateConverter with properties:

   Main
        InputSampleRate: 44100
       OutputSampleRate: 48000
    OutputRateTolerance: 0
          Specification: 'Polynomial order'
        PolynomialOrder: 3

  Use get to show all properties


L =

   160


M =

   147
```

info

System object: dsp.FarrowRateConverter
Package: dsp

Filter implementation details

Syntax

`info(src)`

Description

`info(src)` displays information about the `FarrowRateConverter` System object, `src`.

Input Arguments

`src` — Polynomial sample rate converter
`FarrowRateConverter` System object

Polynomial sample rate converter, specified as a `FarrowRateConverter` System object.

Examples

Displaying Filter Information for Farrow Rate Converter

Use the `info` method to display the filter implementation details of a Farrow rate converter object.

```
frc = dsp.FarrowRateConverter;
info(frc)
```

```
ans =

  10×52 char array
```

```
'Discrete-Time FIR Multirate Filter (real)          '
'-----------------------------------------          '
'Filter Structure      : Farrow Sample-Rate Converter'
'Interpolation Factor  : 160                        '
'Decimation Factor     : 147                        '
'Filter Length         : 4                          '
'Stable                : Yes                        '
'Linear Phase          : No                         '
'                                                   '
'Arithmetic            : double                     '
```

reset

System object: dsp.FarrowRateConverter
Package: dsp

Reset internal states of polynomial sample rate converter

Syntax

`reset(src)`

Description

`reset(src)` resets the internal states of a polynomial `FarrowRateConverter` System object, `src`, to their initial values.

Input Arguments

`src` — Polynomial sample rate converter
`FarrowRateConverter` System object

Polynomial sample rate converter, specified as a `FarrowRateConverter` System object.

step

System object: dsp.FarrowRateConverter
Package: dsp

Convert sample rate of signal

Syntax

`y = step(src,x)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(src,x)` resamples input `x` to create output `y` according to the rate conversion defined by `src`.

Input Arguments

`src` — Polynomial sample rate converter

`FarrowRateConverter` System object

Polynomial sample rate converter, specified as a `FarrowRateConverter` System object.

`x` — Input signal

vector | matrix

Input signal, specified as a vector or matrix. The row length of `x` must be a multiple of the overall decimation factor. Each column of `x` is treated as a separate channel.

Output Arguments

y — Resampled signal

vector | matrix

Resampled signal, returned as a vector or matrix.

Examples

Upsample an Audio Signal Using Farrow Sample Rate Converter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create a Farrow rate converter object to convert an audio signal from 44.1 kHz to 96 kHz. Set the polynomial order for the Farrow filter.

```
fs1 = 44.1e3;
fs2 = 96e3;
LagrangeOrder = 2; % 1 = linear interpolation
frc = dsp.FarrowRateConverter('InputSampleRate',fs1,...
                              'OutputSampleRate',fs2,...
                              'PolynomialOrder',LagrangeOrder);
ar = dsp.AudioFileReader('guitar10min.ogg','SamplesPerFrame',14700);
aw = dsp.AudioFileWriter('guitar10min_96kHz.wav','SampleRate',fs2);
```

Check the resulting interpolation (L) and decimation (M) factors.

```
[L,M] = getRateChangeFactors(frc)
```

```
L =

   320


M =

   147
```

Display the polynomial the object uses to fit the input samples.

```
coeffs = getPolynomialCoefficients(frc)
```

```
coeffs =

    0.5000   -0.5000         0
   -1.0000         0    1.0000
    0.5000    0.5000         0
```

Convert 100 frames of the audio signal. Write the result to a file.

```
for n = 1:1:100
   x = ar();
   y = frc(x);
   aw(y);
end
```

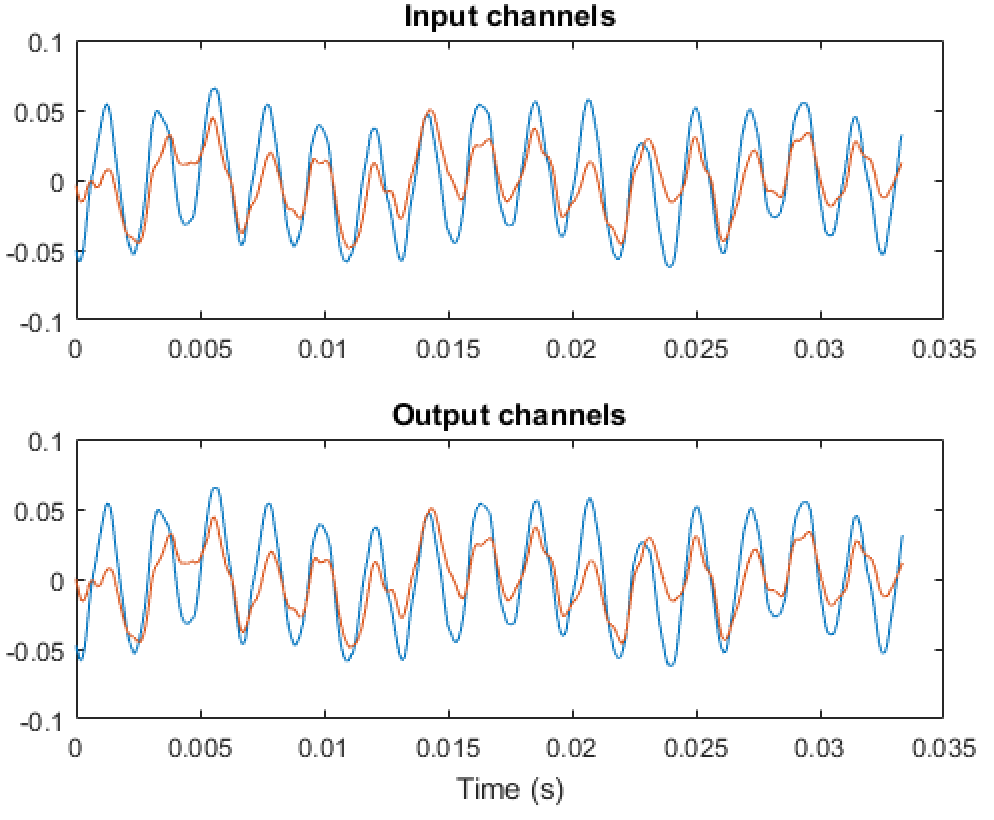
Release the `AudioFileWriter` object to complete creation of the output file.

```
release(aw)
release(ar)
```

Plot the input and output signals of the 100th frame of data. Delay the input to compensate for the latency introduced by the filter.

```
t1 = 0:1/fs1:1/30-1/fs1;
t2 = 0:1/fs2:1/30-1/fs2;

delay = ceil((LagrangeOrder+1)/2)/fs1;
el1 = 1:length(t1)-delay;
el2 = 1:length(t2);
el2(1:delay) = [];

figure

subplot(2,1,1)
plot(t1(1:length(el1)),x(el1,1))
hold on
plot(t1(1:length(el1)),x(el1,2))
title('Input channels')

subplot(2,1,2)
```

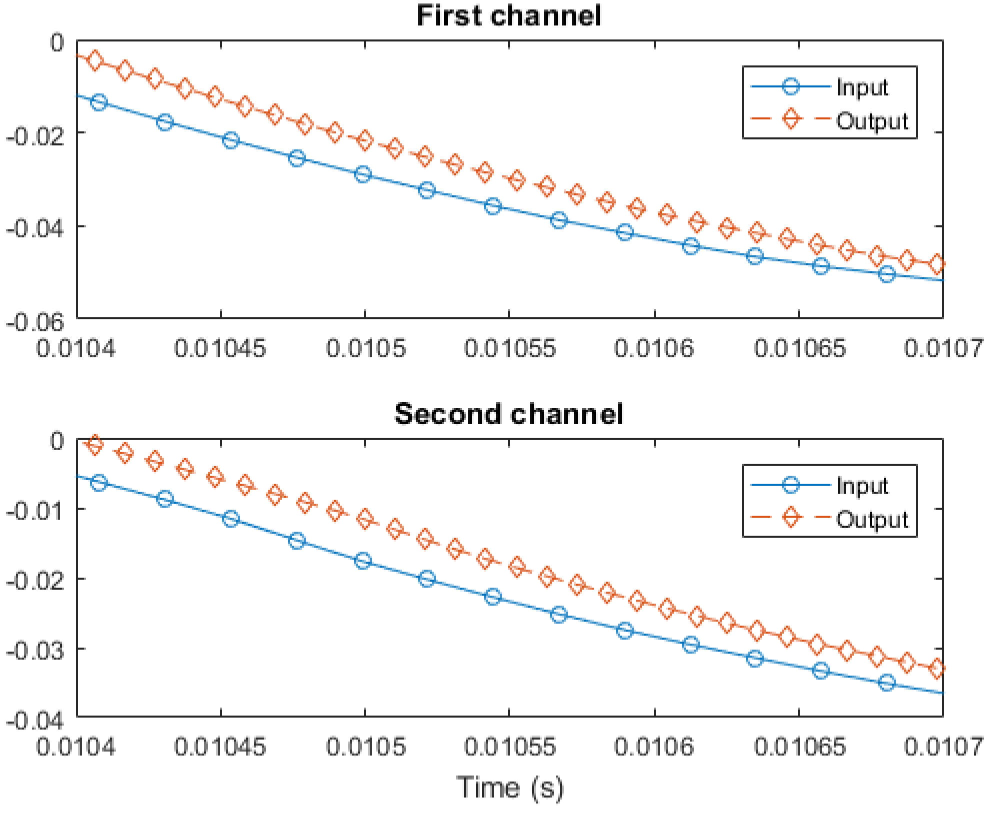
```
plot(t2(1:length(el2)),y(el2,1))
hold on
plot(t2(1:length(el2)),y(el2,2))
xlabel('Time (s)')
title('Output channels')
```



Zoom in to see the difference in sample rates.

```
figure

subplot(2,1,1)
plot(t1(1:length(el1)),x(el1,1),'o-')
hold on
plot(t2(1:length(el2)),y(el2,1),'d--')
```

```
xlim([0.0104 0.0107])
title('First channel')
legend('Input','Output')

subplot(2,1,2)
plot(t1(1:length(el1)),x(el1,2),'o-')
hold on
plot(t2(1:length(el2)),y(el2,2),'d--')
xlim([0.0104 0.0107])
xlabel('Time (s)')
title('Second channel')
legend('Input','Output')
```



dsp.FastTransversalFilter System object

Package: dsp

Fast Transversal filter

Description

The `dsp.FastTransversalFilter` computes output, error and coefficients using a fast transversal least-squares FIR adaptive filter.

To implement the adaptive FIR filter object:

1. Define and set up your adaptive FIR filter object. See “Construction” on page 4-670.
2. Call step to implement the filter according to the properties of `dsp.FastTransversalFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ftf = dsp.FastTransversalFilter` returns a System object, `ftf`, which is a fast transversal, least-squares FIR adaptive filter. This System object is used to compute the filtered output and the filter error for a given input and desired signal.

`ftf = dsp.FastTransversalFilter('`*`PropertyName`*`',` *`PropertyValue`*`,...)` returns a `FastTransversalFilter` System object, `ftf`, with each specified property set to the specified value.

`ftf =`
`dsp.FastTransversalFilter(LEN,'`*`PropertyName`*`',`*`PropertyValue`*`,...)`

returns a `FastTrasversalFilter` System object, `ftf`. In this case, the Length property set to `LEN`, and other specified properties set to the specified values.

Properties

`Method`

Method to calculate filter coefficients

Specify the method used to calculate filter coefficients as one of `'Fast transversal least-squares'` |`'Sliding-window fast transversal least-squares'`. The default value is `'Fast transversal least-squares'`. For algorithms used to implement these three different methods, refer to [1]. This property is nontunable.

`Length`

Length of filter coefficients vector

Specify the length of the FIR filter coefficients vector as a positive integer value. This property is nontunable.

The default value is 32.

`SlidingWindowBlockLength`

Width of sliding window

Specify the width of the sliding window as a positive integer value greater than or equal to the `Length` property value. This property applies only if the `Method` property is set to `'Sliding-window fast transversal least-squares'`. The default vale is the value of the `Length` property. This property is nontunable.

`ForgettingFactor`

Fast transversal filter forgetting factor

Specify the fast transversal filter forgetting factor as a positive numeric value. Setting this value to 1 denotes infinite memory while adaptation. Setting this property value to 1 denotes infinite memory while adapting to find the new filter. For best results, set this property to a value that lies in the range `[1-0.5/L, 1]`, where `L` is the Length property value. This property applies only if the Method property is set to `'Fast transversal least-squares'`. The default value is `1`. This property is tunable.

InitialPredictionErrorPower

Initial prediction error power

Specify the initial value of the forward and backward prediction error vectors as a positive numeric scalar. This scalar should be sufficiently large to maintain stability and prevent an excessive number of Kalman gain rescues. The default value is `10`.

This property is tunable.

InitialConversionFactor

Initial conversion factor (gamma)

Specify the initial value of the conversion factor of the fast transversal filter. If the `Method` property is set to `'Fast transversal least-squares'`, this property must be a positive numeric value less than or equal to `1`. In this case, the default value is `1`. If the `Method` property is set to `'Sliding-window fast transversal least-squares'`, this property must be a 2-element numeric vector. The first element of this vector must lie within the range `[0,1]`, and the second element must be less than or equal to `-1`. In this case, the default value is `[1, -1]`.

This property is tunable.

InitialCoefficients

Initial coefficients of the filter

Specify the initial values of the FIR adaptive filter coefficients as a scalar or a vector of length equal to the value of the `Length` property. The default value is `0`.

This property is tunable.

LockCoefficients

Locked status of the coefficient updates

Specify whether to lock the filter coefficient values. By default, the value of this property is `false`, and the object continuously updates the filter coefficients. If this property is set to `true`, the filter coefficients do not update and their values remain the same.

This property is tunable.

Methods

msesim	Mean-square error for Fast Transversal filter
reset	Reset filter states for Fast Transversal filter
step	Apply Fast Transversal filter to input

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

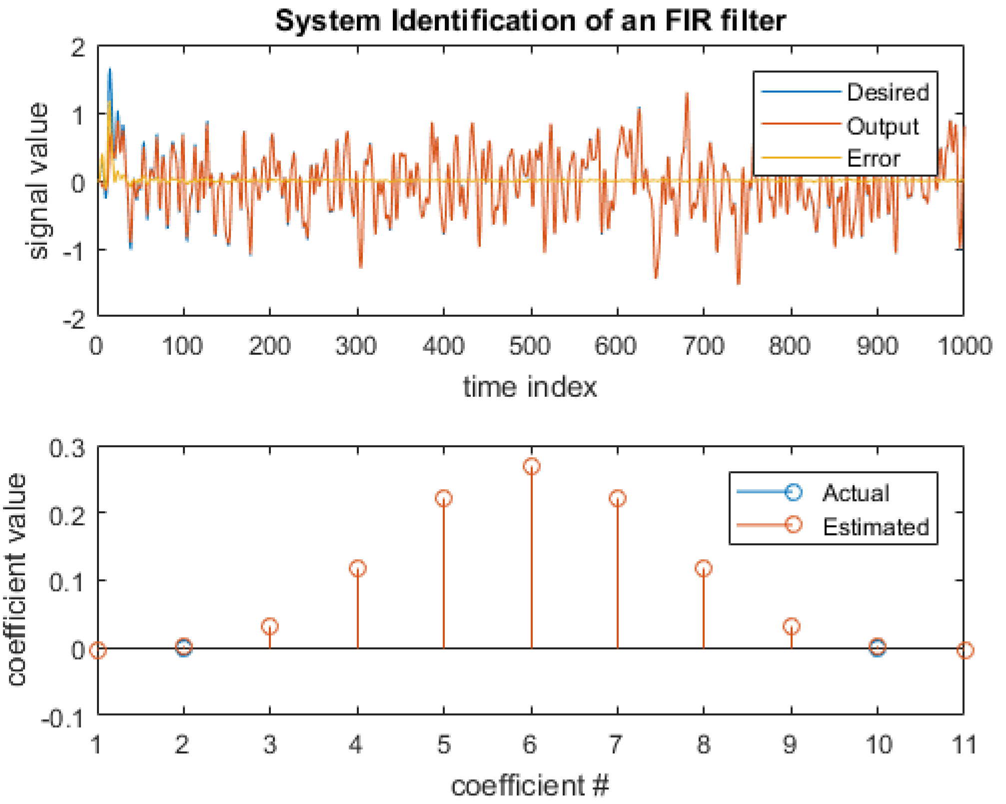
Examples

System Identification Using Fast Transversal Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
ftf1 = dsp.FastTransversalFilter(11,'ForgettingFactor',0.99);
filt = dsp.FIRFilter;
filt.Numerator = fir1(10,.25);
x = randn(1000,1);
d = filt(x) + 0.01*randn(1000,1);
[y,e] = ftf1(x,d);
w = ftf1.Coefficients;
subplot(2,1,1);
plot(1:1000,[d,y,e]);
title('System Identification of an FIR filter');
```

```
legend('Desired','Output','Error');
xlabel('time index');
ylabel('signal value');
subplot(2,1,2);
stem([filt.Numerator; w].');
legend('Actual','Estimated');
xlabel('coefficient #');
ylabel('coefficient value');
```



References

[1] Haykin, Simon. *Adaptive Filter Theory*, 4th Ed. Upper Saddle River, NJ: Prentice Hall, 2002

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.LMSFilter | dsp.AffineProjectionFilter | dsp.FIRFilter | dsp.RLSFilter | dsp.FrequencyDomainAdaptiveFilter | dsp.FilteredXLMSFilter

Introduced in R2013b

msesim

System object: dsp.FastTransversalFilter
Package: dsp

Mean-square error for Fast Transversal filter

Syntax

```
MSE = msesim(sysObj,X,D)
[MSE,MEANW,W,TRACEK] = msesim(sysObj,X,D)
[...] = msesim(sysObj,X,D,M)
```

Description

`MSE = msesim(sysObj,X,D)` returns a sequence of mean-square errors. This column vector contains estimates of the mean-square error of the adaptive filter at each time instant. The length of `MSE` is equal to `SIZE(X,1)`. The columns of the matrix `X` contain individual input signal sequences, and the columns of the matrix `D` contain corresponding desired response signal sequences.

`[MSE,MEANW,W,TRACEK] = msesim(sysObj,X,D)` calculates three parameters corresponding to the simulated behavior of the adaptive filter defined by `sysObj`. `MEANW` is a sequence of coefficient vector means. The columns of this matrix contain estimates of the mean values of the adaptive filter coefficients at each time instant. The dimensions of `MEANW` are `(SIZE(X,1))` by `(sysObj.length)`. `W` is an estimate of the final values of the adaptive filter coefficients for the algorithm corresponding to `sysObj`. `TRACEK` is a sequence of total coefficient error powers. This column vector contains estimates of the total coefficient error power of the adaptive filter at each time instant. The length of `TRACEK` is equal to `SIZE(X,1)`.

`[...] = msesim(sysObj,X,D,M)` specifies an optional decimation factor for computing `MSE`, `MEANW`, and `TRACEK`. If `M` > `1`, every M^{th} predicted value of each of these sequences is saved. If omitted, the value of `M` is the default, which is 1.

reset

System object: dsp.FastTransversalFilter
Package: dsp

Reset filter states for Fast Transversal filter

Syntax

`reset(sysObj)`

Description

`reset(sysObj)` resets the internal states of the System object, `sysObj`, to their initial values. The reset method is always a no-op for unlocked System objects, as the states may not be allocated when the object is not locked.

step

System object: dsp.FastTransversalFilter
Package: dsp

Apply Fast Transversal filter to input

Syntax

```
[Y, ERR] = step(sysObj, x, D)
Y = step(sysObj,x)
[Y1,...,YN] = step(sysObj,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[Y, ERR] = step(sysObj, x, D)` filters the input `x`, using D as the desired signal, and returns the filtered output in `Y` and the filter error in `ERR`. The System object estimates the filter weights needed to minimize the error between the output signal and the desired signal.

`Y = step(sysObj,x)` processes the input data, `x`, to produce the output, `Y`, from the System object, `sysObj`. `[Y1,...,YN] = step(sysObj,x)` produces `N` outputs.

Every System object has a step method. The step method processes the input data according to the object algorithm. The number of input and output arguments depends on the algorithm, and may depend also on one or more property settings. The step method for some objects accepts fixed-point (fi) inputs.

Calling step on an object puts that object into a locked state. When locked, you cannot change nontunable properties or any input characteristics (size, data type and complexity) without reinitializing (unlocking and relocking) the object.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.FFT System object

Package: dsp

Discrete Fourier transform

Description

The `FFT` object computes the discrete Fourier transform (DFT) of an input. The object uses one or more of the following fast Fourier transform (FFT) algorithms depending on the complexity of the input and whether the output is in linear or bit-reversed order:

- Double-signal algorithm
- Half-length algorithm
- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm
- An algorithm chosen by FFTW ,

To compute the DFT of an input:

1. Define and set up your `FFT` object. See "Construction" on page 4-680.
2. Call step to compute the DFT of the input according to the properties of `dsp.FFT`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`fft = dsp.FFT` returns a `FFT` object, `H`, that computes the DFT of an *N*-D array. For column vectors or multidimensional arrays, the `FFT` object computes the DFT along the first dimension. If the input is a row vector, the `FFT` object computes a row of single-sample DFTs and issues a warning.

`H = dsp.FFT('PropertyName',PropertyValue, ...)` returns a `FFT` object, `H`, with each property set to the specified value.

Properties

`FFTImplementation`

FFT implementation

Specify the implementation used for the FFT as one of `Auto | Radix-2 | FFTW`. When you set this property to `Radix-2`, the FFT length must be a power of two.

`BitReversedOutput`

Order of output elements relative to input elements

Designate order of output channel elements relative to order of input elements. Set this property to `true` to output the frequency indices in bit-reversed order. The default is `false`, which corresponds to a linear ordering of frequency indices.

`Normalize`

Divide butterfly outputs by two

Set this property to `true` if the output of the FFT should be divided by the FFT length. This option is useful when you want the output of the FFT to stay in the same amplitude range as its input. This is particularly useful when working with fixed-point data types.

The default value of this property is `false` with no scaling.

`FFTLengthSource`

Source of FFT length

Specify how to determine the FFT length as `Auto` or `Property`. When you set this property to `Auto`, the FFT length equals the number of rows of the input signal. The default is `Auto`.

`FFTLength`

FFT length

Specify the FFT length. This property applies when you set the `FFTLengthSource` on page 4-681 property to `Property`. The default is `64`.

This property must be a power of two when the input is a fixed-point data type, or when you set the `BitReversedOutput` on page 4-681 property to `true`, or when you set the `FFTImplementation` on page 4-681 property to `Radix-2`.

`WrapInput`

Boolean value of wrapping or truncating input

Wrap input data when FFT length is shorter than input length. If this property is set to true, modulo-length data wrapping occurs before the FFT operation, given FFT length is shorter than the input length. If this property is set to false, truncation of the input data to the FFT length occurs before the FFT operation. The default is `true`.

Fixed-Point Properties

`RoundingMethod`

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, `Zero`. The default is `Floor`.

`OverflowAction`

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

`SineTableDataType`

Sine table word and fraction lengths

Specify the sine table data type as `Same word length as input` or `Custom`. The default is `Same word length as input`.

`CustomSineTableDataType`

Sine table word and fraction lengths

Specify the sine table fixed-point type as an unscaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SineTableDataType` on page 4-682 property to `Custom`. The default is `numerictype([],16)`.

`ProductDataType`

Product word and fraction lengths

Specify the product data type as `Full precision`, `Same as input`, or `Custom`. The default is `Full precision`.

`CustomProductDataType`

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` on page 4-683 property to `Custom`. The default is `numerictype([],32,30)`.

`AccumulatorDataType`

Accumulator word and fraction lengths

Specify the accumulator data type as `Full precision`, `Same as input`, `Same as product`, or `Custom`. The default is `Full precision`.

`CustomAccumulatorDataType`

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` on page 4-683 property to `Custom`. The default is `numerictype([],32,30)`.

`OutputDataType`

Output word and fraction lengths

Specify the output data type as one of `Full precision`, `Same as input`, `Custom`. The default is `Full precision`.

`CustomOutputDataType`

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` on page 4-683 property to `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Discrete Fourier transform of input

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Single-Sided Amplitude Spectrum Of a Signal

Find frequency components of a signal in additive noise.

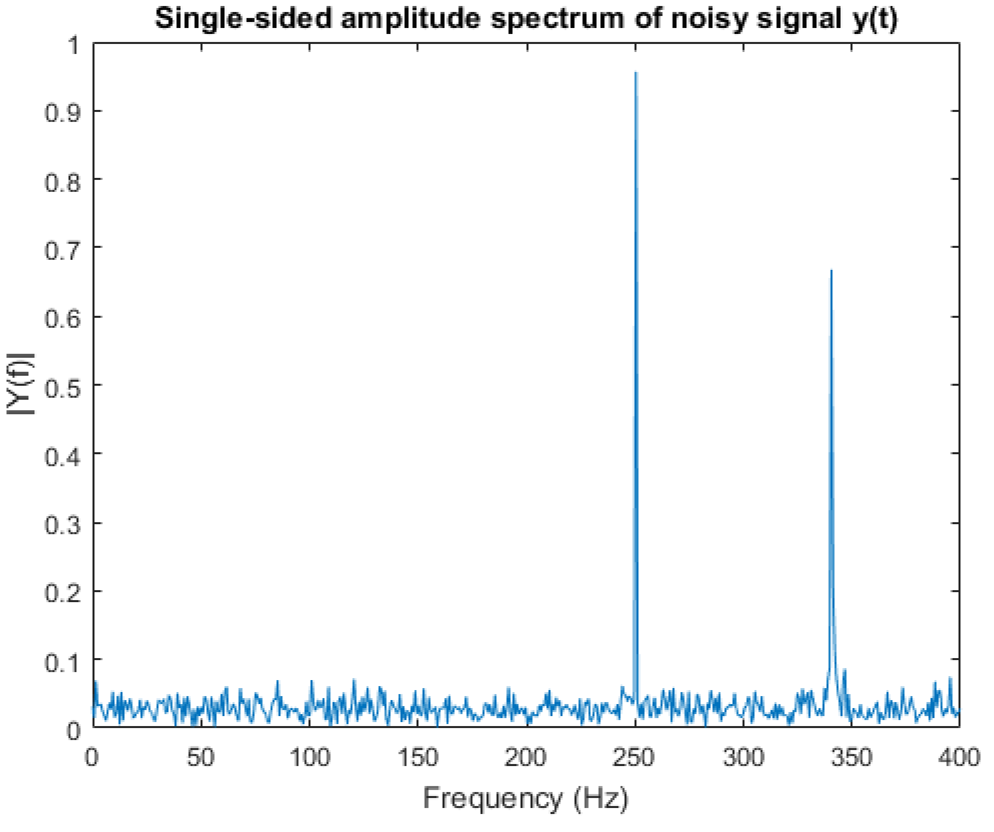
Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
Fs = 800; L = 1000;
t = (0:L-1)'/Fs;
x = sin(2*pi*250*t) + 0.75*cos(2*pi*340*t);
y = x + .5*randn(size(x)); % noisy signal
ft = dsp.FFT('FFTLengthSource', 'Property', ...
    'FFTLength', 1024);
Y = ft(y);
```

Plot the single-sided amplitude spectrum

```
plot(Fs/2*linspace(0,1,512), 2*abs(Y(1:512)/1024));
```

```
title('Single-sided amplitude spectrum of noisy signal y(t)');
xlabel('Frequency (Hz)'); ylabel('|Y(f)|');
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the FFT block reference page. The object properties correspond to the block parameters.

References

[1] FFTW (http://www.fftw.org)

[2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT,"*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- "System Objects in MATLAB Code Generation" (MATLAB Coder).
- When the following conditions apply, the executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB:
  - `FFTImplementation` is set to `'FFTW'`.
  - `FFTImplementation` is set to `'Auto'`, `FFTLengthSource` is set to `'Property'`, and `FFTLength` is not a power of two.

  Use the `packNGo` function to package the code generated from this System object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "How To Run a Generated Executable Outside MATLAB".
- When the FFT length is a power of two, you can generate standalone C and C++ code from this System object.

See Also

See Also

System Objects
dsp.DCT | dsp.IDCT | dsp.IFFT

Introduced in R2012a

step

System object: dsp.FFT
Package: dsp

Discrete Fourier transform of input

Syntax

`Y = step(ft,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(ft,X)` computes the FFT, `Y` , of the input `X` along the first dimension of `X`. When the `FFTLengthSource` property is `Auto`, the length of `X` along the first dimension must be a positive integer power of two. This length is also the FFT length . When the `FFTLengthSource` property is `Property`, the `FFTLength` property must be a positive integer power of two.

dsp.FilterCascade System object

Package: dsp

Create a cascade of filter System objects

Description

The `dsp.FilterCascade` object creates a multistage System object that enables cascading of filter System objects, delays, and scalar gains. To see the System objects you can assign, at the MATLAB command prompt, enter:

```
dsp.FilterCascade.helpSupportedSystemObjects
```

You can use the cascaded filter for filtering a signal, and for generating the filtering code. You can pass the `dsp.FilterCascade` System object as a stage to another `dsp.FilterCascade` System object. You can also pass `dsp.FilterCascade` System object as an input to the `cascade` function. This object supports variable-size signals if the stages within support variable-size signals.

To filter a signal using the cascaded filter:

1 Define and set up your `dsp.FilterCascade` System object. See “Construction” on page 4-689.
2 Call step to filter each channel of the input signal using the stages in the cascade. The size, data type, and complexity of the input signal must be supported by the stages in the filter cascade.
3 Run the dsp.FilterCascade.generateFilteringCode method to generate MATLAB code using the design specified through the System objects.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`filtCasc = dsp.FilterCascade` returns a System object, `filtCasc`. This System object has a single stage of the dsp.FIRFilter object.

`filtCasc = dsp.FilterCascade(`*`Filter_1`*`,`*`Filter_2`*`,...)` returns a multistage System object, `filtCasc`, with the first stage set to *`Filter_1`*, the second stage set to *`Filter_2`*, and so on. *`Filter_1`*, *`Filter_2`*, etc. can be a filter System object, a `dsp.FilterCascade` System object, delay, or a scalar gain.

`filtCasc = cascade(`*`Filter_1`*`,`*`Filter_2`*`,...)` returns a `dsp.FilterCascade` System object, `filtCasc`, with the first stage set to *`Filter_1`*, the second stage set to *`Filter_2`*, and so on. *`Filter_1`*, *`Filter_2`*, etc. can be a filter System object, a `dsp.FilterCascade` System object, or a scalar gain. `cascade` function does not support delay as a filter stage.

Properties

`Stage`*`i`* `(Stage1, Stage2,...)`

Specify the *`i`*th stage of the `dsp.FilterCascade` System object. You can assign any of the filter System objects, delay, or a scalar gain to this property. To see the System objects you can assign, at the MATLAB command prompt, enter:

```
dsp.FilterCascade.helpSupportedSystemObjects
```

The default value of this property is `dsp.FIRFilter`. For more information, see dsp.FIRFilter.

Methods

addStage	Add a new filter stage to the cascade
generateFilteringCode	Generate MATLAB code to filter using cascade
getNumStages	Get the number of stages in the cascade object

releaseStages	Release the locked state of all stages in the cascade
removeStage	Remove a stage from filter cascade
step	Process inputs using the filter cascade

More "Analysis Methods for Filter System Objects" on page 3-2.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Filter a Signal Using Cascaded Lowpass and Highpass Filters

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, `myObject(x)` becomes `step(myObject,x)`.

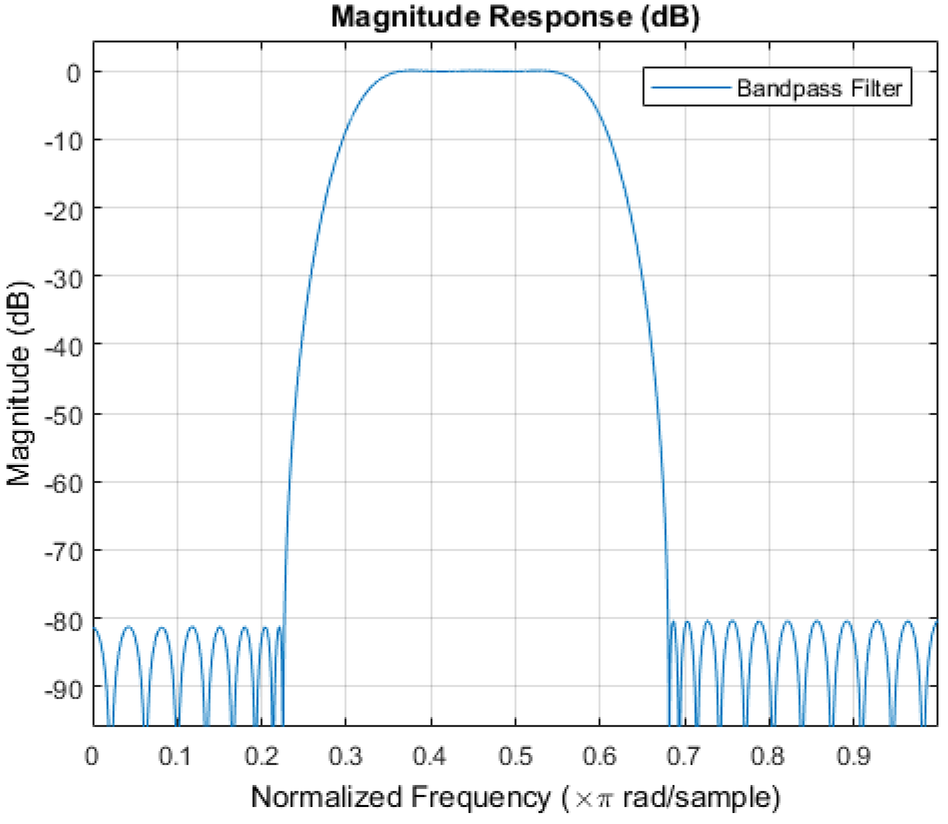
Design a bandpass filter by cascading:

- A highpass filter with a stopband frequency of 5000 Hz and a passband frequency of 8000 Hz.
- A lowpass filter with a passband frequency of 12,000 Hz and a stopband frequency of 15,000 Hz.

Visualize the frequency response using `fvtool`.

```
lpFilt = dsp.LowpassFilter('StopbandFrequency',15000,...
    'PassbandFrequency',12000);
hpFilt = dsp.HighpassFilter('StopbandFrequency',5000,...
    'PassbandFrequency',8000);

bpFilt = dsp.FilterCascade(lpFilt,hpFilt);
```

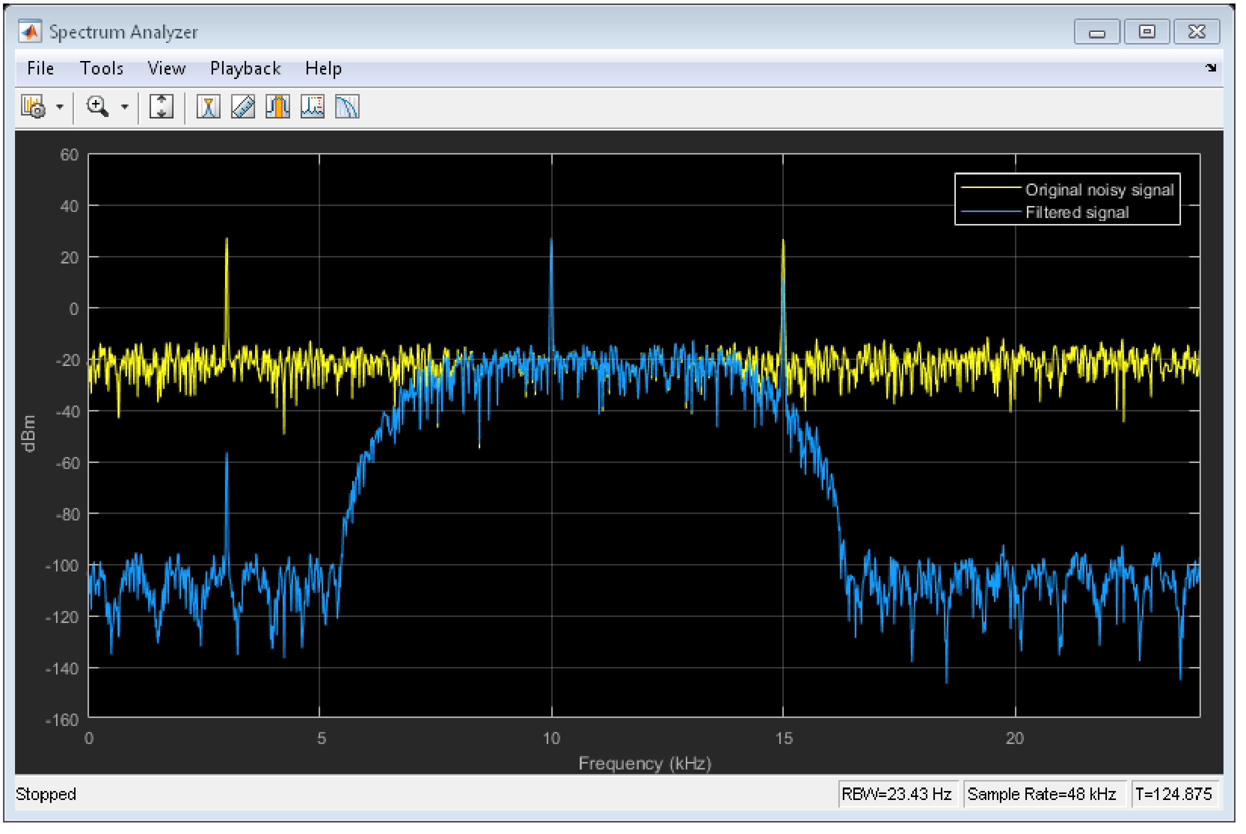
```
fvtool(bpFilt);
legend('Bandpass Filter');
```



Pass a noisy sine wave as the input to the bandpass filter. The input is a sum of three sine waves with frequencies at 3 kHz, 10 kHz, and 15 kHz. The sampling frequency is 48 kHz. View the input and the filtered output on a spectrum analyzer.

```
Sine1 = dsp.SineWave('Frequency',3e3,'SampleRate',48e3,'SamplesPerFrame',6000);
Sine2 = dsp.SineWave('Frequency',10e3,'SampleRate',48e3,'SamplesPerFrame',6000);
Sine3 = dsp.SineWave('Frequency',15e3,'SampleRate',48e3,'SamplesPerFrame',6000);

SpecAna = dsp.SpectrumAnalyzer('PlotAsTwoSidedSpectrum',false, ...
    'SampleRate',Sine1.SampleRate, ...
    'NumInputPorts',2,...
```

```
    'ShowLegend',true, ...
    'YLimits',[-160,60]);

SpecAna.ChannelNames = {'Original noisy signal','Filtered signal'};


for i = 1 : 1000
    x = Sine1()+Sine2()+Sine3()+0.1.*randn(Sine1.SamplesPerFrame,1);
    y = bpFilt(x);
    SpecAna(x,y);

end
release(SpecAna)
```



The tones at 3 kHz and 15 kHz are attenuated, and the tone at 10 kHz is preserved by the bandpass filter.

Design a Compensation Decimator for a CIC Decimator

Create a CIC decimator. Cascade the decimator with a gain.

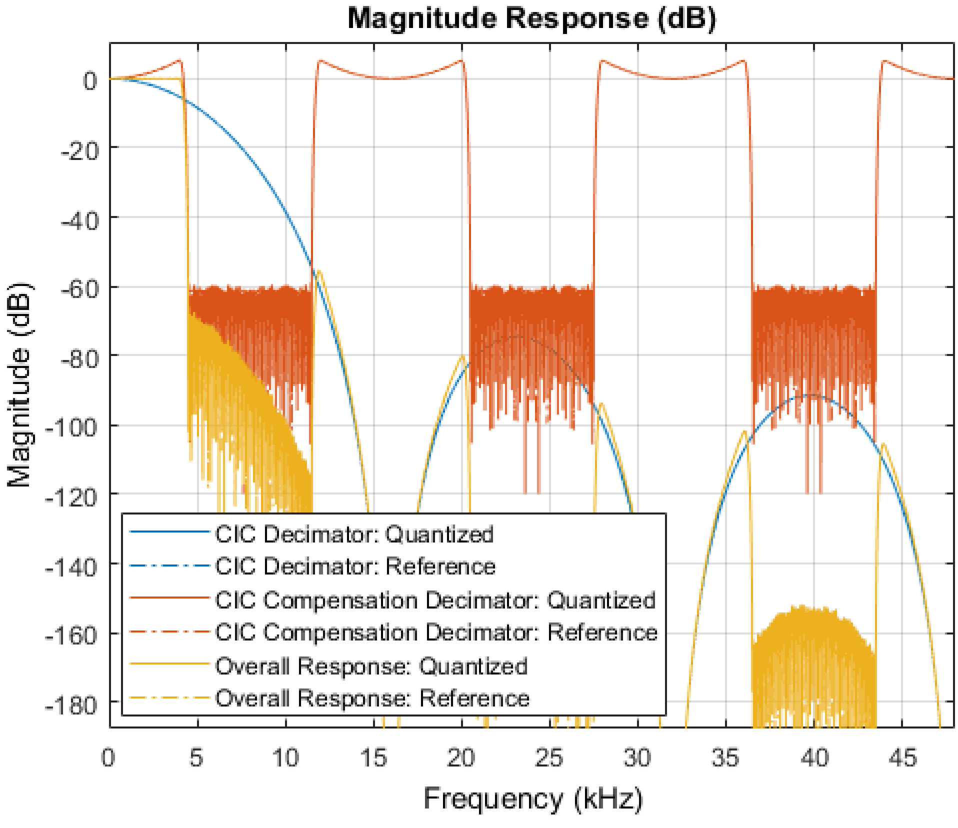
```
cicdecim = dsp.CICDecimator('DecimationFactor', 6, ...
    'NumSections', 6);
decimcasc = dsp.FilterCascade(cicdecim, 1/gain(cicdecim));
```

Design a compensation decimator and cascade it with the filter cascade, `decimcasc`.

```
fs = 16e3;     % Sampling frequency of input of compensation decimator
fPass = 4e3;   % Passband frequency
fStop = 4.5e3; % Stopband frequency
ciccomp = dsp.CICCompensationDecimator(cicdecim, ...
    'DecimationFactor', 2, ...
    'PassbandFrequency', fPass, ...
    'StopbandFrequency', fStop, ...
    'SampleRate', fs);
filtchain = dsp.FilterCascade(decimcasc, ciccomp);
```

Visualize the frequency response of the cascade of cascades.

```
f = fvtool(decimcasc, ciccomp, filtchain, 'Fs',[fs*6,fs,fs*6],...
    'Arithmetic', 'fixed');
legend(f,'CIC Decimator','CIC Compensation Decimator', ...
    'Overall Response');
```



The CIC compensation decimator has an inherent gain, `gain(cicdecim)`. By cascading with a factor of `1/gain(cicdecim)`, the filter cascade compensates for this gain.

Generate Code to Filter Using Cascade

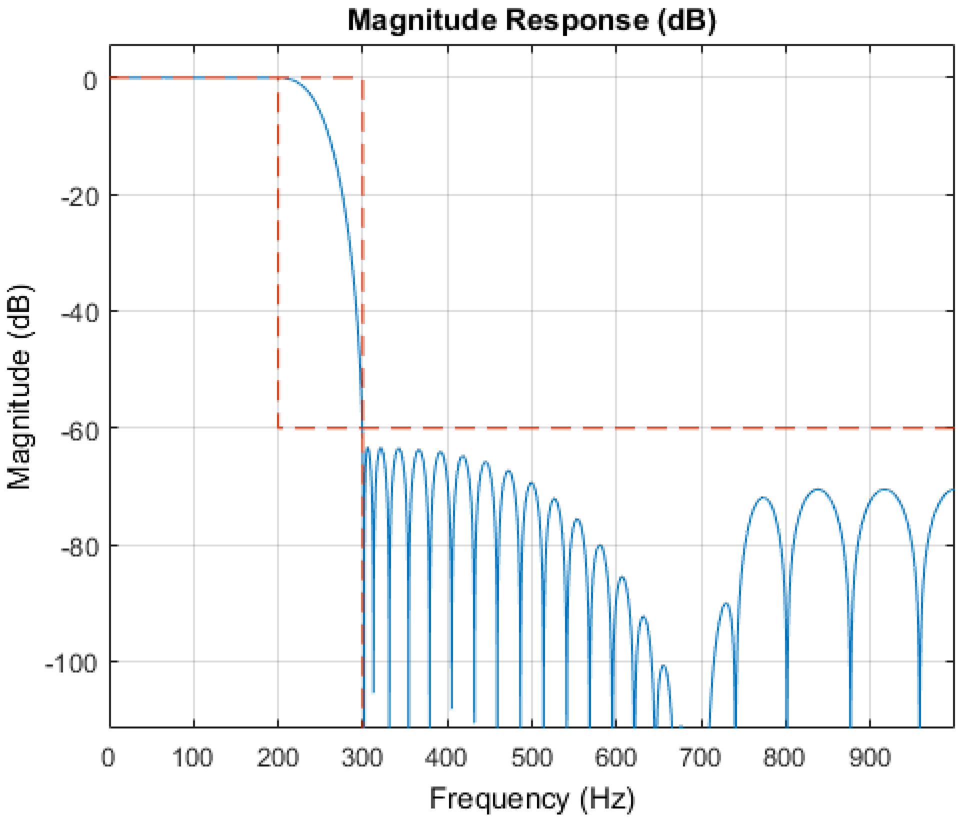
Design a two-stage decimator with a 100 Hz transition width, a 2 kHz sampling frequency, and 60 dB attenuation in the stopband. The decimator needs to downsample by a factor of 4.

```
decimSpec = fdesign.decimator(4, 'Nyquist', 4, 'Tw,Ast', 100,60,2000);
filtCasc = design(decimSpec, 'multistage', 'SystemObject', true);
```

Verify your design using `fvtool`.

```
info(filtCasc)
fvtool(filtCasc)
```

```
ans =

    'Discrete-Time Filter Cascade
     ----------------------------
     Number of stages: 2

     Stage1: dsp.FIRDecimator
     -------
     Discrete-Time FIR Multirate Filter (real)
     -----------------------------------------
     Filter Structure   : Direct-Form FIR Polyphase Decimator
     Decimation Factor  : 2
     Polyphase Length   : 10
     Filter Length      : 19
     Stable             : Yes
     Linear Phase       : Yes (Type 1)

     Arithmetic         : double


     Stage2: dsp.FIRDecimator
     -------
     Discrete-Time FIR Multirate Filter (real)
     -----------------------------------------
     Filter Structure   : Direct-Form FIR Polyphase Decimator
     Decimation Factor  : 2
     Polyphase Length   : 18
     Filter Length      : 35
     Stable             : Yes
     Linear Phase       : Yes (Type 1)

     Arithmetic         : double

     '
```



Generate code to filter data using this design. The code is saved in a file called `stepDecimator.m` in the current directory.

```
generateFilteringCode(filtCasc, 'stepDecimator');
```

The `stepDecimator` function creates a filter cascade and calls the `step` method on each stage.

```
type stepDecimator
```

```
function y = stepDecimator(x)
%STEPDECIMATOR Construct filter cascade and process each stage
```

```
% MATLAB Code
% Generated by MATLAB(R) 9.2 and the DSP System Toolbox 9.4.
% Generated on: 24-Feb-2017 14:58:21

% To generate C/C++ code from this function use the codegen command.
% Type 'help codegen' for more information.
%#codegen

%% Construction
persistent filter1 filter2
if isempty(filter1)
    filter1 = dsp.FIRDecimator(  ...
        'Numerator', [0.0021878514650437845 0 -0.010189095418136306 0 0.031140395225498
    filter2 = dsp.FIRDecimator(  ...
        'Numerator', [0.001155501175048853 0 -0.0027482166351234854 0 0.005768198228952
end

%% Process
y1 = filter1( x );
y  = filter2( y1);
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You cannot generate code directly from `dsp.FilterCascade`. You can use the `generateFilteringCode` method to generate a MATLAB function. You can generate C/C++ code from this MATLAB function, if the filters in each stage support code generation.
- See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

System Objects
dsp.CICCompensationDecimator | dsp.CICDecimator | dsp.FIRFilter

Introduced in R2014b

addStage

System object: dsp.FilterCascade
Package: dsp

Add a new filter stage to the cascade

Syntax

```
addStage(FC, FILTERSYSOBJ)
addStage(FC, FILTERSYSOBJ, STAGEID)
```

Description

`addStage(FC, FILTERSYSOBJ)` adds `FILTERSYSOBJ` at the end of the filter cascade stored in the `dsp.FilterCascade` object `FC`. To see the System objects you can assign to `FILTERSYSOBJ`, at the MATLAB command prompt, enter

```
dsp.FilterCascade.helpSupportedSystemObjects
```

`addStage(FC, FILTERSYSOBJ, STAGEID)` adds `FILTERSYSOBJ` at stage position `STAGEID` of the filter cascade stored in the `dsp.FilterCascade` object `FC`. All existing filters from `STAGEID` to end of the cascade are shifted down in the cascade.

`FILTERSYSOBJ` can be any of the filter System objects provided by the DSP System Toolbox.

Examples

Add Stage to Cascade End

Add a stage at the end of the filter cascade:

```
FC = dsp.FilterCascade;
addStage(FC, dsp.IIRFilter);
```

Add Stage inside Cascade

Add a filter at the second stage of a filter cascade consisting of three filters:

```
FC = dsp.FilterCascade(dsp.CICDecimator,dsp.FIRDecimator,dsp.FIRFilter);
addStage(FC, dsp.IIRFilter, 2);
```

generateFilteringCode

System object: dsp.FilterCascade
Package: dsp

Generate MATLAB code to filter using cascade

Syntax

```
generateFilteringCode(FC)
generateFilteringCode(FC, FNAME)
```

Description

`generateFilteringCode(FC)` creates a MATLAB function in the editor that contains code to create stages of the cascade and calls `step` method on each stage in sequence.

`generateFilteringCode(FC, FNAME)` generates code and saves it to the file specified in the string `FNAME`.

Examples

Create a filter cascade object and generate code from it:

```
FC = dsp.FilterCascade(dsp.FIRFilter('Numerator',ones(1,5)),...
                           dsp.FIRDecimator);
fName = 'filterCascade.m';  % The code will be saved in this
                                % file in the current directory
generateFilteringCode(FC, fName);
```

getNumStages

System object: dsp.FilterCascade
Package: dsp

Get the number of stages in the cascade object

Syntax

`getNumStages(FC)`

Description

`getNumStages(FC)` returns the number of stages in the `dsp.FilterCascade` object, `FC`.

Examples

Get the number of stages in a filter cascade object:

```
FC = cascade(dsp.FIRFilter, dsp.IIRFilter);
N = getNumStages(FC);    % This will return 2
```

releaseStages

System object: dsp.FilterCascade
Package: dsp

Release the locked state of all stages in the cascade

Syntax

`releaseStages(FC)`

Description

`releaseStages(FC)` calls the `release` method of individual stages in the `dsp.FilterCascade` object `FC`.

For instance, if a `dsp.FilterCascade` object consists of a `dsp.FIRFilter` object and a `dsp.FIRInterpolator` object, the `releaseStages` method calls the:

- `release` method of the dsp.FIRFilter object.
- `release` method of the dsp.FIRInterpolator object.

Examples

Release Stages of a Filter Cascade

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create and release stages of a filter cascade.

```
firfilt = dsp.FIRFilter;
y = firfilt(randn);
FC = dsp.FilterCascade(dsp.FIRInterpolator, firfilt);
isLocked(FC.Stage2) % Returns 1
```

```
releaseStages(FC);
isLocked(FC.Stage2) % Returns 0
```

```
ans =

  logical

   1


ans =

  logical

   0
```

removeStage

System object: dsp.FilterCascade
Package: dsp

Remove a stage from filter cascade

Syntax

```
removeStage(FC)
removeStage(FC, STAGEID)
```

Description

`removeStage(FC)` removes a stage from the end of the filter cascade stored in the `dsp.FilterCascade` object `FC`.

`removeStage(FC, STAGEID)` removes the stage from the stage position `STAGEID` of the filter cascade stored in the `dsp.FilterCascade` object `FC`. All existing filters from `STAGEID` to end of the cascade are shifted up in the cascade.

Examples

Remove Stage from Cascade End

Remove a stage from the end of a filter cascade:

```
FC = dsp.FilterCascade(dsp.FIRFilter, dsp.IIRFilter);
removeStage(FC);
```

Remove Stage from Inside Cascade

Remove a filter from the third stage of a cascade consisting of four stages:

```
FC = cascade(dsp.FIRInterpolator, dsp.FIRInterpolator, ...
                dsp.FIRDecimator, dsp.FIRDecimator);
```

```
removeStage(FC, 3);
```

step

System object: dsp.FilterCascade
Package: dsp

Process inputs using the filter cascade

Syntax

```
y = step(filtCasc,x)
```

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(filtCasc,x)` runs the input `x` through the filter cascade, `filtCasc`, and returns the filtered output, `y`. The size, data type, and complexity of the input signal must be supported by the stages in the filter cascade.

Introduced in R2017a

dsp.FilteredXLMSFilter System object

Package: dsp

Filtered XLMS filter

Description

The `dsp.FilteredXLMSFilter` computes output, error and coefficients using Filtered-X Least Mean Squares FIR adaptive filter.

To implement the adaptive FIR filter object:

1. Define and set up your adaptive FIR filter object. See “Construction” on page 4-708.
2. Call step to implement the filter according to the properties of `dsp.FilteredXLMSFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`fxlms = dsp.FilteredXLMSFilter` returns a filtered-x Least Mean Square FIR adaptive filter System object, `fxlms`. This System object is used to compute the filtered output and the filter error for a given input and desired signal.

`fxlms = dsp.FilteredXLMSFilter('`*`PropertyName`*`',` *`PropertyValue`*`,...)` returns a `FilteredXLMSFilter` System object, `fxlms`, with each specified property set to the specified value.

`fxlms= dsp.FilteredXLMSFilter(LEN,'`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a `FilteredXLMSFilter` System object, `fxlms`, with the Length property set to

LEN, and other specified properties set to the specified values. For the algorithm on how to implement this filter, refer to [1], [2].

Properties

Length

Length of filter coefficients vector

Specify the length of the FIR filter coefficients vector as a positive integer value. This property is nontunable.

The default value is 10.

StepSize

Adaptation step size

Specify the adaptation step size factor as a positive numeric scalar. The default value is 0.1. This property is tunable.

LeakageFactor

Adaptation leakage factor

Specify the leakage factor used in a leaky adaptive filter as a numeric value between 0 and 1, both inclusive. When the value is less than 1, the System object implements a leaky adaptive algorithm. The default value is 1, providing no leakage in the adapting method. This property is tunable.

SecondaryPathCoefficients

Coefficients of the secondary path filter model

Specify the coefficients of the secondary path filter model as a numeric vector. The secondary path connects the output actuator and the error sensor. The default value is a vector that represents the coefficients of a 10th-order FIR lowpass filter. This property is tunable.

SecondaryPathEstimate

An estimate of the secondary path filter model

Specify the estimate of the secondary path filter model as a numeric vector. The secondary path connects the output actuator and the error sensor. The default value equals to the SecondayPathCoefficients property value. This property is not tunable.

InitialCoefficients

Initial coefficients of the filter

Specify the initial values of the FIR adaptive filter coefficients as a scalar or a vector of length equal to the value of the Length property. The default value is 0.

This property is tunable.

LockCoefficients

Locked status of the coefficient updates

Specify whether to lock the filter coefficient values. By default, the value of this property is false, and the object continuously updates the filter coefficients. If this property is set to true, the filter coefficients do not update and their values remain the same.

This property is tunable.

Methods

msesim	Mean-square error for Filtered-X LMS filter
reset	Reset filter states for Filtered-X LMS filter
step	Apply Filtered-X LMS filter to input

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Active Noise Control of a Random Noise Signal

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Generate noise, create FIR primary path system model, generate observation noise, filter the primary path system model output with added noise, and create FIR secondary path system model.

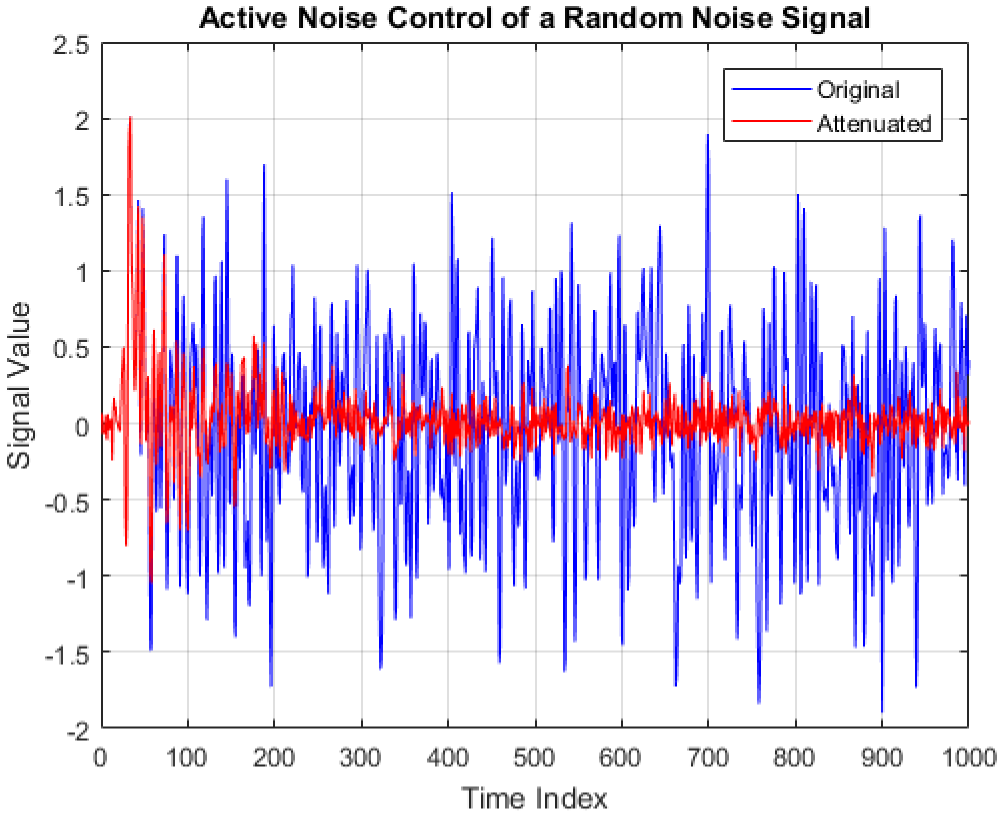
```
x  = randn(1000,1);
g  = fir1(47,0.4);
n  = 0.1*randn(1000,1);
d  = filter(g,1,x) + n;
b  = fir1(31,0.5);
```

Use the Filtered-X LMS Filter to compute the filtered output and the filter error for the input and the signal to be cancelled.

```
mu = 0.008;
fxlms = dsp.FilteredXLMSFilter(32, 'StepSize', mu, 'LeakageFactor', ...
     1, 'SecondaryPathCoefficients', b);
[y,e] = fxlms(x,d);
```

Plot the results.

```
plot(1:1000,d,'b',1:1000,e,'r');
title('Active Noise Control of a Random Noise Signal');
legend('Original','Attenuated');
xlabel('Time Index'); ylabel('Signal Value');  grid on;
```



References

[1] Kuo, S.M. and Morgan, D.R. *Active Noise Control Systems: Algorithms and DSP Implementations*. New York: John Wiley & Sons, 1996.

[2] Widrow, B. and Stearns, S.D. *Adaptive Signal Processing*. Upper Saddle River, N.J: Prentice Hall, 1985.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.LMSFilter | dsp.AffineProjectionFilter | dsp.FrequencyDomainAdaptiveFilter | dsp.RLSFilter | dsp.AdaptiveLatticeFilter | dsp.FIRFilter

Introduced in R2013b

msesim

System object: dsp.FilteredXLMSFilter
Package: dsp

Mean-square error for Filtered-X LMS filter

Syntax

```
MSE = msesim(sysObj,X,D)
[MSE,MEANW,W,TRACEK] = msesim(sysObj,X,D)
[...] = msesim(sysObj,X,D,M)
```

Description

`MSE = msesim(sysObj,X,D)` returns a sequence of mean-square errors. This column vector contains estimates of the mean-square error of the adaptive filter at each time instant. The length of `MSE` is equal to `SIZE(X,1)`. The columns of the matrix `X` contain individual input signal sequences, and the columns of the matrix `D` contain corresponding desired response signal sequences.

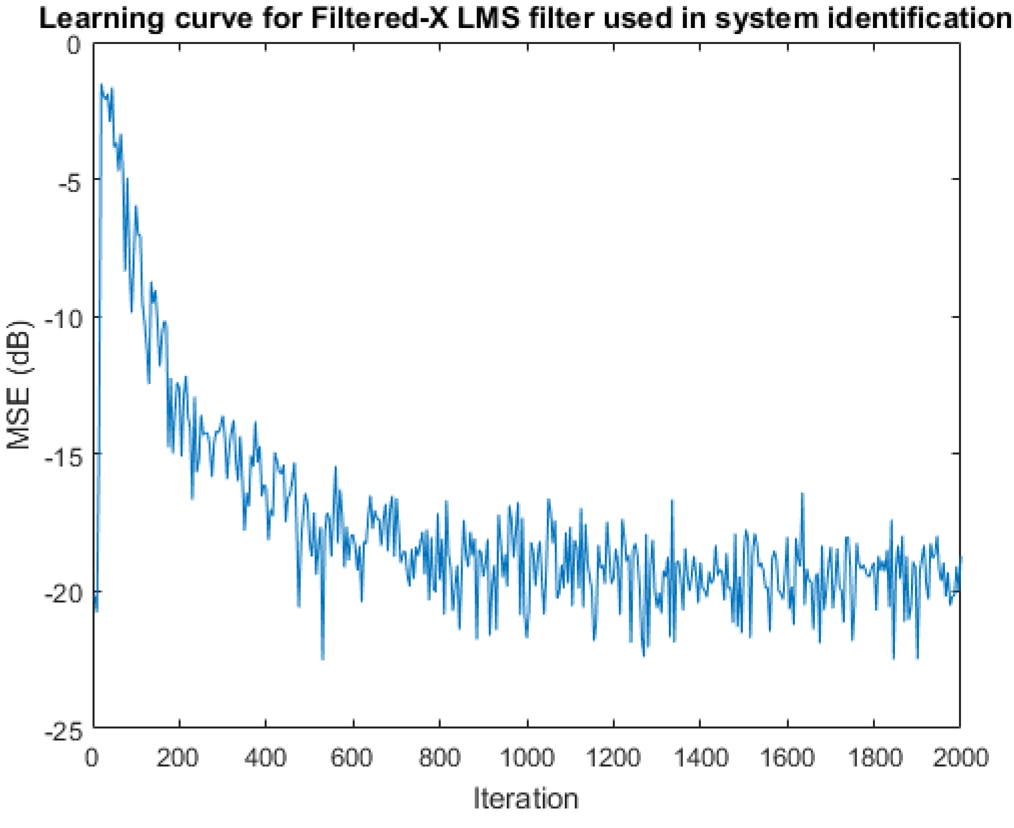
`[MSE,MEANW,W,TRACEK] = msesim(sysObj,X,D)` calculates three parameters corresponding to the simulated behavior of the adaptive filter defined by `sysObj`. `MEANW` is a sequence of coefficient vector means. The columns of this matrix contain estimates of the mean values of the adaptive filter coefficients at each time instant. The dimensions of `MEANW` are `(SIZE(X,1))` by `(sysObj.length)`. `W` is an estimate of the final values of the adaptive filter coefficients for the algorithm corresponding to `sysObj`. `TRACEK` is a sequence of total coefficient error powers. This column vector contains estimates of the total coefficient error power of the adaptive filter at each time instant. The length of `TRACEK` is equal to `SIZE(X,1)`.

`[...] = msesim(sysObj,X,D,M)` specifies an optional decimation factor for computing `MSE`, `MEANW`, and `TRACEK`. If `M > 1`, every M^{th} predicted value of each of these sequences is saved. If omitted, the value of `M` is the default, which is 1.

System Identification of an FIR Filter Using Filtered XLMS Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
ha = fir1(31,0.5);
fir = dsp.FIRFilter('Numerator',ha); % FIR system to be identified
iir = dsp.IIRFilter('Numerator',sqrt(0.75),...
        'Denominator',[1 -0.5]);
x = iir(sign(randn(2000,25)));
n = 0.1*randn(size(x));            % Observation noise signal
d = fir(x)+n;                   % Desired signal
l = 32;                            % Filter length
mu = 0.008;                        % Filtered-X LMS filter Step size.
m  = 5;                            % Decimation factor for analysis
                                   % and simulation results
fxlms = dsp.FilteredXLMSFilter(l,'StepSize',mu);
[simmse,meanWsim,Wsim,traceKsim] = msesim(fxlms,x,d,m);
plot(m*(1:length(simmse)),10*log10(simmse));
xlabel('Iteration'); ylabel('MSE (dB)');
title('Learning curve for Filtered-X LMS filter used in system identification')
```


Learning curve for Filtered-X LMS filter used in system identification
MSE (dB)
0
-5
-10
-15
-20
-25
0
200
400
600
800
1000
1200
1400
1600
1800
2000
Iteration

reset

System object: dsp.FilteredXLMSFilter
Package: dsp

Reset filter states for Filtered-X LMS filter

Syntax

`reset(sysObj)`

Description

`reset(sysObj)` resets the internal states of the System object, `sysObj`, to their initial values. The reset method is always a no-op for unlocked System objects, as the states may not be allocated when the object is not locked.

step

System object: dsp.FilteredXLMSFilter
Package: dsp

Apply Filtered-X LMS filter to input

Syntax

```
[Y, ERR] = step(sysObj, x, D)
Y = step(sysObj,x)
[Y1,...,YN] = step(sysObj,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[Y, ERR] = step(sysObj, x, D)` filters the input `x`, using D as the desired signal, and returns the filtered output in `Y` and the filter error in `ERR`. The System object estimates the filter weights needed to minimize the error between the output signal and the desired signal.

`Y = step(sysObj,x)` processes the input data, `x`, to produce the output, `Y`, from the System object, `sysObj`. `[Y1,...,YN] = step(sysObj,x)` produces `N` outputs.

Every System object has a step method. The step method processes the input data according to the object algorithm. The number of input and output arguments depends on the algorithm, and may depend also on one or more property settings. The step method for some objects accepts fixed-point (fi) inputs.

Calling step on an object puts that object into a locked state. When locked, you cannot change nontunable properties or any input characteristics (size, data type and complexity) without reinitializing (unlocking and relocking) the object.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.FIRDecimator System object

Package: dsp

Polyphase FIR decimator

Description

The `FIRDecimator` object resamples vector or matrix inputs along the first dimension. The object reseamples at a rate *M* times slower than the input sampling rate, where *M* is the integer-valued downsampling factor. The decimation combines an FIR anti-aliasing filter with downsampling. The FIR decimator object uses a polyphase implementation of the FIR filter.

To resample vector or matrix inputs along the first dimension:

1 Define and set up your FIR decimator. See “Construction” on page 4-720.
2 Call step to resample the vector or matrix inputs according to the properties of `dsp.FIRDecimator`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`firdecim = dsp.FIRDecimator` returns an FIR decimator, `firdecim`, which applies an FIR filter with a cutoff frequency of `0.4*pi` radians/sample to the input and downsamples the filter output by factor of 2. This System object supports variable-size input.

`firdecim = dsp.FIRDecimator ('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns an FIR decimator, `firdecim`, with each property set to the specified value.

`firdecim = dsp.FIRDecimator(DECIM, NUM, '`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns an FIR decimator, `firdecim`, with the integer-valued DecimationFactor property set to DECIM, the Numerator property set to NUM, and other specified properties set to the specified values.

Properties

DecimationFactor

Decimation factor

Specify the downsampling factor as a positive integer. The FIR decimator reduces the sampling rate of the input by this factor. The size of the input along the first dimension must be a multiple of the decimation factor. The default is 2.

NumeratorSource

FIR filter coefficient source

Specify the source of the numerator coefficients as one of 'Property' (default) or 'Input port'. When you specify 'Input port', the filter object requires the numerator coefficients to be specified as the third argument to every step.

Numerator

FIR filter coefficients

Specify the numerator coefficients of the FIR filter in powers of z^{-1}. The following equation defines the system function for a filter of length L:

$$H(z) = \sum_{l=0}^{L-1} b_l z^{-l}$$

To prevent aliasing as a result of downsampling, the filter transfer function should have a normalized cutoff frequency no greater than 1/DecimationFactor. You can specify the filter coefficients as a vector in the supported data types. The default is fir1(35,0.4).

Structure

Filter structure

Specify the implementation of the FIR filter as either `Direct form` or `Direct form transposed`. The default is `Direct form`.

Fixed-Point Properties

`FullPrecisionOverride`

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

`RoundingMethod`

Rounding method for fixed-point operations

Specify the rounding method as one of `| Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |`. The default is `Floor`. This property applies only if the object is not in full precision mode.

`OverflowAction`

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` or `Saturate`. This property applies only if the object is not in full precision mode.

`CoefficientsDataType`

Coefficients word and fraction lengths

Specify the coefficients fixed-point data type as `Same word length as input` or `Custom`. The default is `Same word length as input`.

`CustomCoefficientsDataType`

Coefficients word and fraction lengths

Specify the coefficients fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `CoefficientsDataType` on page 4-722 property to `Custom`. The default is `numerictype([],16,15)`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of | `Full precision` | `Same as input` | `Custom` |. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` on page 4-723 property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of | `Full precision` | `Same as product` | `Same as input` | `Custom` |. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` on page 4-723 property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of | `Same as accumulator` | `Same as product` | `Same as input` | `Custom` |. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` on page 4-723 property to `Custom`. The default is `numerictype([]1,16,15)`.

Methods

freqz	Frequency response
fvtool	Open filter visualization tool
impz	Impulse response
phasez	Unwrapped phase response
reset	Reset filter states of FIR decimator
step	Decimate input by integer factor

More "Analysis Methods for Filter System Objects" on page 3-2.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Decimate a Sum of Sine Waves

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

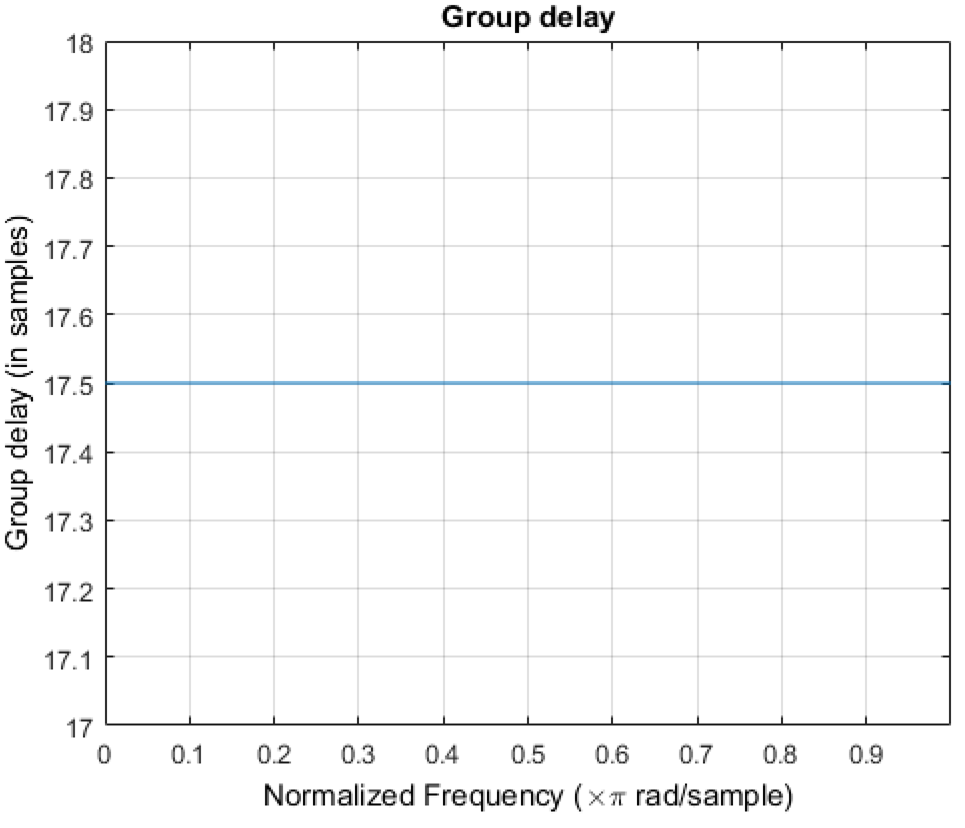
This example shows how to decimate a sum of sine waves with angular frequencies of pi/4 and 2pi/3 radians/sample by a factor of two. To prevent aliasing, the FIR decimator filters out the 2pi/3 radians/sample component before downsampling.

```
x = cos(pi/4*[0:95]')+sin(2*pi/3*[0:95]');
```

```
firdecim = dsp.FIRDecimator;
y = firdecim(x);
```

View group delay of default FIR filter

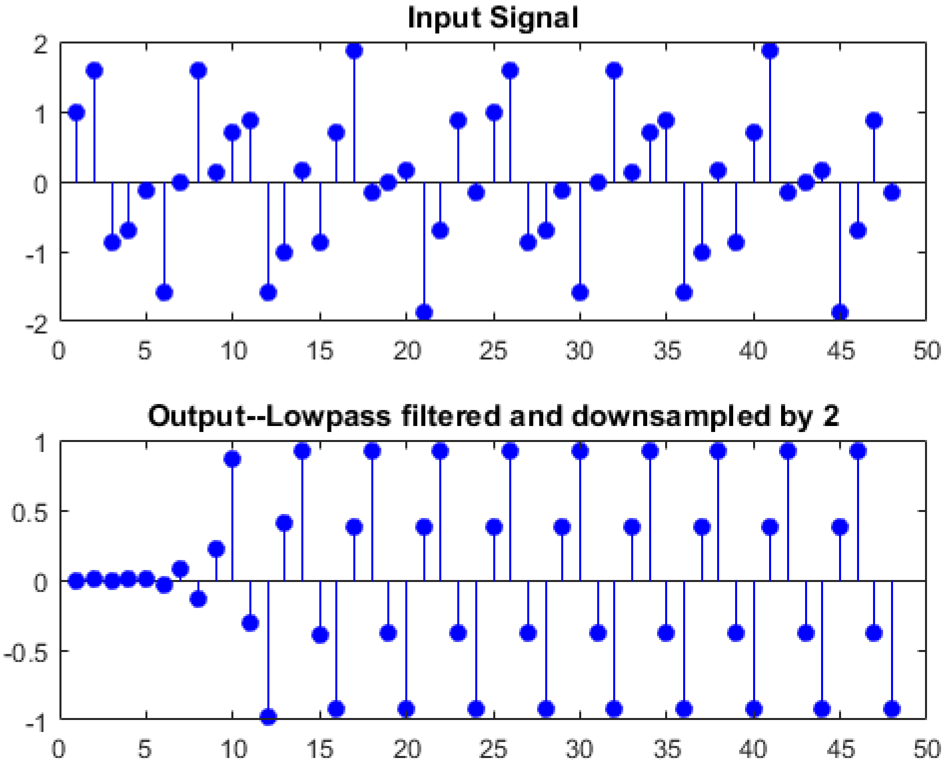
```
fvtool(fir1(35,0.4),1,'analysis','grpdelay');
```



Group delay of the default linear-phase FIR filter is 17.5 samples. Downsampling by a factor of two expect an approx. 8.75 sample delay in the output y with the initial filter states of zero

```
subplot(211);
stem(x(1:length(x)/2),'b','markerfacecolor',[0 0 1]);
title('Input Signal');
```

```
subplot(212);
stem(y,'b','markerfacecolor',[0 0 1]);
title('Output--Lowpass filtered and downsampled by 2');
```



Reduce the Sample Rate of an Audio Signal

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

This example shows how to reduce the sampling rate of an audio signal by 1/2 and plays it.

```
afr = dsp.AudioFileReader('OutputDataType',...
```

```
  'single');
adw = audioDeviceWriter(22050/2);
firdecim = dsp.FIRDecimator;

while ~isDone(afr)
     frame = afr();
     y = firdecim(frame);
     adw(y);
end

release(afr);
pause(0.5);
release(adw);
```

Definitions

Polyphase Subfilters

A polyphase implementation of an FIR decimator *splits* the lowpass FIR filter impulse response into M different subfilters, where M is the downsampling, or decimation factor. Let $h(n)$ denote the FIR filter impulse response of length L and $u(n)$ the input signal. Decimating the filter output by a factor of M is equivalent to the downsampled convolution:

$$y(n) = \sum_{l=0}^{L-1} h(l)u(nM - l)$$

The key to the efficiency of polyphase filtering is that specific input values are only multiplied by select values of the impulse response in the downsampled convolution. For example, letting $M=2$, the input values $u(0),u(2),u(4),$ ... are only combined with the filter coefficients $h(0),h(2),h(4),$..., and the input values $u(1),u(3),u(5),$ ... are only combined with the filter coefficients $h(1),h(3),h(5),$.... By splitting the filter coefficients into two polyphase subfilters, no unnecessary computations are performed in the convolution. The outputs of the convolutions with the polyphase subfilters are interleaved and summed to yield the filter output. The following MATLAB code demonstrates how to construct the two polyphase subfilters for the default order 35 filter in the `Numerator` on page 4-721 property and the default `DecimationFactor` on page 4-721 property value of two:

```
M = 2;
```

```
Num = fir1(35,0.4);
FiltLength = length(Num);
Num = flipud(Num(:));

if (rem(FiltLength, M) ~= 0)
     nzeros = M - rem(FiltLength, M);
     Num = [zeros(nzeros,1); Num];  % Appending zeros
end

len = length(Num);
nrows = len / M;
PolyphaseFilt = flipud(reshape(Num, M, nrows).');
```

The columns of `PolyphaseFilt` are subfilters containing the two *phases* of the filter in `Num`. For a general downsampling factor of M , there are M phases and therefore M subfilters.

Algorithms

This object implements the algorithm, inputs, and outputs described on the `FIR Decimation` block reference page. The object properties correspond to the block parameters, except:

- **Framing** – The FIR decimator object only supports `Maintain input frame rate`
- **Output buffer initial conditions** – The FIR decimator object does not support this parameter.
- **Rate options** – The FIR decimator object does not support this parameter.
- **Input processing** The FIR decimator object does not support this parameter.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.FIRRateConverter | dsp.FIRInterpolator

Introduced in R2012a

freqz

System object: dsp.FIRDecimator
Package: dsp

Frequency response

Syntax

```
[h,w] = freqz(firdecim)
[h,w] = freqz(firdecim,n)
[h,w] = freqz(firdecim,Name,Value)
freqz(firdecim)
```

Description

`[h,w] = freqz(firdecim)` returns the complex, 8192–element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample.

`[h,w] = freqz(firdecim,n)` returns the complex, n-element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[h,w] = freqz(firdecim,Name,Value)` returns the frequency response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`freqz(firdecim)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the filter System object `firdecim`.

fvtool

System object: dsp.FIRDecimator
Package: dsp

Open filter visualization tool

Syntax

```
fvtool(firdecim)
fvtool(firdecim,'Arithmetic',ARITH,...)
```

Description

`fvtool(firdecim)` performs an analysis and computes the magnitude response of the filter System object `firdecim`.

`fvtool(firdecim,'Arithmetic',ARITH,...)` analyzes the filter System object `firdecim`, based on the arithmetic specified in the ARITH input. ARITH can be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The 'Arithmetic' input is only relevant for the analysis of filter System objects.

impz

System object: dsp.FIRDecimator
Package: dsp

Impulse response

Syntax

```
[h,t] = impz(firdecim)
[h,t] = impz(firdecim,Name,Value)
impz(firdecim)
```

Description

`[h,t] = impz(firdecim)` returns the impulse response `h`, and the corresponding time points `t` at which the impulse response of`firdecim` is computed.

`[h,t] = impz(firdecim,Name,Value)` returns the impulse response `h`, and the corresponding time points`t` , with additional options specified by one or more Name, Value pair arguments.

`impz(firdecim)` uses FVTool to plot the impulse response of the filter System object `firdecim`.

Note: You can use `impz` for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

---

phasez

System object: dsp.FIRDecimator
Package: dsp

Unwrapped phase response

Syntax

```
[phi,w] = phasez(firdecim)
[phi,w] = phasez(firdecim,n)
[phi,w] = phasez(firdecim,Name,Value)
phasez(firdecim)
```

Description

`[phi,w] = phasez(firdecim)` returns the 8192–element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample.

`[phi,w] = phasez(firdecim,n)` returns the n-element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[phi,w] = phasez(firdecim,Name,Value)` returns the phase response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`phasez(firdecim)` uses FVTool to plot the phase response of the filter System object `firdecim`.

reset

System object: dsp.FIRDecimator
Package: dsp

Reset filter states of FIR decimator

Syntax

```
reset(firdecim)
```

Description

`reset(firdecim)` resets the filter states of the FIR decimator object,`firdecim`, to the initial values of zero. After the `step` method applies the FIR decimator to nonzero input data, the filter states may change. Invoking the `step` method again without first invoking the `reset` method may produce different outputs for an identical input.

Reset an FIR Decimator

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
firdecim = dsp.FIRDecimator;
x = cos(pi/8*(0:1023)')+sin(pi/4*(0:1023)');
y = firdecim(x);
```

Call the FIR decimator again without reset.

```
y1 = firdecim(x);
isequal(y,y1)
```

```
ans =

  logical

   0
```

Reset filter states.

```
reset(firdecim);
```

Call the FIR decimator again.

```
y2 = firdecim(x);
isequal(y,y2)
```

```
ans =

  logical

   1
```

step

System object: dsp.FIRDecimator
Package: dsp

Decimate input by integer factor

Syntax

```
Y = step(firdecim,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(firdecim,X)` outputs the filtered and downsampled values, `Y`, of the input signal, `X`. A Ki-by-N input matrix is treated as N independent channels. The length of each column must be a multiple of the `DecimationFactor` property value. The FIR decimator operates on each channel separately and generates a Ko-by-N output matrix where $Ko = Ki/M$ with M the decimation factor. The object supports real and complex floating-point and fixed-point inputs, except for complex unsigned fixed-point inputs.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.FIRFilter System object

Package: dsp

Static or time-varying FIR filter

Description

The `FIRFilter` object filters each channel of the input using static or time-varying FIR filter implementations.

To filter each channel of the input:

1. Define and set up your FIR filter. See "Construction" on page 4-737.
2. Call step to filter each channel of the input according to the properties of `dsp.FIRFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`fir = dsp.FIRFilter` returns a default FIR filter object, `fir`, which independently filters each channel of the input over successive calls to the `step` method, using a specified FIR filter implementation. This System object supports variable-size input.

`fir = dsp.FIRFilter('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns an FIR filter system object, `fir`, with each property set to the specified value.

Properties

`Structure`

Filter structure

Specify the filter structure.
You can specify the filter structure as one of | `Direct form` | `Direct form symmetric` | `Direct form antisymmetric` | `Direct form transposed` | `Lattice MA` |. The default is `Direct form`.

`NumeratorSource`

Source of filter coefficients

Specify the source of the filter coefficients as one of | `Property` | `Input port` |. The default is `Property`. When you specify `Input port`, the filter object updates the time-varying filter once every frame.

This applies when you set the `Structure` to `Direct form` | `Direct form symmetric` | `Direct form antisymmetric` | `Direct form transposed`.

`ReflectionCoefficientsSource`

Source of filter coefficients

Specify the source of the Lattice filter coefficients as one of | `Property` | `Input port` |. The default is `Property`. When you specify `Input port`, the filter object updates the time-varying filter once every frame.

This applies when you set the `Structure` to `Lattice MA` .

`Numerator`

Numerator coefficients

Specify the filter coefficients as a real or complex numeric row vector. This property applies when you set the `NumeratorSource` on page 4-738 property to `Property`, and the `Structure` on page 4-737 property is set to `Direct form`, `Direct form symmetric`, `Direct form antisymmetric`, or `Direct form transposed`. The default is `[0.5 0.5]`. This property is tunable.

`ReflectionCoefficients`

Reflection coefficients of lattice filter structure

Specify the reflection coefficients of a lattice filter as a real or complex numeric row vector. This property applies when you set the `Structure` on page 4-737 property to

`Lattice MA`, and the `ReflectionCoefficientsSource` on page 4-738 property to `Property`. The default is `[0.5 0.5]`. This property is tunable.

InitialConditions

Initial conditions for the FIR filter

Specify the initial conditions of the filter states. The number of states or delay elements equals the number of reflection coefficients for the lattice structure, or the number of filter coefficients–1 for the other direct form structures.

You can specify the initial conditions as a scalar, vector, or matrix. If you specify a scalar value, the FIR filter object initializes all delay elements in the filter to that value. If you specify a vector whose length equals the number of delay elements in the filter, each vector element specifies a unique initial condition for the corresponding delay element. The object applies the same vector of initial conditions to each channel of the input signal.

If you specify a vector whose length equals the product of the number of input channels and the number of delay elements in the filter, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel.

If you specify a matrix with the same number of rows as the number of delay elements in the filter, and one column for each channel of the input signal, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel. The default is 0.

This property is tunable.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see "Full Precision for Fixed-Point System Objects".

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero` |. The default is `Floor`. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | `Wrap` | `Saturate` |. The default is `Wrap`. This property applies only if the object is not in full precision mode.

CoefficientsDataType

Coefficients word and fraction lengths

Specify the coefficients fixed-point data type as one of | `Same word length as input` | `Custom` |. This property applies when you set the `NumeratorSource` on page 4-738 property to `Property`. The default is `Same word length as input`.

CustomCoefficientsDataType

Custom coefficients word and fraction lengths

Specify the coefficients fixed-point type as a signed or unsigned `numerictype` object. This property applies when you set the `CoefficientsDataType` on page 4-740 property to `Custom`. The default is `numerictype(true,16,15)`.

ReflectionCoefficientsDataType

Reflection coefficients word and fraction lengths

Specify the reflection coefficients fixed-point data type as one of | `Same word length as input` | `Custom` |. This property applies when you set the `ReflectionCoefficientsSource` on page 4-738 property to `Property`. The default is `Same word length as input`.

CustomReflectionCoefficientsDataType

Custom reflection coefficients word and fraction lengths

Specify the reflection coefficients fixed-point type as a signed or unsigned `numerictype` object. This property applies when you set the `ReflectionCoefficientsDataType` on page 4-740 property to `Custom`. The default is `numerictype(true,16,15)`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of | `Full precision` | `Same as input` | `Custom` |. The default is `Full precision`.

CustomProductDataType

Custom product word and fraction lengths

Specify the product fixed-point type as a signed or unsigned scaled `numerictype` object. This property applies when you set the `ProductDataType` on page 4-741 property to `Custom`. The default is `numerictype(true,32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type to one of | `Full precision` | `Same as input` | `Same as product` | `Custom` |. The default is `Full precision`.

CustomAccumulatorDataType

Custom accumulator word and fraction lengths

Specify the accumulator fixed-point type as a signed or unsigned scaled `numerictype` object. This property applies when you set the `AccumulatorDataType` on page 4-741 property to `Custom`. The default is `numerictype(true,32,30)`.

StateDataType

State word and fraction lengths

Specify the state fixed-point data type as one of | Same as input | Same as accumulator | Custom |. This property does not apply to any of the direct form or direct form I filter structures. The default is Same as accumulator.

CustomStateDataType

Custom state word and fraction lengths

Specify the state fixed-point type as a signed or unsigned scaled numerictype object. This property applies when you set the StateDataType on page 4-741 property to Custom. The default is numerictype(true,16,15).

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of | Same as input | Same as accumulator | Custom |. The default is Same as accumulator.

CustomOutputDataType

Custom output word and fraction lengths

Specify the output fixed-point type as a signed or unsigned scaled numerictype object. This property applies when you set the OutputDataType on page 4-742 property to Custom. The default is numerictype(true,16,15).

Methods

freqz	Frequency response
fvtool	Open filter visualization tool
impz	Impulse response
phasez	Unwrapped phase response
reset	Reset internal states of FIR filter
step	Filter input with FIR filter object

More “Analysis Methods for Filter System Objects” on page 3-2.

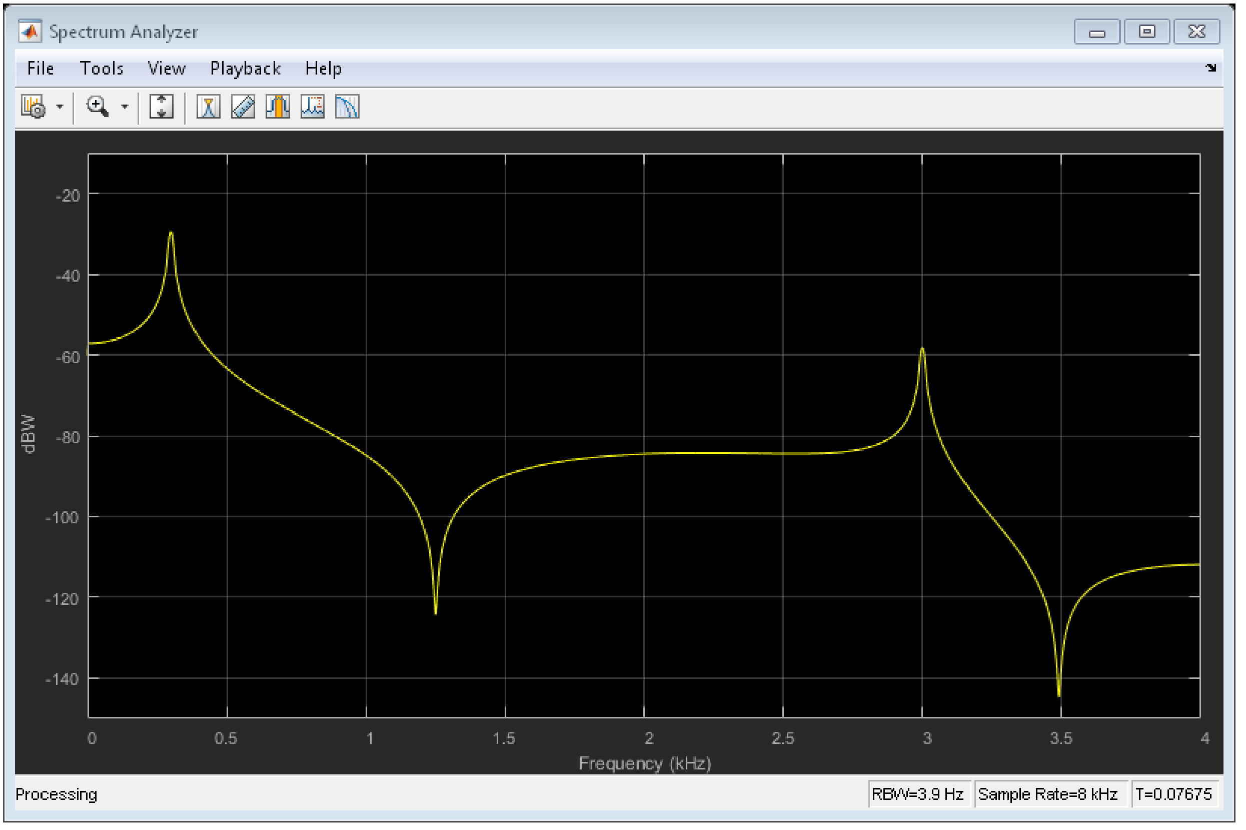
Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

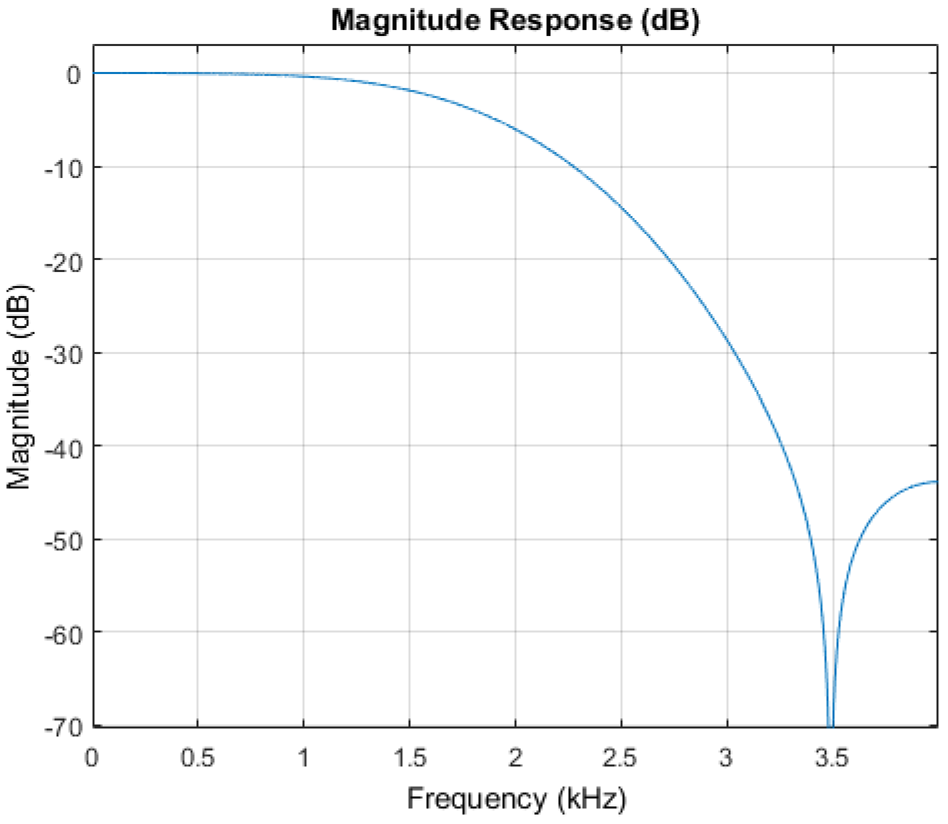
Examples

Lowpass Filter a Sinusoid Signal Using FIRFilter object

Use an FIR filter to apply a low pass filter to a waveform with two sinusoidal components.

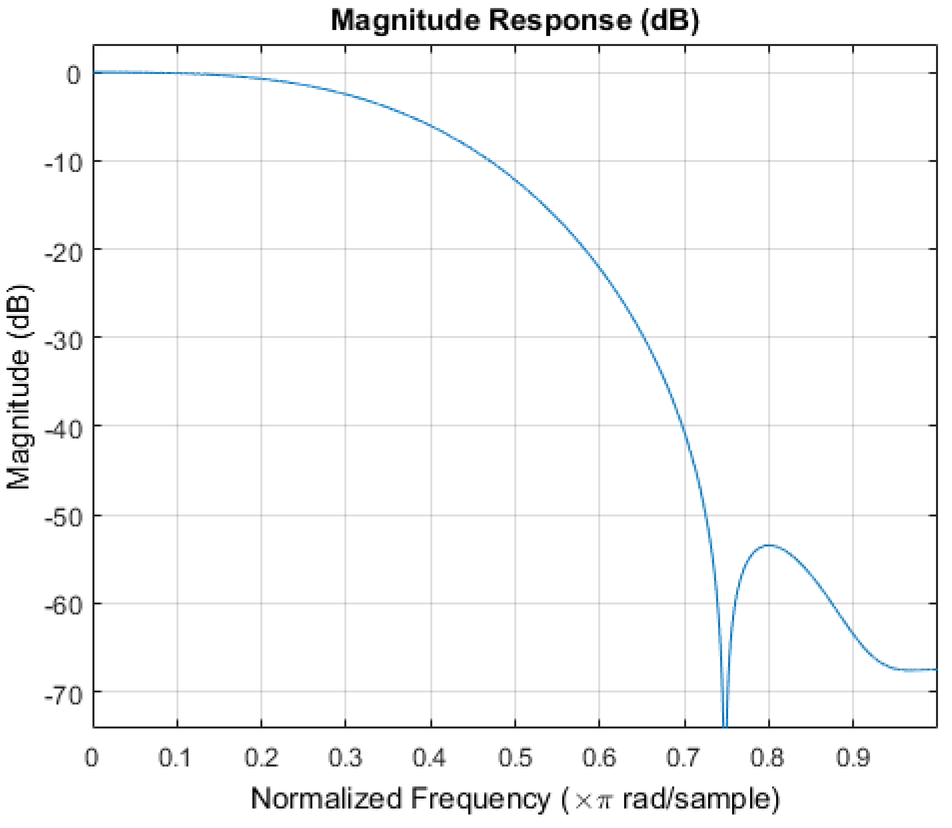
```
 t = (0:1000)'/8e3;
 xin = sin(2*pi*0.3e3*t)+sin(2*pi*3e3*t);

 sr = dsp.SignalSource;
 sr.Signal = xin;
 sink = dsp.SignalSink;

 fir = dsp.FIRFilter;
 fir.Numerator = fir1(10,0.5);

sa = dsp.SpectrumAnalyzer('SampleRate',8e3,...
    'PlotAsTwoSidedSpectrum',false,...
    'OverlapPercent', 80, 'PowerUnits','dBW',...
    'YLimits', [-150 -10]);

 while ~isDone(sr)
      input = sr();
      filteredOutput = fir(input);
      sink(filteredOutput);
      sa(filteredOutput)
 end

 filteredResult = sink.Buffer;
 fvtool(fir,'Fs',8000)
```


Spectrum Analyzer
File Tools View Playback Help
-20
-40
-60
-80
-100
-120
-140
dBW
0
0.5
1
1.5
2
2.5
3
3.5
4
Frequency (kHz)
Processing
RBW=3.9 Hz
Sample Rate=8 kHz
T=0.07675



Design an FIR filter as a System object.

```
N = 10;
Fc = 0.4;
B = fir1(N,Fc);
fir1 = dsp.FIRFilter('Numerator',B);
fvtool(fir1)
```

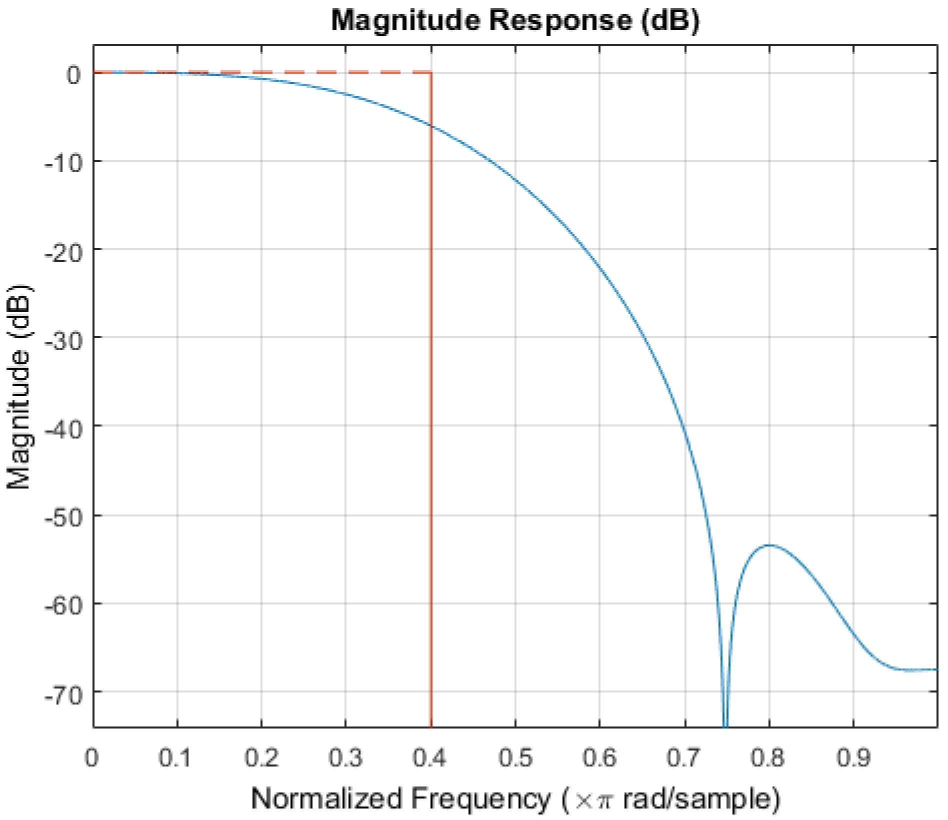


This can also be achieved by using `fdesign` as a constructor and `design` to design the filter.

```
N = 10;
Fc = 0.4;
specLowpass = fdesign.lowpass('N,Fc',N,Fc);
fir2 = design(specLowpass,'systemobject',true)
fvtool(fir2);
```

```
fir2 =

  dsp.FIRFilter with properties:
```

```
          Structure: 'Direct form'
    NumeratorSource: 'Property'
          Numerator: [1×11 double]
  InitialConditions: 0

Use get to show all properties
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Discrete FIR Filter` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `Numerator` property is tunable for code generation.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

“FIR Nyquist (L-th band) Filter Design” | dsp.BiquadFilter | `Discrete FIR Filter`

Introduced in R2012a

freqz

System object: dsp.FIRFilter
Package: dsp

Frequency response

Syntax

```
[h,w] = freqz(fir)
[h,w] = freqz(fir,n)
[h,w] = freqz(fir,Name,Value)
freqz(fir)
```

Description

`[h,w] = freqz(fir)` returns the complex, 8192–element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample.

`[h,w] = freqz(fir,n)` returns the complex, n-element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[h,w] = freqz(fir,Name,Value)` returns the frequency response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`freqz(fir)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the filter System object `fir`.

fvtool

System object: dsp.FIRFilter
Package: dsp

Open filter visualization tool

Syntax

```
fvtool(fir)
fvtool(fir,'Arithmetic',ARITH,...)
```

Description

`fvtool(fir)` performs an analysis and computes the magnitude response of the filter System object `fir`.

`fvtool(fir,'Arithmetic',ARITH,...)` analyzes the filter System object `fir`, based on the arithmetic specified in the ARITH input. ARITH can be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The 'Arithmetic' input is only relevant for the analysis of filter System objects.

impz

System object: dsp.FIRFilter
Package: dsp

Impulse response

Syntax

```
[h,t] = impz(fir)
[h,t] = impz(fir,Name,Value)
impz(fir)
```

Description

`[h,t] = impz(fir)` returns the impulse response `h`, and the corresponding time points `t` at which the impulse response of`fir` is computed.

`[h,t] = impz(fir,Name,Value)` returns the impulse response `h`, and the corresponding time points`t` , with additional options specified by one or more Name, Value pair arguments.

`impz(fir)` uses FVTool to plot the impulse response of the filter System object `fir`.

Note: You can use `impz` for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

phasez

System object: dsp.FIRFilter
Package: dsp

Unwrapped phase response

Syntax

```
[phi,w] = phasez(fir)
[phi,w] = phasez(fir,n)
[phi,w] = phasez(fir,Name,Value)
phasez(fir)
```

Description

`[phi,w] = phasez(fir)` returns the 8192–element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample.

`[phi,w] = phasez(fir,n)` returns the n-element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[phi,w] = phasez(fir,Name,Value)` returns the phase response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`phasez(fir)` uses FVTool to plot the phase response of the filter System object `fir`.

reset

System object: dsp.FIRFilter
Package: dsp

Reset internal states of FIR filter

Syntax

`reset(fir)`

Description

`reset(fir)` resets the filter states of the FIR filter object,`fir`, to their initial values of 0. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the FIR filter object to nonzero input data, the states may be nonzero. Invoking the `step` method again without first invoking the`reset` method may produce different outputs for an identical input.

Reset an FIR Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
fir = dsp.FIRFilter;
fir.Numerator = fir1(20,0.25);
n = 0:100;
x = cos(0.2*pi*n)+sin(0.8*pi*n);
y = fir(x);
```

Filter states are nonzero. Invoke step method again without resetting states

```
y1 = fir(x);
isequal(y,y1)
```

```
ans =
```

```
  logical

   0
```

Now reset filter states to 0

```
reset(fir)
```

Invoke step method

```
y2 = fir(x);
isequal(y,y2)
```

```
ans =

  logical

   1
```

step

System object: dsp.FIRFilter
Package: dsp

Filter input with FIR filter object

Syntax

```
Y = step(fir,X)
Y = step(fir,X,COEFF)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(fir,X)` filters the real or complex input signal `X` using the FIR filter, `fir`, to produce the output `Y` . When the input data is of a fixed-point type, it must be signed when the structure is set to `Direcform symmetric` or `Direct form antisymmetric`. The FIR filter object operates on each channel of the input signal independently over successive calls to step method.

`Y = step(fir,X,COEFF)` uses the time-varying coefficients, `COEFF` , to filter the input signal `X` and produce the output `Y` . You can use this option when you set the `NumeratorSource` or `ReflectionCoefficientsSource` property to `Input port`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change

nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.HDLFIRFilter System object

Package: dsp

Finite impulse response filter—optimized for HDL code generation

Description

The `HDLFIRFilter` System object models FIR filter architectures optimized for HDL code generation. The object is sample based, accepting one scalar at a time. It provides a hardware-friendly interface with optional input and output flow control signals.

The object implements a direct-form systolic FIR filter architecture based on multiply-accumulate operations with pipeline registers. The choice of architecture makes the object well-suited for high-throughput HDL code generation targeted for FPGAs with dedicated DSP blocks. The generated HDL code runs at a high clock rate and can filter new input data on every cycle.

The object provides flexible and efficient resource sharing options. This property results in tradeoffs between throughput and resource utilization. A sharing factor of *N* indicates that the object reduces overall DSP resource utilization by a factor of *N*. In this case, the object can process only input samples that are at least *N* time steps apart. To determine when the object is ready for new input data, use the optional `readyPort` property.

To provide a cycle-accurate simulation of the generated HDL code, the object models pipeline registers and resource sharing. These operations cause a delay between valid input data and the corresponding valid output data. The actual latency depends on the number of coefficients and the sharing factor.

To process input data with an HDL-optimized FIR filter:

1. Create and configure a `dsp.HDLFIRFilter` System object. See “Construction” on page 4-759.
2. Call the step method to filter the input data according to the properties of the `dsp.HDLFIRFilter` object.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as

dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`firFilt = dsp.HDLFIRFilter` creates an HDL-optimized discrete FIR filter System object, `firFilt`, with default filter properties.

`firFilt = dsp.HDLFIRFilter(num)` creates the filter with the Numerator property set to `num`.

`firFilt = dsp.HDLFIRFilter(___,Name,Value)` creates the filter with optional `Name,Value` pair arguments, in addition to any of the input arguments in previous syntaxes. Name is a property name from the list of "Properties" on page 4-759 and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

Main

`Numerator` — Discrete FIR filter coefficients

`[0.5 0.5]` (default) | vector of numeric values

Discrete FIR filter coefficients, specified as a vector of numeric values. You can also specify the vector as a workspace variable, or as a call to a filter design function. Complex coefficients are not supported. When the input data type is floating point, the object casts the coefficients to the same data type as the input. When the input data type is integer or fixed point, you can modify coefficient type casting details using the `CoefficientsDataType` property.

Example: `'Numerator',firpm(30,[0 .1 .2 .5]*2,[1 1 0 0])` defines the coefficients using a linear-phase filter design function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`Sharing` — Resource sharing option
`false` (default) | `true`

Resource sharing option, specified as a logical value. To reduce the overall DSP resource utilization on the FPGA, set this property to `true`. Then set the `SharingFactor` property to the value by which you want to reduce the number of DSPs.

`SharingFactor` — Factor by which the number of DSPs is reduced
`2` (default) | positive integer

Factor by which the number of DSPs is reduced, specified as a positive integer. A sharing factor of *N* indicates that the object reduces DSP resource utilization by a factor of *N*. In this case, the object can process only input samples that are at least *N* time steps apart.

Dependencies

To enable this property, set the `Sharing` property to `true`.

Data Types

`RoundingMethod` — Rounding method for type casting
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding method for type casting, specified as `'Floor'`, `'Ceiling'`, `'Convergent'`, `'Nearest'`, `'Round'`, or `'Zero'`. The rounding method is used when casting the output to the data type specified by the `OutputDataType` property. When the input data type is floating point, the `RoundingMethod` property has no effect. See “Rounding Modes” for more details.

`OverflowAction` — Overflow handling for type casting
`'Wrap'` (default) | `'Saturate'`

Overflow handling for type casting, specified as `'Wrap'` or `'Saturate'`. Overflow handling is used when casting the output to the data type specified by the `OutputDataType` property. When the input data type is floating point, the `OverflowAction` property has no effect. See "Overflow Handling" for more details.

`CoefficientsDataType` — Data type of discrete FIR filter coefficients

`'Same word length as input'` (default) | `numerictype` object

Data type of discrete FIR filter coefficients, specified as `'Same word length as input'`. Alternatively, you can specify the coefficients data type as a `numerictype` object by calling `numerictype(s,w,f)`. `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits. The object casts the filter coefficients of the discrete FIR filter to the specified data type. The quantization uses nearest rounding and saturate overflow modes. When the input data type is floating point, the `Coefficients` property has no effect.

`OutputDataType` — Data type of discrete FIR filter output

`'Same word length as input'` (default) | `'Full precision'` | `numerictype` object

Data type of discrete FIR filter output, specified as `'Same word length as input'` or `'Full precision'`. Alternatively, you can specify the output data type as a `numerictype` object by calling `numerictype(s,w,f)`. `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits. The object casts the output of the discrete FIR filter to the specified data type. The quantization uses the settings of `RoundingMethod` and `OverflowAction`. When the input data type is floating point, the `OutputDataType` property has no effect.

Control Ports

`ValidInPort` — Enable optional valid input control signal

`true` (default) | `false`

Option to enable valid input control signal, specified as `true` or `false`. When `ValidInPort` is `true`, you must call the object with an extra argument, `validIn`. The first argument of the object, `dataIn`, is treated as valid only when `validIn` is `true`. The extra argument is mandatory when DSP resources are shared. In this case, the object ignores the value of `ValidInPort`.

`ReadyPort` — Enable optional `ready` output control signal

`false` (default) | `true`

Option to enable `ready` output control signal, specified as `true` or `false`. When you set `ReadyPort` to `true`, the object returns an extra value, `ready`. The object sets `ready` to `true` to indicate that it is ready to accept new input data on the next time step.

Methods

step	Filter input with HDL-optimized FIR filter object

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Use HDL FIR Filter System Object with Default Settings

Create an HDL FIR Filter System object with default settings.

```
firFilt = dsp.HDLFIRFilter;
```

Create an input signal of some random noise, and allocate memory for outputs.

```
L = 100;
dataIn = randn(L,1);
dataOut = zeros(L,1);
validOut = false(L,1);
```

Call the object on the input signal, asserting that the input data is always valid. The object processes one data sample at a time.

```
for k=1:L
```

```
    [dataOut(k),validOut(k)] = firFilt(dataIn(k),true);
end
```

Use HDL FIR Filter System Object with Resource Sharing

Operate a 31-tap low pass filter with resource sharing.

Design the filter coefficients. Then create an HDL FIR Filter System object. Enable resource sharing and the `ready` output port.

```
numerator = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
sharingFactor = 10;
firFilt = dsp.HDLFIRFilter('Numerator',numerator, ...
    'Sharing',true,'SharingFactor',sharingFactor,'ReadyPort',true);
```

The specified `sharingFactor` indicates that the filter requires 10 time steps to calculate each output. Create input signals `dataIn` and `validIn` such that new data is applied only every `sharingFactor` time steps.

```
L = 16;
x = fi(randn(L,1),1,16);
dataIn = zeros(L*sharingFactor,1,'like',x);
dataIn(1:sharingFactor:end) = x;
validIn = false(L*sharingFactor,1);
validIn(1:sharingFactor:end) = true;
```

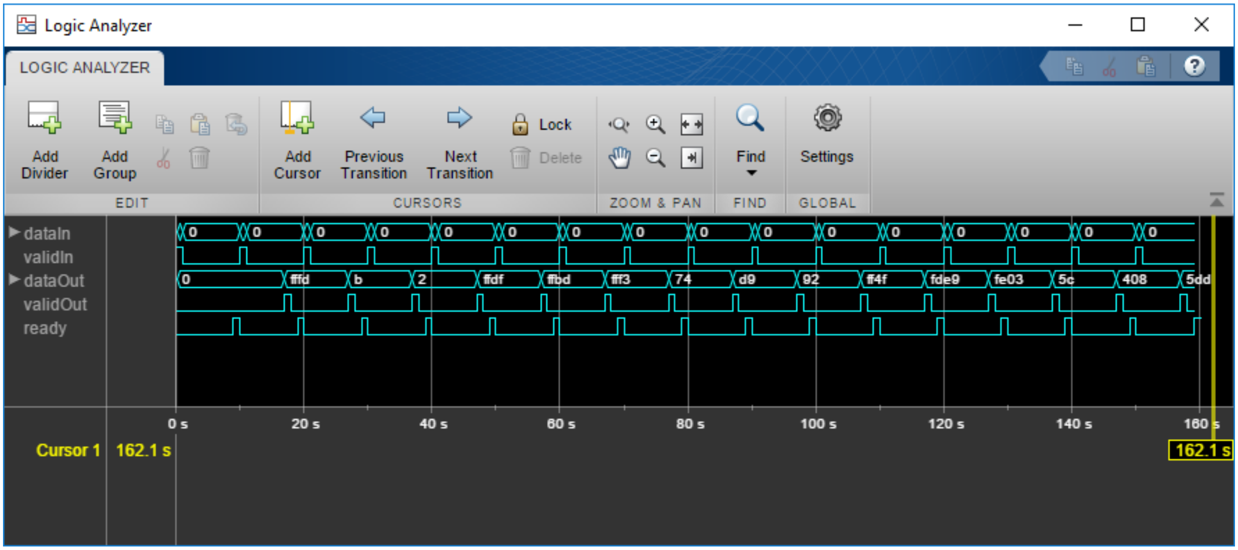
Create a `LogicAnalyzer` object to view the inputs and outputs

```
la = dsp.LogicAnalyzer('NumInputPorts',5, ...
    'SampleTime',1,'TimeSpan',length(dataIn));
tags = getDisplayChannelTags(la);
modifyDisplayChannel(la,tags{1},'Name','dataIn');
modifyDisplayChannel(la,tags{2},'Name','validIn');
modifyDisplayChannel(la,tags{3},'Name','dataOut');
modifyDisplayChannel(la,tags{4},'Name','validOut');
modifyDisplayChannel(la,tags{5},'Name','ready');
```

Call the filter System object on the input signals, and view the results in the **Logic Analyzer**.

```
for k=1:length(dataIn)
    [dataOut,validOut,ready] = firFilt(dataIn(k),validIn(k));
    la(dataIn(k),validIn(k),dataOut,validOut,ready)
```

```
end
```



The object models HDL pipeline registers and resource sharing. Therefore, there is an initial delay before valid output samples are returned by the object.

Create HDL FIR Filter System Object for HDL Code Generation

Use a function containing the HDL FIR filter System object which supports HDL code generation.

Create Function

Write a function that creates and calls an 11-tap HDL FIR Filter System object. You can generate HDL from this function.

```
function [dataOut,validOut] = HDLFIR11Tap(dataIn)
%HDLFIR11TAP
% Process one sample of data using the dsp.HDLFIRFilter System
% object.
% dataIn is a fixed-point scalar value.
% You can generate HDL code from this function.
    persistent fir
    if isempty(fir)
```

```
        Numerator = firpm(10,[0,0.1,0.5,1],[1,1,0,0]);
        fir = dsp.HDLFIRFilter(Numerator,'ValidInPort',false);
    end
    [dataOut,validOut] = fir(dataIn);
end
```

Create Test Bench for the Function

Clear the workspace, create an input signal of some random noise, and allocate memory for outputs.

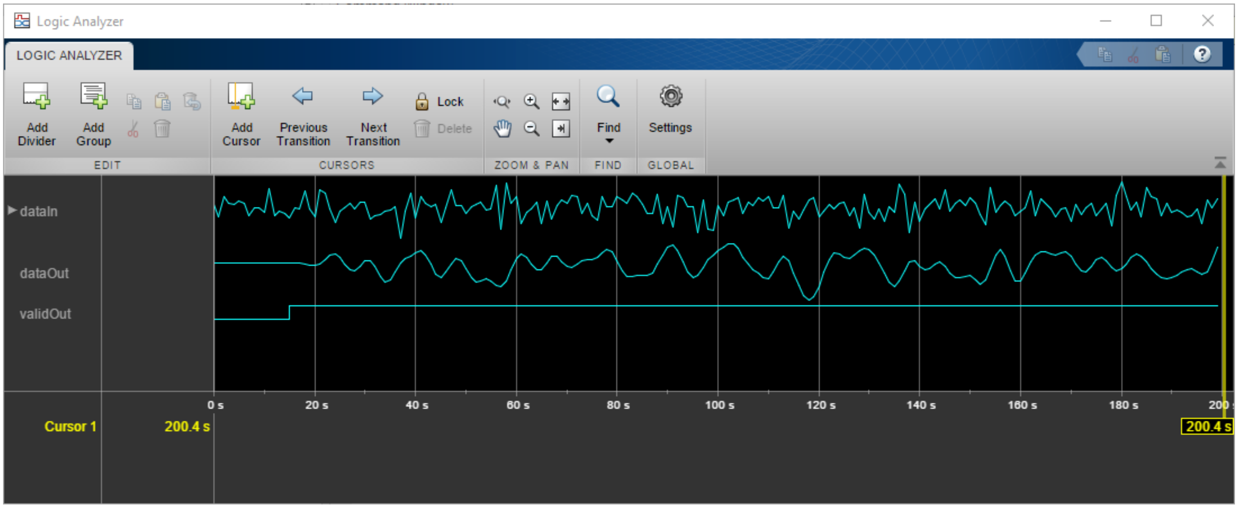
```
clear variables
clear HDLFIR11Tap
L = 200;
dataIn = fi(randn(L,1),1,16);
dataOut = zeros(L,1);
validOut = false(L,1);
```

Call the function on the input signal.

```
for k = 1:L
    [dataOut(k),validOut(k)] = HDLFIR11Tap(dataIn(k));
end
```

Plot the signals with the **Logic Analyzer**.

```
la = dsp.LogicAnalyzer('NumInputPorts',3,'SampleTime',1,'TimeSpan',L);
tags = getDisplayChannelTags(la);
modifyDisplayChannel(la,tags{1},'Name','dataIn','Format','Analog','Height',50);
modifyDisplayChannel(la,tags{2},'Name','dataOut','Format','Analog','Height',50);
modifyDisplayChannel(la,tags{3},'Name','validOut');
la(dataIn,dataOut,validOut)
```



Algorithms

This object implements the algorithms described on the `Discrete FIR Filter HDL Optimized` block reference page.

References

See Also

See Also

Blocks
`Discrete FIR Filter` | `Discrete FIR Filter HDL Optimized`

System Objects
dsp.FIRFilter

Introduced in R2017a

step

System object: dsp.HDLFIRFilter
Package: dsp

Filter input with HDL-optimized FIR filter object

Syntax

```
[dataOut,validOut] = step(firFilt,dataIn)
[dataOut,validOut] = step(firFilt,dataIn,validIn)
[dataOut,validOut,ready] = step(firFilt,dataIn,validIn)
```

Description

`[dataOut,validOut] = step(firFilt,dataIn)` filters the input data, `dataIn`, using the specified `firFilt` filter object. The returned output data, `dataOut`, is valid only when `validOut` is `true`.

`[dataOut,validOut] = step(firFilt,dataIn,validIn)` filters the input data only when `validIn` is true. This argument is required when `Sharing` is set to `true`. Otherwise, it is required only when the `ValidInPort` property of `firFilt` is set to `true`.

`[dataOut,validOut,ready] = step(firFilt,dataIn,validIn)` filters the input data when `validIn` is true, and sets `ready` to `true`, indicating that the object is ready to accept new input data on the next time step. To return `ready`, set the `ReadyPort` property of `firFilt` to `true`.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Input Arguments

`firFilt` — Discrete FIR filter
`dsp.HDLFIRFilter` System object

Discrete FIR filter, specified as a `dsp.HDLFIRFilter` System object.

`dataIn` — Input data
real or complex scalar

Input data, specified as a real or complex scalar. When the input data type is integer or fixed point, the object uses fixed-point arithmetic for internal calculations. `double` and `single` are accepted for simulation but not for HDL code generation.

Data Types: `fi` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`
Complex Number Support: Yes

`validIn` — Indicate valid input data
logical scalar

Indicator of whether input data is valid, specified as a logical scalar. The first argument, `dataIn`, is treated as valid only when `validIn` is `true`. `validIn` is required when `Sharing` is set to `true`. Otherwise, it is required only when `ValidInPort` property is set to `true`.

Data Types: `logical`

Output Arguments

`dataOut` — Filtered output data
real or complex scalar

Filtered output data, returned as a real or complex scalar. When the input data is floating point, the output data inherits the data type of the input data. When the input data is integer or fixed point, the `OutputDataType` property determines the output data type.

Data Types: `fi` | `single` | `double`
Complex Number Support: Yes

`validOut` — Indicates valid output data
logical scalar

Indicator of whether output data is valid, returned as a logical scalar. The object sets `validOut` to `true` with each valid output data on `dataOut`.

Data Types: `logical`

`ready` — Indicates that the object is ready for new input data
logical scalar

Indicator of whether the object is ready for new input data, returned as a logical scalar. The object sets `ready` to `true` to indicate that it is ready to accept new input data on the next time step. The object returns this value only when the `ReadyPort` property is set to `true`.

Data Types: `logical`

See Also

Introduced in R2017a

dsp.FIRHalfbandDecimator System object

Package: dsp

Halfband decimator

Description

`dsp.FIRHalfbandDecimator` performs an efficient polyphase decimation of the input signal by a factor of two. You can use `dsp.FIRHalfbandDecimator` to implement the analysis portion of a two-band filter bank to filter a signal into lowpass and highpass subbands. `dsp.FIRHalfbandDecimator` uses an FIR equiripple design to construct the halfband filters and a polyphase implementation to filter the input. This object supports fixed-point operations and ARM Cortex code generation.

To filter and downsample your data:

1. Define and set up your halfband decimator. See “Construction” on page 4-770.
2. Call step to filter the input signal according to the properties of `dsp.FIRHalfbandDecimator`. The input signal can be a real- or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The number of rows in the input signal must be a multiple of 2.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`firhalfbanddecim = dsp.FIRHalfbandDecimator` returns a halfband decimator, `firhalfbanddecim`, with the default settings. Calling `step` with the default property settings filters and downsamples the input data with a halfband frequency of 11025 Hz, a transition width of 4.1 kHz, and a stopband attenuation of 80 dB.

`firhalfbanddecim = dsp.FIRHalfbandDecimator(Name,Value)` returns a halfband decimator, with additional properties specified by one or more `Name,Value` pair arguments. `Name` is the property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Properties

`Specification` — Filter design parameters

`'Transition width and stopband attenuation'` (default) | `'Filter order and stopband attenuation'` | `'Filter order and transition width'`

Filter design parameters, specified as a character vector. Valid options are `'Transition width and stopband attenuation'` (default), `'Filter order and stopband attenuation'`, or `'Filter order and transition width'`. The filter design has only two degrees of freedom so you can only specify two of the following: filter order, transition width, or stopband attenuation. Setting a filter design specification without additionally specifying values for the design parameters results in default values for the specifications.

`FilterOrder` — Filter order

52 (default) | even positive integer

Filter order, specified as the comma-separated pair of `'FilterOrder'` and an even positive integer. Specifying a filter order is only valid when the value of `‘Specification’` is `'Filter order and stopband attenuation'` or `'Filter order and transition width'`.

`StopbandAttenuation` — Stopband attenuation

80 (default) | positive real scalar

Stopband attenuation in dB, specified as the comma-separated pair of `'StopbandAttenuation'` and a positive real scalar. Specifying the stopband attenuation is only valid when the value of `‘Specification’` is `'Filter order and stopband attenuation'` or `'Transition width and stopband attenuation'`.

`TransitionWidth` — Transition width

4100 (default) | positive real scalar

Transition width in Hz, specified as the comma-separated pair of `'TransitionWidth'` and a positive real scalar. The value of the transition width in Hz must be less than

1/2 the input sample rate. Specifying the transition width is valid only when the value of `‘Specification’` is `'Transition width and stopband attenuation'` or `'Filter order and transition width'`.

`SampleRate` — Input sample rate

44100 (default) | positive real scalar

Input sample rate in Hz, specified as a comma-separated pair of `'SampleRate'` and a positive real scalar. The input sample rate defaults to 44100 Hz. If you specify a transition width as one of your filter design parameters, the transition width cannot exceed 1/2 the input sample rate.

Fixed-Point Properties

`CoefficientsDataType` — Word and fraction lengths of coefficients

`numerictype(1,16)` (default) | `numerictype` object

Word and fraction lengths of coefficients, specified as a signed or unsigned `numerictype` object. The default, `numerictype(1,16)` corresponds to a signed numeric type object with 16-bit coefficients and a fraction length determined based on the coefficient values, to give the best possible precision.

This property is not tunable.

Word length of the output is same as the word length of the input. Fraction length of the output is computed such that the entire dynamic range of the output can be represented without overflow. For details on how the fraction length of the output is computed, see “Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters”.

`RoundingMethod` — Rounding method for output fixed-point operations

`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`

Rounding method for output fixed-point operations, specified as a character vector. For more information on the rounding modes, see “Precision and Range”.

Methods

reset	Reset internal states of FIR halfband decimator

step	Filter input with FIR halfband decimator

For additional methods, see "Analysis Methods for Filter System Objects" on page 3-2.

For a complete list of analysis methods supported for the `dsp.FIRHalfbandDecimator` object, enter `dsp.FIRHalfbandDecimator.helpFilterAnalysis` at the MATLAB command prompt.

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

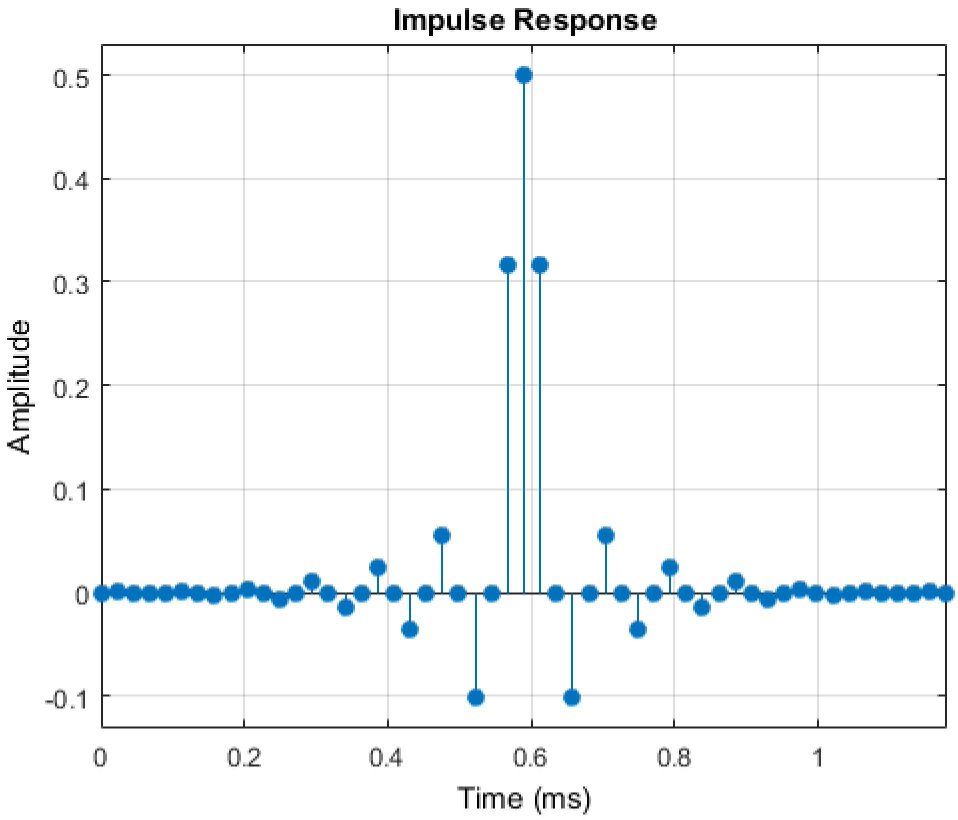
Impulse and Frequency Response of Half-band Decimation Filter

Create a lowpass half-band decimation filter for data sampled at 44.1 kHz. The output data rate is 1/2 the input sampling rate, or 22.05 kHz. Specify the filter order to be 52 with a transition width of 4.1 kHz.

```
Fs = 44.1e3;
filterspec = 'Filter order and transition width';
Order = 52;
TW = 4.1e3;
firhalfbanddecim =dsp.FIRHalfbandDecimator('Specification',filterspec,...
                                            'FilterOrder',Order,...
                                            'TransitionWidth',TW,...
                                            'SampleRate',Fs);
```

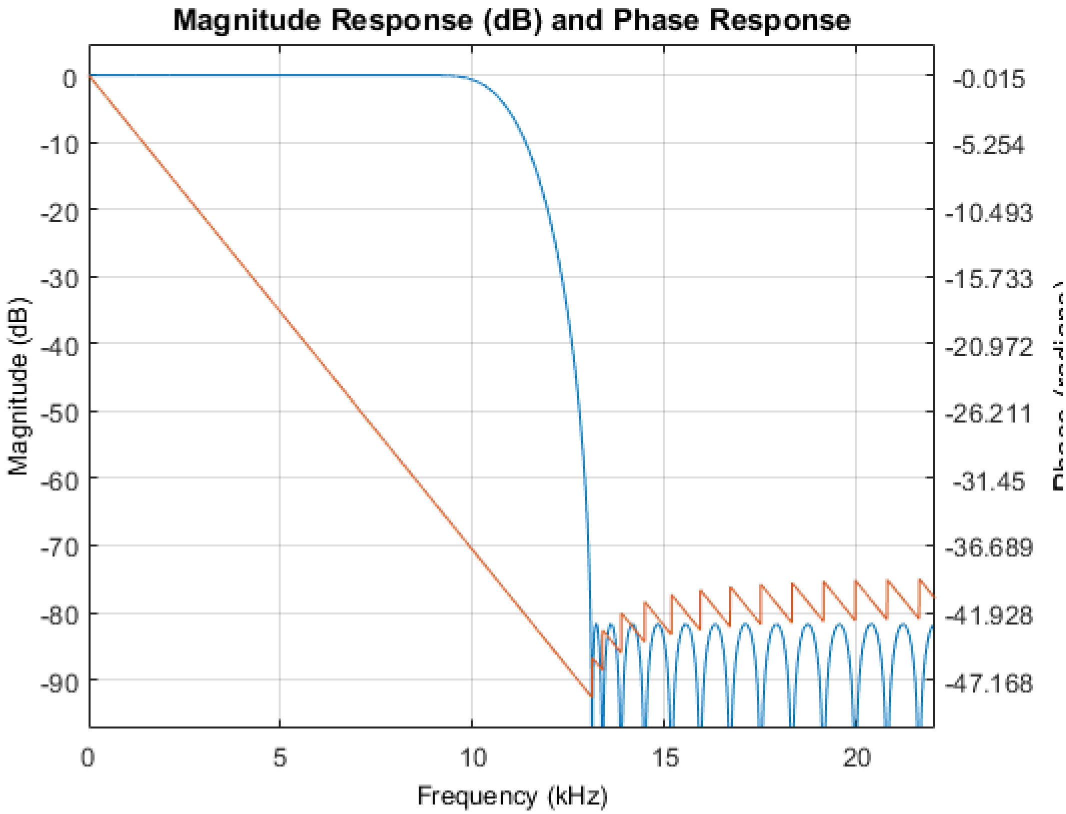
Plot the impulse response. The zero-th order coefficient is delayed 26 samples, which is equal to the group delay of the filter. This yields a causal half-band filter.

```
fvtool(firhalfbanddecim,'Analysis','impulse')
```



Plot the magnitude and phase response.

```
fvtool(firhalfbanddecim,'Analysis','freq')
```



Extract Low Frequency Subband From Speech

Use a halfband analysis filter bank and interpolation filter to extract the low frequency subband from a speech signal.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

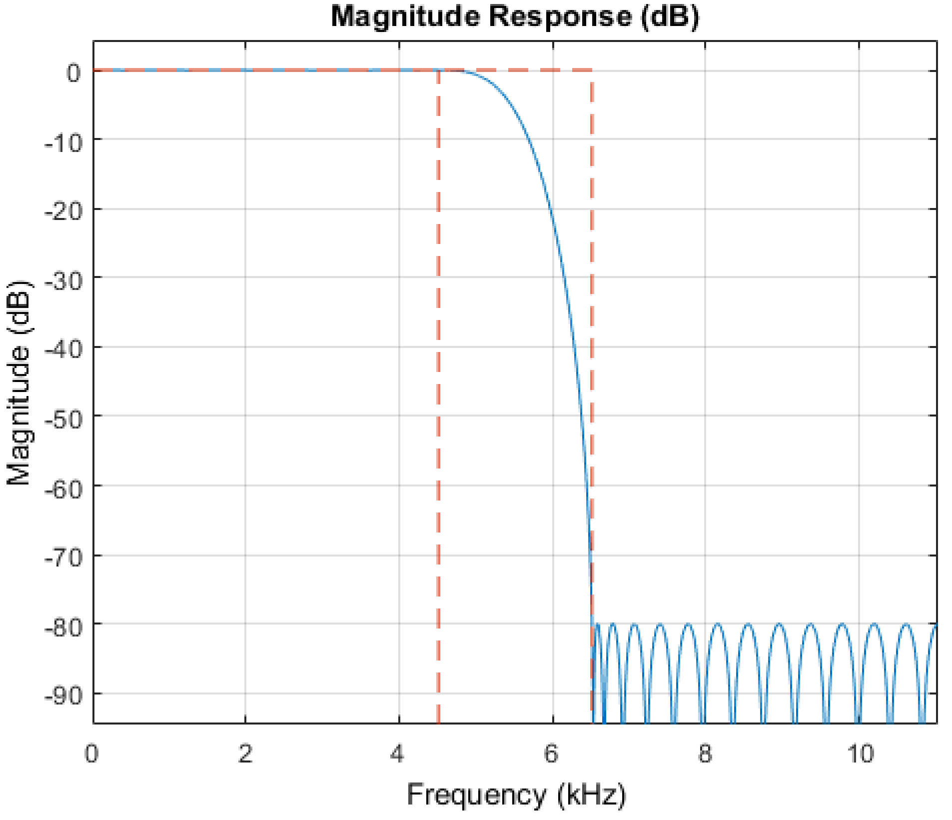
Set up the audio file reader, the analysis filter bank, audio device writer, and interpolation filter. The sampling rate of the audio data is 22050 Hz. The order of the halfband filter is 52, with a transition width of 2 kHz.

```
afr = dsp.AudioFileReader('speech_dft.mp3','SamplesPerFrame',1024);
```

```
filterspec = 'Filter order and transition width';
Order = 52;
TW = 2000;

firhalfbanddecim = dsp.FIRHalfbandDecimator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',afr.SampleRate);

firhalfbandinterp = dsp.FIRHalfbandInterpolator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',afr.SampleRate/2);

adw = audioDeviceWriter('SampleRate',afr.SampleRate);
```

View the magnitude response of the halfband filter.

```
fvtool(firhalfbanddecim)
```



Read the speech signal from the audio file in frames of 1024 samples. Filter the speech signal into lowpass and highpass subbands with a halfband frequency of 5512.5 Hz. Reconstruct a lowpass approximation of the speech signal by interpolating the lowpass subband. Play the filtered output.

```
while ~isDone(afr)
  audioframe = afr();
  xlo = firhalfbanddecim(audioframe);
  ylow = firhalfbandinterp(xlo);
  adw(ylow);
end
```

Wait until the audio file is played to the end, then close the input file and release the audio output resource.

```
release(afr);
release(adw);
```

Two-Channel Filter Bank

Use a halfband decimator and interpolator to implement a two-channel filter bank. This example uses an audio file input and shows that the power spectrum of the filter bank output does not differ significantly from the input.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

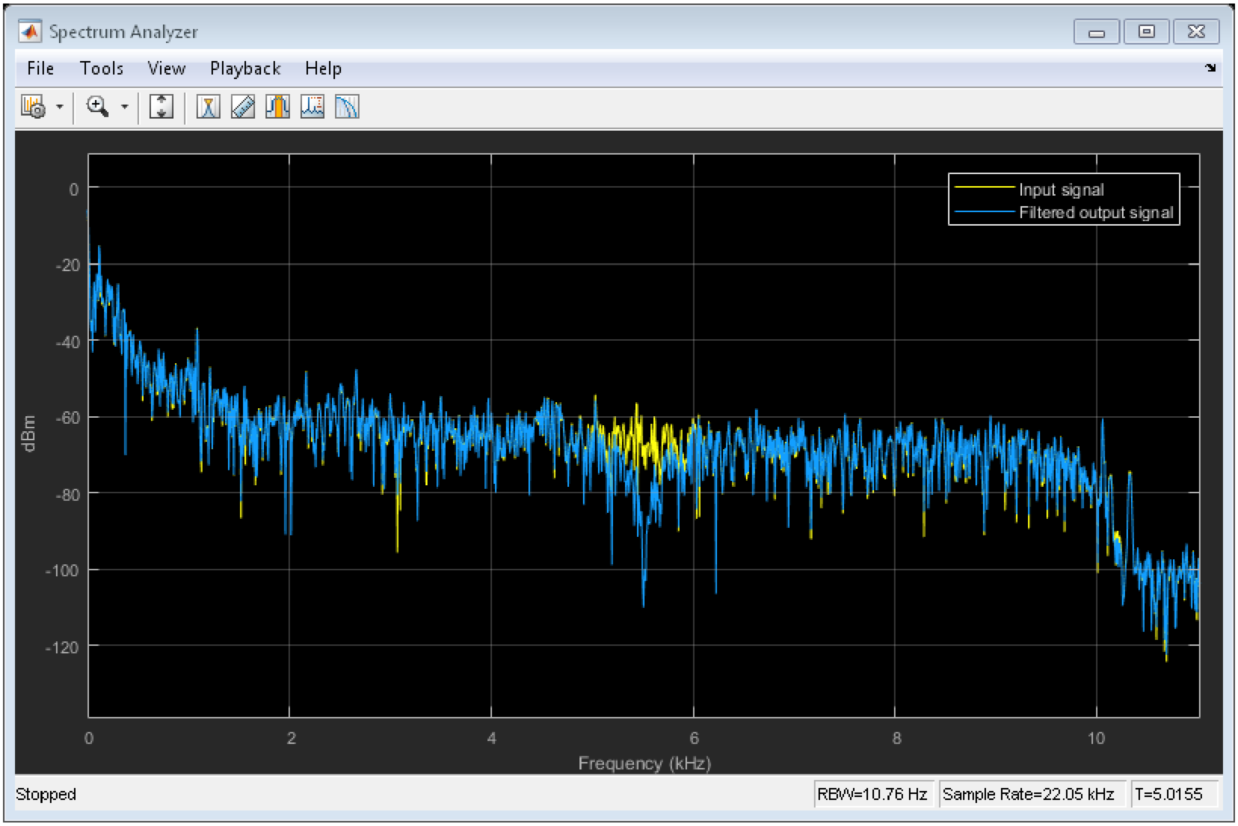
Set up the audio file reader and device writer. Construct the FIR halfband decimator and interpolator. Finally, set up the spectrum analyzer to display the power spectra of the filter-bank input and output.

```
AF = dsp.AudioFileReader('speech_dft.mp3','SamplesPerFrame',1024);
AP = audioDeviceWriter('SampleRate',AF.SampleRate);

filterspec = 'Filter order and transition width';
Order = 52;
TW = 2000;

firhalfbanddecim = dsp.FIRHalfbandDecimator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',AF.SampleRate);

firhalfbandinterp = dsp.FIRHalfbandInterpolator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',AF.SampleRate/2,...
    'FilterBankInputPort',true);

SpecAna = dsp.SpectrumAnalyzer('SampleRate',AF.SampleRate,...
    'PlotAsTwoSidedSpectrum',false,'ReducePlotRate',false,...
    'ShowLegend',true,...
    'ChannelNames',{'Input signal','Filtered output signal'});
```

Read the audio 1024 samples at a time. Filter the input to obtain the lowpass and highpass subband signals decimated by a factor of two. This is the analysis filter bank. Use the halfband interpolator as the synthesis filter bank. Display the running power spectrum of the audio input and the output of the synthesis filter bank. Play the output.

```
while ~isDone(AF)
    audioInput = AF();
```

```
    [xlo,xhigh] = firhalfbanddecim(audioInput);
    audioOutput = firhalfbandinterp(xlo,xhigh);
    spectrumInput = [audioInput audioOutput];
    SpecAna(spectrumInput);
    AP(audioOutput);
end

release(AF);
release(AP);
release(SpecAna);
```



Definitions

Halfband Filters

The ideal lowpass halfband filter is given by

$$h(n) = \frac{1}{2\pi}\int_{-\pi/2}^{\pi/2} e^{j\omega n}\, d\omega = \frac{\sin(\frac{\pi}{2} n)}{\pi n}.$$

The ideal filter is not realizable because the impulse response is noncausal and not absolutely summable. However, the impulse response of the ideal lowpass filter possesses some important properties that are required of a realizable approximation. Specifically, the ideal lowpass halfband filter's impulse response is:

- equal to 0 for all even-indexed samples
- equal to 1/2 at n=0. You can see this by using L'Hopital's rule on the continuous-valued equivalent of the discrete-time impulse response.

The ideal highpass halfband filter is given by

$$g(n) = \frac{1}{2\pi}\int_{-\pi}^{-\pi/2} e^{j\omega n}\, d\omega + \frac{1}{2\pi}\int_{\pi/2}^{\pi} e^{j\omega n}\, d\omega.$$

Evaluating the preceding integral gives the following impulse response

$$g(n) = \frac{\sin(\pi n)}{\pi n} - \frac{\sin(\frac{\pi}{2} n)}{\pi n}.$$

The ideal highpass halfband filter's impulse is:

- equal to 0 for all even-indexed samples
- equal to 1/2 at n=0.

`dsp.FIRHalfbandDecimator` uses a causal FIR approximation to the ideal halfband response, which is based on minimizing the ℓ^∞ norm of the error (minimax). See "Algorithms" on page 4-781 for more details.

Algorithms

Halfband Equiripple Design

`dsp.FIRHalfbandDecimator` uses a minimax FIR design to design a fullband linear phase filter with the desired specifications. The fullband filter is upsampled so that the even-indexed samples of the filter are replaced with zeros. The upsampling of the filter produces a halfband filter. Finally, the filter tap corresponding to the group delay of the filter in samples is set equal to 1/2. This yields a causal linear-phase FIR filter approximation to the ideal halfband filter defined in "Halfband Filters" on page 4-780. See [1] for a description of this filter design method using the Remez exchange algorithm.

Polyphase Implementation with Halfband Filters

`dsp.FIRHalfbandDecimator` uses an efficient polyphase implementation for halfband filters when you call step to filter your signal. The chief advantage of the polyphase implementation is that you can downsample the signal prior to filtering. This allows you to filter at the lower sampling rate.

Splitting a filter's impulse response, *h(n)*, into two polyphase components results in an even polyphase component with *z*-transform

$$H_0(z) = \sum_n h(2n) z^{-n}.$$

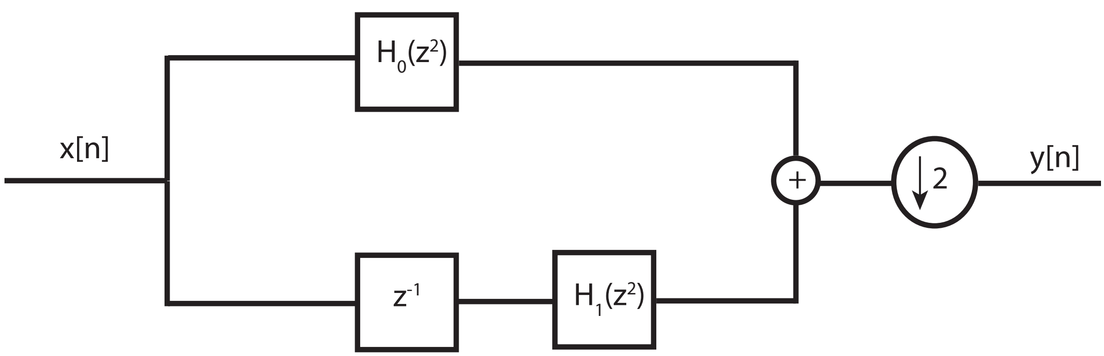
and an odd polyphase component with *z*-transform

$$H_1(z) = \sum_n h(2n+1) z^{-n}.$$

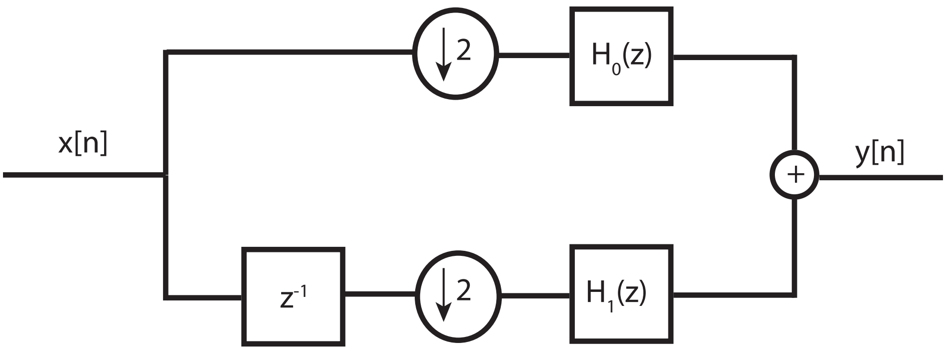
The *z*-transform of the filter can be written in terms of the even and odd polyphase components as

$$H(z) = H_0(z^2) + z^{-1} H_1(z^2).$$

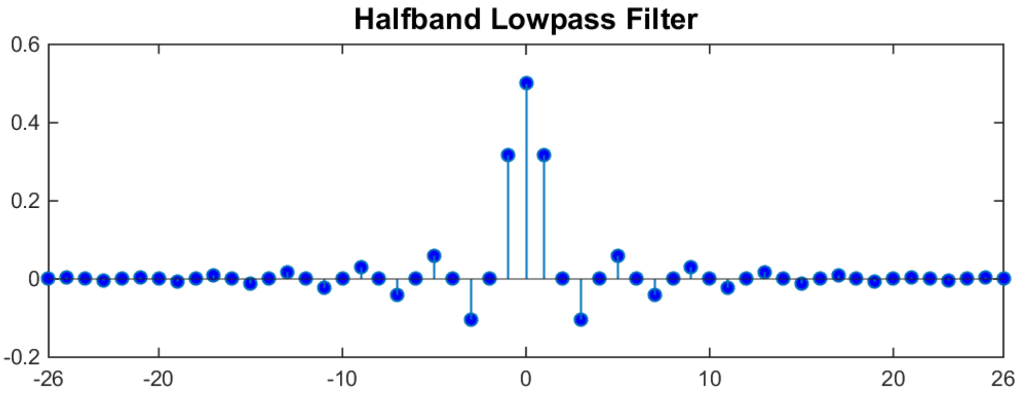
Graphically, you can represent filtering and input followed by downsampling by two with the following figure

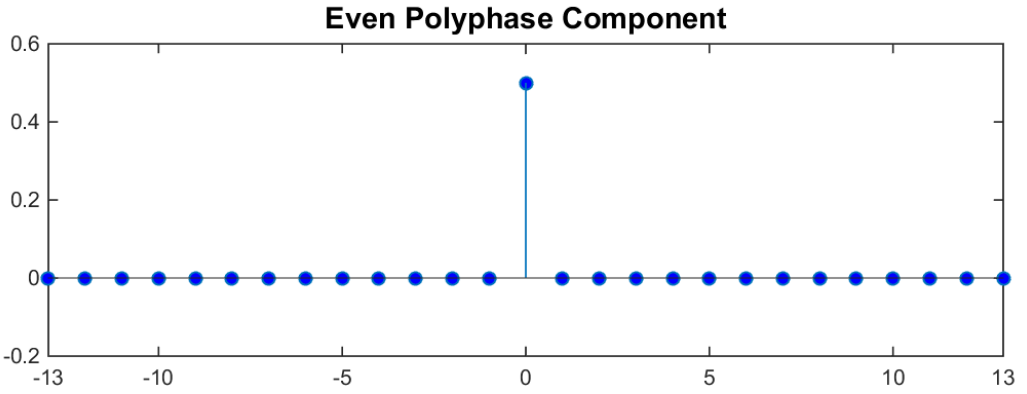


Using the multirate noble identity for downsampling, you can move the downsampling operation before filtering. This allows you to filter at the lower rate.



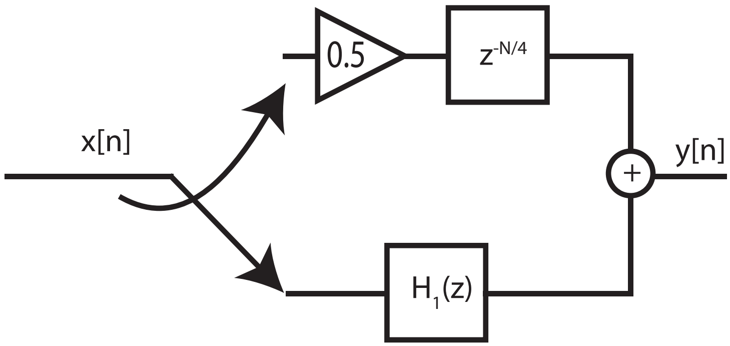
For a halfband filter, the only nonzero coefficient in the even polyphase component is the coefficient corresponding to z^0. Implementing the halfband filter as a causal FIR filter shifts the nonzero coefficient to approximately $z^{-N/4}$ where N is the number of filter taps. This process is illustrated in the following figure.





The top plot shows a halfband filter of order 52. The bottom plot shows the even polyphase component. Both of these filters are noncausal. Delaying the even polyphase component by 13 samples creates a causal FIR filter.

To efficiently implement the halfband decimator, `dsp.FIRHalfbandDecimator` replaces the delay block and downsampling operator with a commutator switch. This is illustrated in the following figure where one polyphase component is replaced by a gain and delay.



The commutator switch takes input samples from a single branch and supplies every other sample to one of the two polyphase components for filtering. This halves the sampling rate of the input signal. Which polyphase component reduces to a simple delay depends on whether the half order of the filter is even or odd. This is because the delay required to make the even polyphase component causal can be odd or even depending on the filter half order. You can see this by inspecting the polyphase components of the following filters.

```
filterspec = 'Filter order and stopband attenuation' ;
halfOrderEven = dsp.FIRHalfbandDecimator('Specification',filterspec,...
    'FilterOrder',64,'StopbandAttenuation',80);
halfOrderOdd = dsp.FIRHalfbandDecimator('Specification',filterspec,...
    'FilterOrder', 54,'StopbandAttenuation',80);
polyphase(halfOrderEven)
polyphase(halfOrderOdd)
```

To summarize, `dsp.FIRHalfbandDecimator`

- decimates the input prior to filtering and filters the even and odd polyphase components of the input separately with the even and odd polyphase components of the filter.
- exploits the fact that one filter polyphase component is a simple delay for a halfband filter.

References

[1] Harris, F.J. *Multirate Signal Processing for Communication Systems*, Prentice Hall, 2004, pp. 208–209.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

System Objects

dsp.IIRHalfbandDecimator | dsp.DyadicAnalysisFilterBank | dsp.Channelizer | dsp.FIRHalfbandInterpolator

Blocks

`Dyadic Analysis Filter Bank` | `FIR Halfband Decimator` | `FIR Halfband Interpolator` | `IIR Halfband Decimator`

Topics

“FIR Halfband Filter Design”

Introduced in R2014b

reset

System object: dsp.FIRHalfbandDecimator
Package: dsp

Reset internal states of FIR halfband decimator

Syntax

```
reset(firhalfbanddecim)
```

Description

`reset(firhalfbanddecim)` resets the filter states of the FIR halfband decimator, `firhalfbanddecim`, to their initial values of 0. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the half-band decimator to nonzero input data, the states may be nonzero. Invoking the `step` method again without first invoking the`reset` method may produce different outputs for an identical input.

step

System object: dsp.FIRHalfbandDecimator
Package: dsp

Filter input with FIR halfband decimator

Syntax

```
Y = step(firhalfbanddecim,X)
[Ylow,Yhigh] = step(firhalfbanddecim,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(firhalfbanddecim,X)` filters the real or complex input signal `X` using the FIR halfband filter, `firhalfbanddecim`, and downsamples the output by a factor of 2 . `Y` is a lowpass halfband filtered and downsampled version of the input `X`. The input `X` must be a column vector or matrix with an even number of rows. If `X` is a matrix, each column is treated as an independent channel. The input `X` supports single, double, and fixed-point data types.

`[Ylow,Yhigh] = step(firhalfbanddecim,X)` outputs the lowpass, `Ylow`, and highpass, `Yhigh`, subbands.

Examples

Filter Input into Lowpass and Highpass Subbands

Create a halfband decimator for data sampled at 44.1 kHz. Use a minimum-order design with a transition width of 2 kHz and a stopband attenuation of 60 dB.

```
hfirhalfbanddecim = dsp.FIRHalfbandDecimator(...
    'Specification','Transition width and stopband attenuation',...
    'TransitionWidth',2000,'StopbandAttenuation',60,'SampleRate',44.1e3);
```

Filter a two-channel input into low and highpass subbands

```
x = randn(1024,2);
[ylow,yhigh] = step(hfirhalfbanddecim,x);
```

dsp.FIRHalfbandInterpolator System object

Package: dsp

Halfband interpolator

Description

`dsp.FIRHalfbandInterpolator` performs efficient polyphase interpolation of the input signal using an upsampling factor of two. You can use `dsp.FIRHalfbandInterpolator` to implement the synthesis portion of a two-band filter bank to synthesize a signal from lowpass and highpass subbands. `dsp.FIRHalfbandInterpolator` uses an FIR equiripple design to construct the halfband filters and a polyphase implementation to filter the input. This object supports fixed-point operations and ARM Cortex code generation.

To upsample and interpolate your data:

1. Define and set up your halfband interpolator. See “Construction” on page 4-790.
2. Call step to filter the input signal according to the properties of `dsp.FIRHalfbandInterpolator`. The input signal can be a real- or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`firhalfbandinterp = dsp.FIRHalfbandInterpolator` returns a FIR halfband interpolation filter, `firhalfbandinterp`, with the default settings. Calling `step` with the default property settings upsamples and interpolates the input data using a halfband frequency of 11025 Hz, a transition width of 4.1 kHz, and a stopband attenuation of 80 dB.

`firhalfbandinterp = dsp.FIRHalfbandInterpolator(Name,Value)` returns a halfband interpolator, with additional properties specified by one or more `Name,Value` pair arguments. `Name` is the property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Properties

`Specification` — Filter design parameters

`'Transition width and stopband attenuation'` (default) | `'Filter order and stopband attenuation'` | `'Filter order and transition width'`

Filter design parameters, specified as a character vector. Valid options are `'Transition width and stopband attenuation'` (default), `'Filter order and stopband attenuation'`, or `'Filter order and transition width'`. The filter design has only two degrees of freedom, so you can specify only two of the following: filter order, transition width, or stopband attenuation. Setting a filter design specification without additionally specifying values for the design parameters results in default values for the specifications.

`FilterOrder` — Filter order

52 (default) | even positive integer

Filter order, specified as the comma-separated pair of `'FilterOrder'` and an even positive integer. Specifying a filter order is valid only when the value of `‘Specification’` is `'Filter order and stopband attenuation'` or `'Filter order and transition width'`.

`StopbandAttenuation` — Stopband attenuation

80 (default) | positive real scalar

Stopband attenuation in dB, specified as the comma-separated pair of `'StopbandAttenuation'` and a positive real scalar. Specifying the stopband attenuation is valid only when the value of `‘Specification’` is `'Filter order and stopband attenuation'` or `'Transition width and stopband attenuation'`.

`TransitionWidth` — Transition width

4100 (default) | positive real scalar

Transition width in Hz, specified as the comma-separated pair of `'TransitionWidth'` and a positive real scalar. The value of the transition width in Hz must be less than 1/2 the output sample rate. Specifying the transition width is valid only when the value of ‘Specification’ is 'Transition width and stopband attenuation' or 'Filter order and transition width'.

SampleRate — Input sample rate
44100 (default) | positive real scalar

Input sample rate in Hz, specified as a comma-separated pair of 'SampleRate' and a positive real scalar. The input sample rate defaults to 44100 Hz. If you specify a transition width as one of your filter design parameters, the transition width cannot exceed 1/2 the input sample rate.

FilterBankInputPort — Synthesis filter bank
false (default) | true

Synthesis filter bank, specified as a comma-separated pair consisting of 'FilterBankInputPort' and a logical value. If this property is false, dsp.FIRHalfbandInterpolator is an interpolation filter for a single vector- or matrix-valued input when you use step. If this property is true, dsp.FIRHalfbandInterpolator is a synthesis filter bank and step accepts two inputs, the lowpass and highpass subbands to synthesize.

Fixed-Point Properties

CoefficientsDataType — Word and fraction lengths of coefficients
numerictype(1,16) (default) | numerictype object

Word and fraction lengths of coefficients, specified as a signed or unsigned numerictype object. The default, numerictype(1,16) corresponds to a signed numeric type object with 16-bit coefficients and a fraction length determined based on the coefficient values, to give the best possible precision.

This property is not tunable.

Word length of the output is same as the word length of the input. Fraction length of the output is computed such that the entire dynamic range of the output can be represented without overflow. For details on how the fraction length of the output is computed, see “Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters”.

RoundingMethod — Rounding method for output fixed-point operations
'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Rounding method for output fixed-point operations, specified as a character vector. For more information on the rounding modes, see “Precision and Range”.

Methods

reset	Reset internal states of FIR halfband interpolator
step	Filter input with FIR halfband interpolator

For additional methods, see “Analysis Methods for Filter System Objects” on page 3-2.

For a complete list of analysis methods supported for the `dsp.FIRHalfbandInterpolator` object, enter `dsp.FIRHalfbandInterpolator.helpFilterAnalysis` at the MATLAB command prompt.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Impulse and Frequency Response of Halfband Interpolation Filter

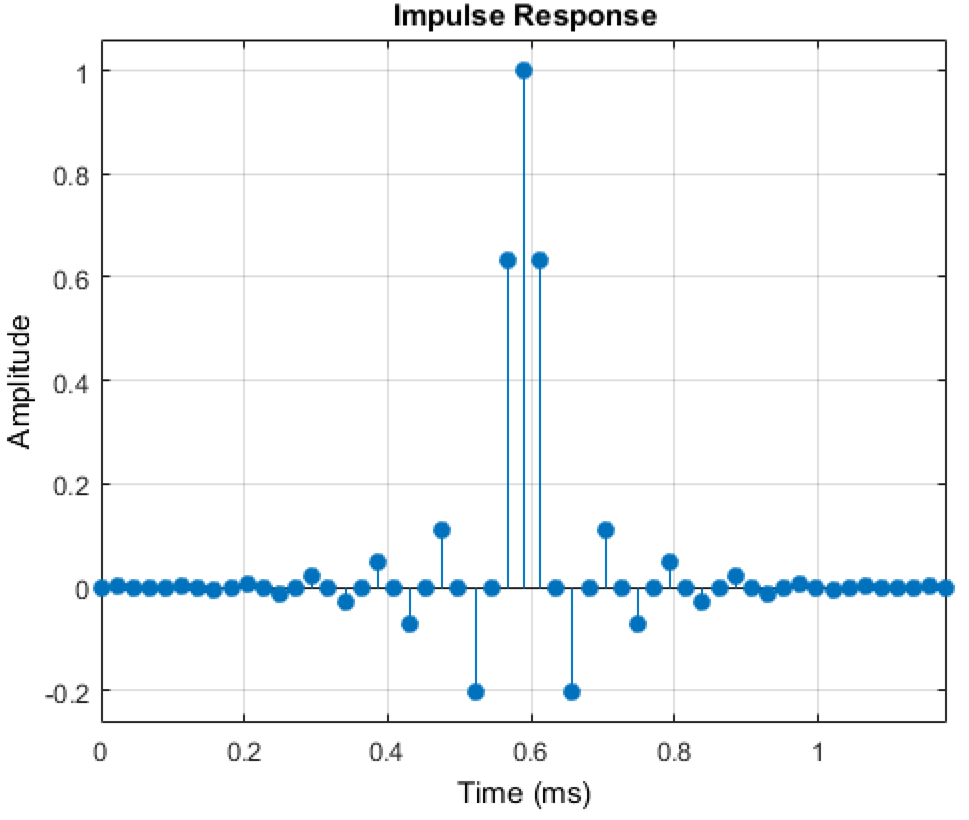
Create a lowpass halfband interpolation filter for upsampling data to 44.1 kHz. Specify a filter order of 52 and a transition width of 4.1 kHz.

```
Fs = 44.1e3;
InputSampleRate = Fs/2;
Order = 52;
```

```
TW = 4.1e3;
filterspec = 'Filter order and transition width';

firhalfbandinterp = dsp.FIRHalfbandInterpolator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',InputSampleRate);
```

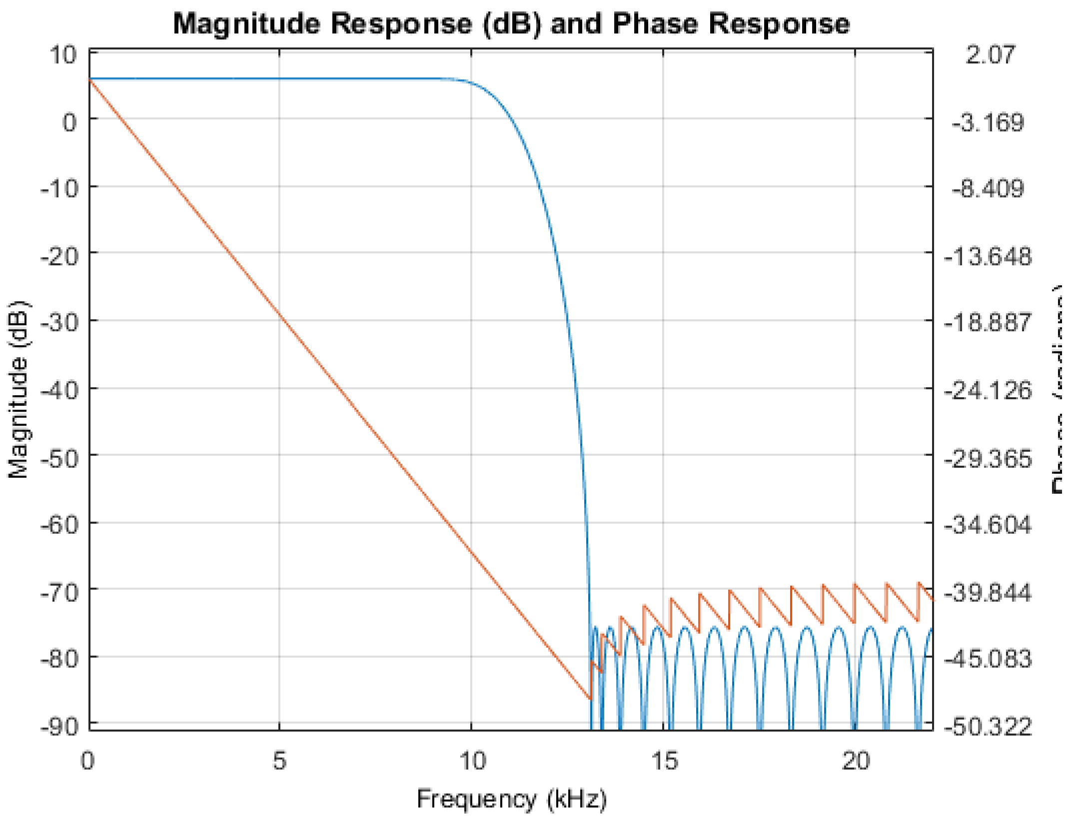
Plot the impulse response. The 0th order coefficient is delayed 26 samples, which is equal to the group delay of the filter. This yields a causal halfband filter.

```
fvtool(firhalfbandinterp,'Analysis','Impulse');
```



Plot the magnitude and phase response.

```
fvtool(firhalfbandinterp,'Analysis','freq');
```



Extract Low Frequency Subband From Speech

Use a halfband analysis filter bank and interpolation filter to extract the low frequency subband from a speech signal.

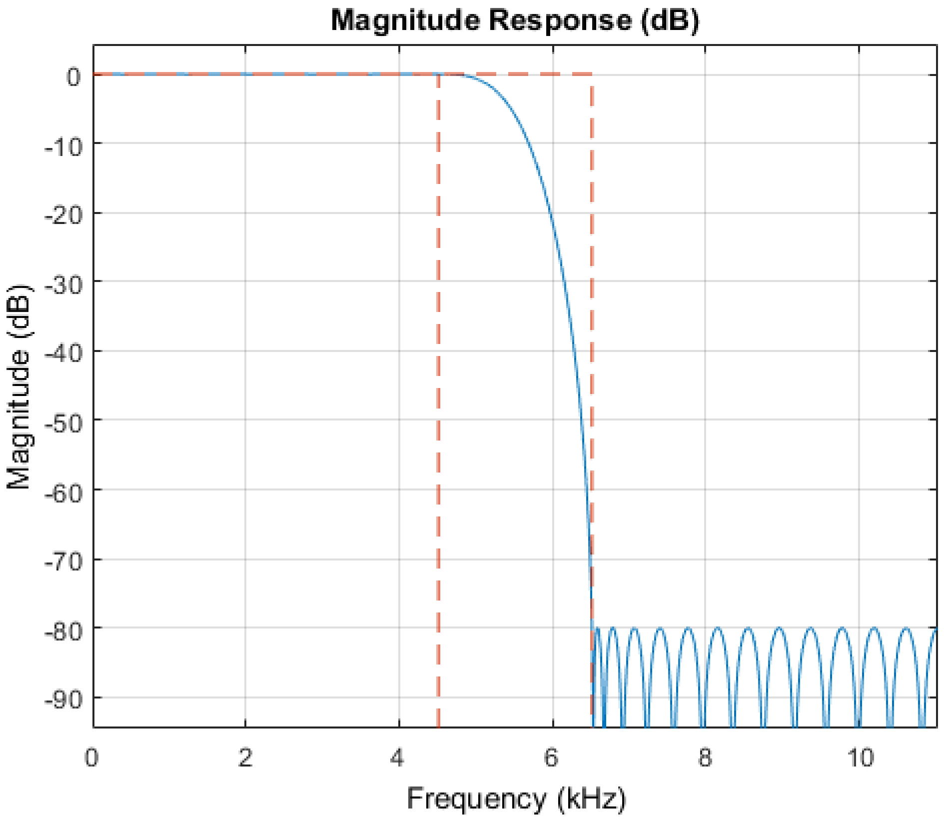
Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Set up the audio file reader, the analysis filter bank, audio device writer, and interpolation filter. The sampling rate of the audio data is 22050 Hz. The order of the halfband filter is 52, with a transition width of 2 kHz.

```
afr = dsp.AudioFileReader('speech_dft.mp3','SamplesPerFrame',1024);

filterspec = 'Filter order and transition width';
Order = 52;
TW = 2000;

firhalfbanddecim = dsp.FIRHalfbandDecimator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',afr.SampleRate);

firhalfbandinterp = dsp.FIRHalfbandInterpolator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',afr.SampleRate/2);

adw = audioDeviceWriter('SampleRate',afr.SampleRate);
```

View the magnitude response of the halfband filter.

```
fvtool(firhalfbanddecim)
```



Read the speech signal from the audio file in frames of 1024 samples. Filter the speech signal into lowpass and highpass subbands with a halfband frequency of 5512.5 Hz. Reconstruct a lowpass approximation of the speech signal by interpolating the lowpass subband. Play the filtered output.

```
while ~isDone(afr)
  audioframe = afr();
  xlo = firhalfbanddecim(audioframe);
  ylow = firhalfbandinterp(xlo);
  adw(ylow);
end
```

Wait until the audio file is played to the end, then close the input file and release the audio output resource.

```
release(afr);
release(adw);
```

Two-Channel Filter Bank

Use a halfband decimator and interpolator to implement a two-channel filter bank. This example uses an audio file input and shows that the power spectrum of the filter bank output does not differ significantly from the input.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

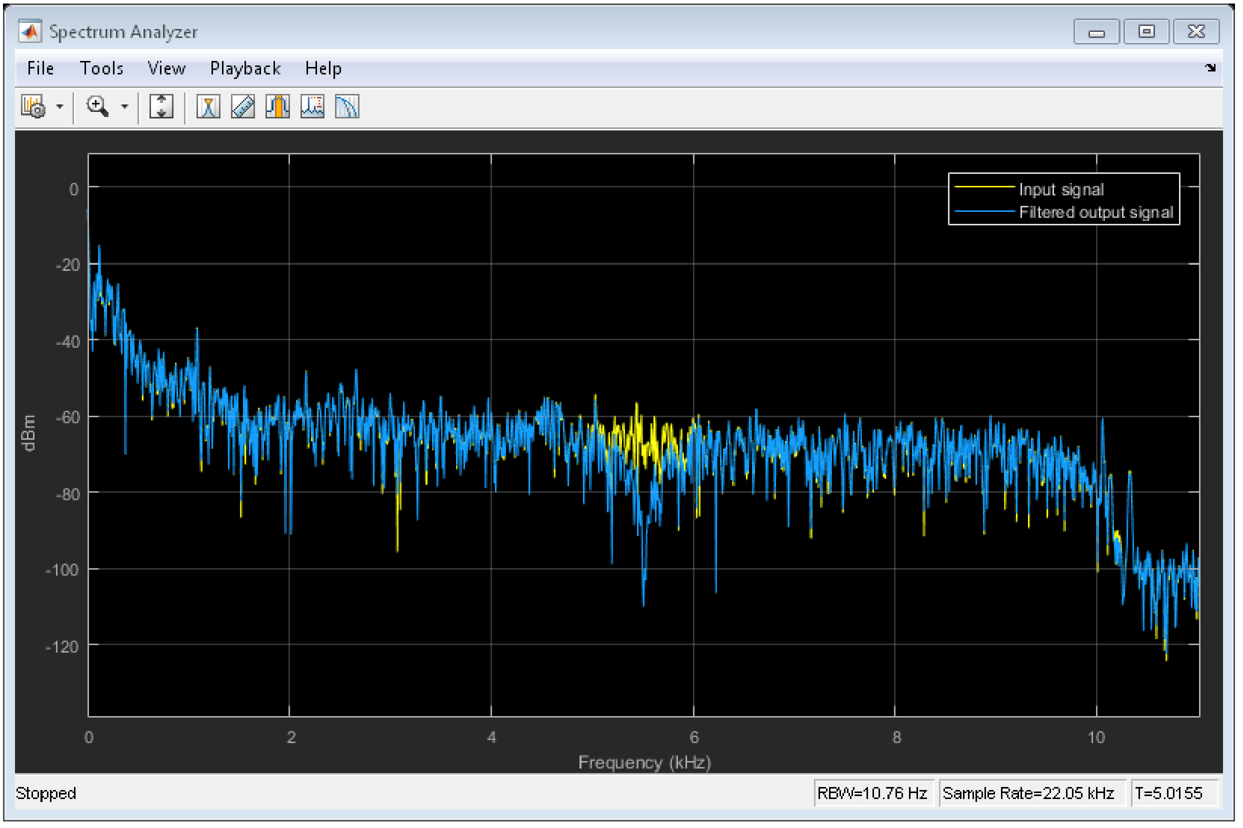
Set up the audio file reader and device writer. Construct the FIR halfband decimator and interpolator. Finally, set up the spectrum analyzer to display the power spectra of the filter-bank input and output.

```
AF = dsp.AudioFileReader('speech_dft.mp3','SamplesPerFrame',1024);
AP = audioDeviceWriter('SampleRate',AF.SampleRate);

filterspec = 'Filter order and transition width';
Order = 52;
TW = 2000;

firhalfbanddecim = dsp.FIRHalfbandDecimator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',AF.SampleRate);

firhalfbandinterp = dsp.FIRHalfbandInterpolator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',AF.SampleRate/2,...
    'FilterBankInputPort',true);

SpecAna = dsp.SpectrumAnalyzer('SampleRate',AF.SampleRate,...
    'PlotAsTwoSidedSpectrum',false,'ReducePlotRate',false,...
    'ShowLegend',true,...
    'ChannelNames',{'Input signal','Filtered output signal'});
```

Read the audio 1024 samples at a time. Filter the input to obtain the lowpass and highpass subband signals decimated by a factor of two. This is the analysis filter bank. Use the halfband interpolator as the synthesis filter bank. Display the running power spectrum of the audio input and the output of the synthesis filter bank. Play the output.

```
while ~isDone(AF)
    audioInput = AF();
```

```
    [xlo,xhigh] = firhalfbanddecim(audioInput);
    audioOutput = firhalfbandinterp(xlo,xhigh);
    spectrumInput = [audioInput audioOutput];
    SpecAna(spectrumInput);
    AP(audioOutput);
end

release(AF);
release(AP);
release(SpecAna);
```



Definitions

Halfband Filters

The ideal lowpass halfband filter is given by

$$h(n)=\frac{1}{2\pi}\int_{-\pi/2}^{\pi/2} e^{j\omega n}\,d\omega=\frac{\sin(\frac{\pi}{2}n)}{\pi n}.$$

The ideal filter is not realizable because the impulse response is noncausal and not absolutely summable. However, the impulse response of the ideal lowpass filter possesses some important properties that are required of a realizable approximation. Specifically, the ideal lowpass halfband filter's impulse response is:

- equal to 0 for all even-indexed samples
- equal to 1/2 at n=0. You can see this by using L'Hopital's rule on the continuous-valued equivalent of the discrete-time impulse response.

The ideal highpass halfband filter is given by

$$g(n) = \frac{1}{2\pi}\int_{-\pi}^{-\pi/2} e^{j\omega n}\, d\omega + \frac{1}{2\pi}\int_{\pi/2}^{\pi} e^{j\omega n}\, d\omega.$$

Evaluating the preceding integral gives the following impulse response

$$g(n) = \frac{\sin(\pi n)}{\pi n} - \frac{\sin(\frac{\pi}{2} n)}{\pi n}.$$

The ideal highpass halfband filter's impulse is:

- Equal to 0 for all even-indexed samples
- Equal to 1/2 at n=0.

`dsp.FIRHalfbandInterpolator` uses a causal FIR approximation to the ideal halfband response, which is based on minimizing the ℓ^{∞} norm of the error (minimax). See "Algorithms" on page 4-801 for more details.

Algorithms

Halfband Equiripple Design

`dsp.FIRHalfbandInterpolator` uses a minimax FIR design to design a fullband linear phase filter with the desired specifications. The fullband filter is upsampled so that the even-indexed samples of the filter are replaced with zeros. The upsampling of the filter produces a halfband filter. Finally, the filter tap corresponding to the group delay of the filter in samples is set equal to 1/2. This yields a causal linear-phase FIR filter approximation to the ideal halfband filter defined in "Halfband Filters" on page 4-800. See [1] for a description of this filter design method using the Remez exchange algorithm.

The coefficients of the halfband interpolation filter are scaled by the interpolation factor, two, to preserve the output power of the signal.

Polyphase Implementation with Halfband Filters

`dsp.FIRHalfbandInterpolator` uses an efficient polyphase implementation for halfband filters when you call step to filter your signal. You can use a polyphase implementation to move the upsampling operation after filtering. This allows you to filter at the lower sampling rate.

Splitting a filter's impulse response, *h(n)*, into two polyphase components results in an even polyphase component with *z*-transform

$$H_0(z) = \sum_n h(2n) z^{-n}.$$

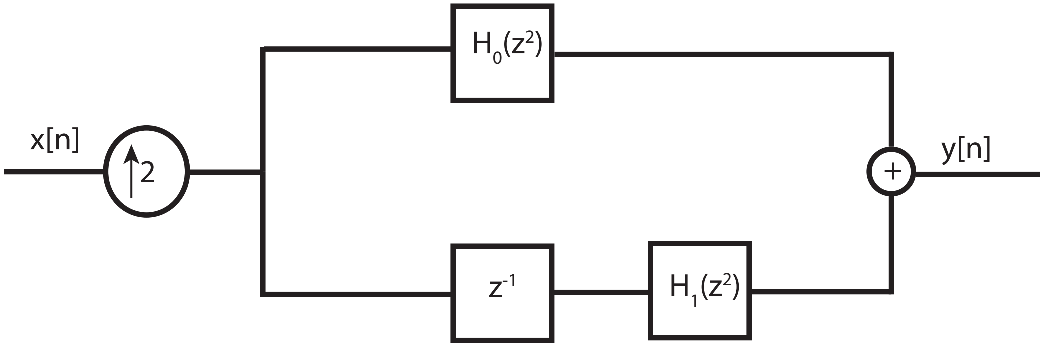
and an odd polyphase component with *z*-transform

$$H_1(z) = \sum_n h(2n+1) z^{-n}.$$

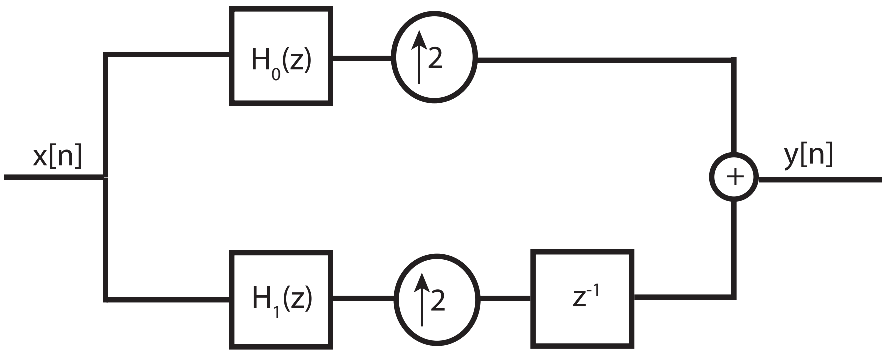
The *z*-transform of the filter can be written in terms of the even and odd polyphase components as

$$H(z) = H_0(z^2) + z^{-1} H_1(z^2).$$

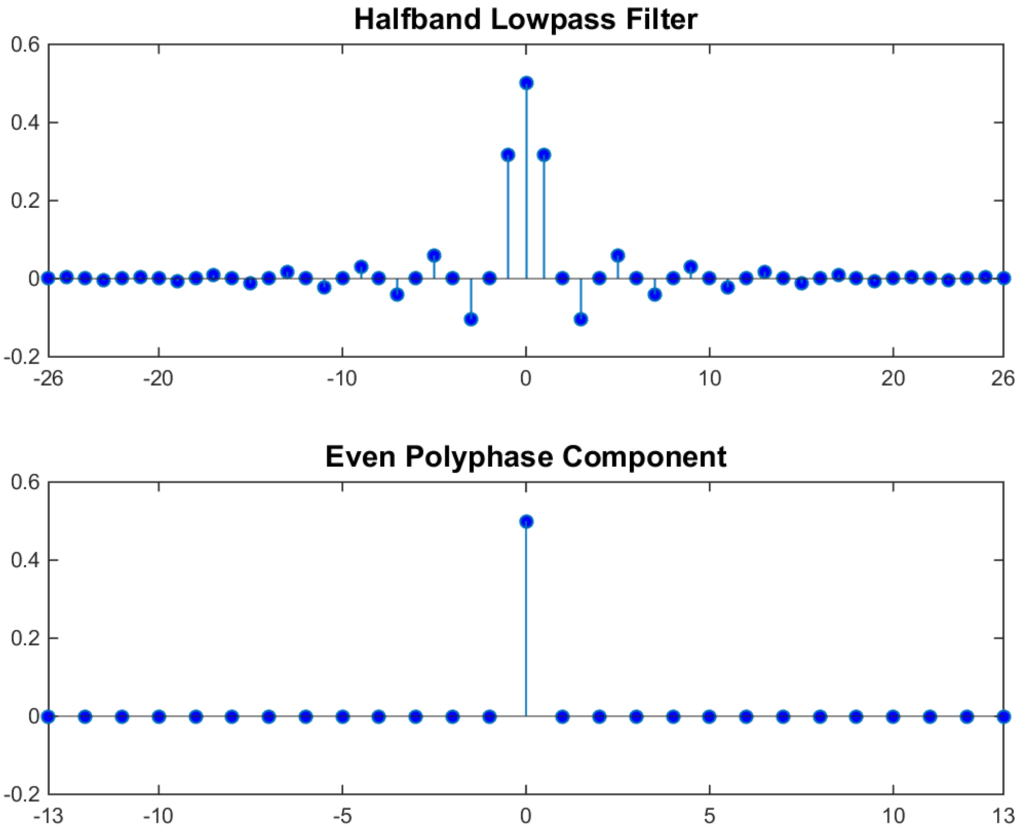
Graphically, you can represent upsampling by two followed by filtering with the following figure



Using the multirate noble identity for upsampling, you can move the upsampling operation after filtering. This enables you to filter at the lower rate.

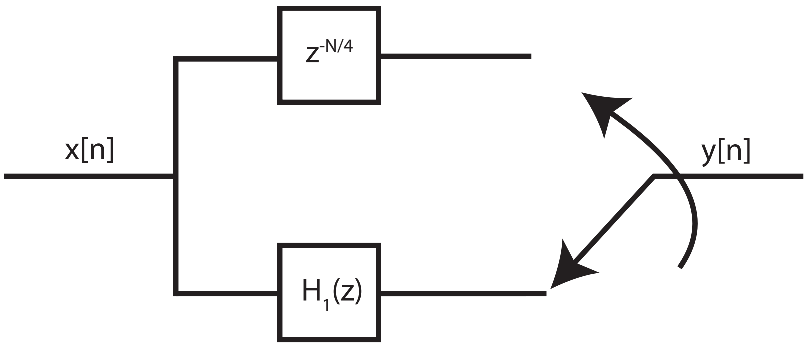


For a halfband filter, the only nonzero coefficient in the even polyphase component is the coefficient corresponding to z^0. Implementing the halfband filter as a causal FIR filter shifts the nonzero coefficient to approximately $z^{-N/4}$, where N is the number of filter taps. This process is shown in the following figure.



The top plot shows a halfband filter of order 52. The bottom plot shows the even polyphase component. Both of these filters are noncausal. Delaying the even polyphase component by 13 samples creates a causal FIR filter.

To efficiently implement the halfband interpolator, `dsp.FIRHalfbandInterpolator` replaces the upsampling operator, delay block, and adder with a commutator switch. This is shown in the following figure, where one polyphase component is replaced by a delay.



The commutator switch takes input samples from the two branches alternately, one sample at a time. This doubles the sampling rate of the input signal. The polyphase component that reduces to a simple delay depends on whether the half order of the filter is even or odd. This is because the delay required to make the even polyphase component causal can be odd or even, depending on the filter half order. For an example of this behavior, inspect the polyphase components of the following filters.

```
filterspec = 'Filter order and stopband attenuation';
halfOrderEven = dsp.FIRHalfbandInterpolator('Specification',filterspec,...
    'FilterOrder',64,'StopbandAttenuation',80);
halfOrderOdd = dsp.FIRHalfbandInterpolator('Specification',filterspec,...
    'FilterOrder', 54,'StopbandAttenuation',80);
polyphase(halfOrderEven)
polyphase(halfOrderOdd)
```

One of the polyphase components has a single nonzero coefficient indicating that it is a simple delay. To preserve the output power of the signal, the coefficients are scaled by the interpolation factor, two. To see this scaling, compare the polyphase components of a halfband interpolator with the coefficients of a halfband decimator.

```
hfirinterp = dsp.FIRHalfbandInterpolator;
hfirdecim = dsp.FIRHalfbandDecimator;
polyphase(hfirdecim)
polyphase(hfirinterp)
```

To summarize, `dsp.FIRHalfbandInterpolator`

- Filters the input before upsampling with the even and odd polyphase components of the filter.
- Exploits the fact that one filter polyphase component is a simple delay for a halfband filter.

References

[1] Harris, F.J. *Multirate Signal Processing for Communication Systems*, Prentice Hall, 2004, pp. 208–209.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

System Objects

dsp.IIRHalfbandInterpolator | dsp.DyadicSynthesisFilterBank | dsp.ChannelSynthesizer | dsp.FIRHalfbandDecimator

Blocks

`Dyadic Synthesis Filter Bank` | `FIR Halfband Decimator` | `FIR Halfband Interpolator` | `IIR Halfband Interpolator`

Topics

“FIR Halfband Filter Design”

Introduced in R2014b

reset

System object: dsp.FIRHalfbandInterpolator
Package: dsp

Reset internal states of FIR halfband interpolator

Syntax

`reset(firhalfbandinterp)`

Description

`reset(firhalfbandinterp)` resets the filter states of the FIR halfband interpolator, `firhalfbandinterp`, to their initial values of 0. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the halfband interpolator to nonzero input data, the states may be nonzero. Invoking the `step` method again without first invoking the`reset` method may produce different outputs for an identical input.

step

System object: dsp.FIRHalfbandInterpolator
Package: dsp

Filter input with FIR halfband interpolator

Syntax

```
Y = step(firhalfbandinterp,X)
Y = step(firhalfbandinterp,X1,X2)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(firhalfbandinterp,X)` upsamples by two and interpolates the real or complex input signal `X` using the FIR halfband interpolator, `firhalfbandinterp`. The input `X` must be a column vector or matrix. If `X` is a matrix, each column is treated as an independent channel. The input `X` supports single, double, and fixed-point data types.

`Y = step(firhalfbandinterp,X1,X2)` implements a halfband synthesis filter bank for the inputs `X1` and `X2`. `X1` is the lowpass output of a halfband analysis filter bank and `X2` is the highpass output of a halfband analysis filter bank. `dsp.FIRHalfbandInterpolator` implements a synthesis filter bank only when the `'FilterBankInputPort'` property is `true`.

Examples

Upsample and Interpolate Multichannel Input

Create a half-band interpolation filter for data sampled at 44.1 kHz. The filter order is 52 with a transition width of 4.1 kHz. Use the filter to upsample and interpolate a multichannel input.

```
Fs = 44.1e3;
filterspec = 'Filter order and transition width';
Order = 52;
TW = 4.1e3;
firhalfbandinterp = dsp.FIRHalfbandInterpolator(...
                                               'Specification',filterspec,...
                                               'FilterOrder',Order,...
                                               'TransitionWidth',TW,...
                                                'SampleRate',Fs);

x = randn(1024,4);
y = step(firhalfbandinterp,x);
```

dsp.FIRInterpolator System object

Package: dsp

Polyphase FIR interpolator

Description

The `FIRInterpolator` object upsamples an input by the integer upsampling factor, *L*, followed by an FIR anti-imaging filter. The filter coefficients are scaled by the interpolation factor. A polyphase interpolation structure implements the filter. The resulting discrete-time signal has a sampling rate *L* times the original sampling rate.

To upsample an input:

1. Define and set up your FIR interpolator. See "Construction" on page 4-810.
2. Call step to upsample the input according to the properties of `dsp.FIRInterpolator`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`firinterp = dsp.FIRInterpolator` returns an FIR interpolator, `firinterp`, which upsamples an input signal by a factor of 3 and applies an FIR filter to interpolate the output.

`firinterp = dsp.FIRInterpolator('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns an FIR interpolator, `firinterp`, with each property set to the specified value.

`firinterp = dsp.FIRInterpolator(INTERP, NUM,'`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns an FIR interpolation object, firinterp, with the InterpolationFactor property set to INTERP, the Numerator property set to NUM, and other properties set to the specified values.

Properties

InterpolationFactor

Interpolation factor

Specify the integer factor, L, by which to increase the sampling rate of the input signal. The polyphase implementation uses L polyphase subfilters to compute convolutions at the lower sample rate. The FIR interpolator delays and interleaves these lower-rate convolutions to obtain the higher-rate output. The property value defaults to 3.

NumeratorSource

FIR filter coefficient source

Specify the source of the numerator coefficients as one of 'Property' (default) or 'Input port'. When you specify 'Input port', the filter object requires the numerator coefficients to be specified as the third argument to every step.

Numerator

FIR filter coefficients

Specify the numerator coefficients of the FIR anti-imaging filter as the coefficients of a polynomial in z^{-1}. Indexing from zero, the filter coefficients are:

$$H(z) = \sum_{n=0}^{N-1} b(n) z^{-n}$$

To act as an effective anti-imaging filter, the coefficients must correspond to a lowpass filter with a normalized cutoff frequency no greater than the reciprocal of the InterpolationFactor on page 4-811. The filter coefficients are scaled by the value of the InterpolationFactor property before filtering the signal. To form the L polyphase subfilters, Numerator is appended with zeros if necessary. The default is the output of fir1(15,0.25). This property is valid only when the NumeratorSource on page 4-811 property is 'Property'.

Fixed-Point Properties

`FullPrecisionOverride`

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

`RoundingMethod`

Rounding method for fixed-point operations

Specify the rounding method as one of `| Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |`. The default is `Floor`. This property applies only if the object is not in full precision mode.

`OverflowAction`

Overflow action for fixed-point operations

Specify the overflow action as one of `| Wrap | Saturate |`. The default is `Wrap`. This property applies only if the object is not in full precision mode.

`CoefficientsDataType`

Coefficient word and fraction lengths

Specify the coefficients fixed-point data type as one of `| Same word length as input | Custom |`. The default is `Same word length as input`.

`CustomCoefficientsDataType`

Coefficient word and fraction lengths

Specify the coefficients fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `CoefficientsDataType` on page 4-812 property is `Custom`. The default is `numerictype([],16,15)`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of | `Full precision` | `Same as input` | `Custom` |. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `ProductDataType` on page 4-813 property is `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of | `Full precision` | `Same as product` | `Same as input` | `Custom` |. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `AccumulatorDataType` on page 4-813 property is `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of | `Same as accumulator` | `Same as product` | `Same as input` | `Custom` |. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `OutputDataType` on page 4-813 property is `Custom`. The default is `numerictype([],16,15)`.

Methods

freqz	Frequency response
fvtool	Open filter visualization tool
impz	Impulse response
phasez	Unwrapped phase response
reset	Reset FIR interpolator filter states
step	Upsample and interpolate input

More “Analysis Methods for Filter System Objects” on page 3-2.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Double the Sample Rate Using FIR Intepolator

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

This example shows how to double the sampling rate of an audio signal from 22.05 kHz to 44.1 kHz, and play the audio.

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

```
adw = audioDeviceWriter(44100);
firinterp = dsp.FIRInterpolator(2, ...
   firpm(30, [0 0.45 0.55 1], [1 1 0 0]));

while ~isDone(afr)
     frame = afr();
     y = firinterp(frame);
     adw(y);
end

pause(1);
release(afr);
release(adw);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `FIR Interpolation` block reference page. The object properties correspond to the block parameters, except:

- The `FIRInterpolator` object does not have a property that corresponds to the **Input processing** parameter of the FIR Interpolation block.
- The **Rate options** block parameter is not supported by the `FIRInterpolator` object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

dsp.FIRRateConverter | dsp.FIRDecimator

Introduced in R2012a

freqz

System object: dsp.FIRInterpolator
Package: dsp

Frequency response

Syntax

```
[h,w] = freqz(firinterp)
[h,w] = freqz(firinterp,n)
[h,w] = freqz(firinterp,Name,Value)
freqz(firinterp)
```

Description

`[h,w] = freqz(firinterp)` returns the complex, 8192–element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample.

`[h,w] = freqz(firinterp,n)` returns the complex, n-element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[h,w] = freqz(firinterp,Name,Value)` returns the frequency response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`freqz(firinterp)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the filter System object `firinterp`.

fvtool

System object: dsp.FIRInterpolator
Package: dsp

Open filter visualization tool

Syntax

```
fvtool(firinterp)
fvtool(firinterp,'Arithmetic',ARITH,...)
```

Description

`fvtool(firinterp)` performs an analysis and computes the magnitude response of the filter System object `firinterp`.

`fvtool(firinterp,'Arithmetic',ARITH,...)` analyzes the filter System object `firinterp`, based on the arithmetic specified in the ARITH input. ARITH can be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The 'Arithmetic' input is only relevant for the analysis of filter System objects.

impz

System object: dsp.FIRInterpolator
Package: dsp

Impulse response

Syntax

```
[h,t] = impz(firinterp)
[h,t] = impz(firinterp,Name,Value)
impz(firinterp)
```

Description

`[h,t] = impz(firinterp)` returns the impulse response `h`, and the corresponding time points `t` at which the impulse response of`firinterp` is computed.

`[h,t] = impz(firinterp,Name,Value)` returns the impulse response `h`, and the corresponding time points`t` , with additional options specified by one or more Name, Value pair arguments.

`impz(firinterp)` uses FVTool to plot the impulse response of the filter System object `firinterp`.

Note: You can use `impz` for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

phasez

System object: dsp.FIRInterpolator
Package: dsp

Unwrapped phase response

Syntax

```
[phi,w] = phasez(firinterp)
[phi,w] = phasez(firinterp,n)
[phi,w] = phasez(firinterp,Name,Value)
phasez(firinterp)
```

Description

`[phi,w] = phasez(firinterp)` returns the 8192–element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample.

`[phi,w] = phasez(firinterp,n)` returns the n-element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[phi,w] = phasez(firinterp,Name,Value)` returns the phase response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`phasez(firinterp)` uses FVTool to plot the phase response of the filter System object `firinterp`.

reset

System object: dsp.FIRInterpolator
Package: dsp

Reset FIR interpolator filter states

Syntax

`reset(firinterp)`

Description

`reset(firinterp)` resets the filter states of the FIR filter in the interpolator object, `firinterp`, to their initial values of 0. The initial filter state values correspond to the initial conditions for the constant coefficient linear difference equation defining the FIR filter. After the `step` method applies the interpolator to nonzero input data, the states may be nonzero. Invoking the `step` method again without first invoking the `reset` method may produce different outputs for an identical input.

Reset an FIR Interpolator

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Reset filter states to 0 to produce consistent output.

```
firinterp = dsp.FIRInterpolator(2);
x =[1 -1]'; x = repmat(x,8,1);
y = firinterp(x); % Filter states are nonzero
```

Use reset method to set states to zero.

```
reset(firinterp);
```

Apply FIR interpolator to input x

```
y1 = firinterp(x);
```

```
isequal(y,y1)
```

```
ans =

  logical

   1
```

step

System object: dsp.FIRInterpolator
Package: dsp

Upsample and interpolate input

Syntax

```
Y = step(firinterp,X)
Y = step(firinterp,X,Num)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(firinterp,X)` outputs the upsampled and interpolated values, `Y` , of the input signal `X` . A K-by-N input matrix is treated as N independent channels. The FIR interpolator object interpolates each channel over the first dimension and generates a M-by-N output matrix, where M is the product of K and the upsampling factor, L.

`Y = step(firinterp,X,Num)` uses the FIR filter, `Num`, to interpolate the input signal. This configuration is valid only when the `'NumeratorSource'` property is `'Input port'`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change

nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.FIRRateConverter System object

Package: dsp

Sample rate converter

Description

The `FIRRateConverter` performs sampling rate conversion by a rational factor on a vector or matrix input. The FIR rate convertor cascades an interpolator with a decimator. The interpolator upsamples the input by the upsampling factor, *L*, followed by a lowpass FIR filter. The FIR filter acts both as an anti-imaging filter and an anti-aliasing filter prior to decimation. The decimator downsamples the output of upsampling and FIR filtering by the downsampling factor *M*. You must use upsampling and downsampling factors that are relatively prime, or coprime. The resulting discrete-time signal has a sampling rate *L*/*M* times the original sampling rate.

To perform sampling rate conversion:

1 Define and set up your FIR sample rate converter. See “Construction” on page 4-825.
2 Call step to perform sampling rate conversion according to the properties of `dsp.FIRRateConverter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`firrc = dsp.FIRRateConverter` returns a FIR sample rate converter, `firrc`, that resamples an input signal at a rate 3/2 times the original sampling rate.

`firrc = dsp.FIRRateConverter('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns an FIR sample rate converter, `firrc`, with each property set to the specified value.

firrc = dsp.FIRRateConverter(L,M,NUM,'*PropertyName*', *PropertyValue*, ...) returns an FIR sample rate converter, firrc, with the InterpolationFactor property set to L, the DecimationFactor property set to M, the Numerator property set to NUM, and other specified properties set to the specified values.

Properties

InterpolationFactor

Interpolation factor

Specify the integer upsampling factor. The default is 3.

DecimationFactor

Decimation factor

Specify the integer downsampling factor. The default is 2.

Numerator

FIR filter coefficients

Specify the FIR filter coefficients in powers of z^{-1}. The length of filter coefficients must exceed the interpolation factor. Use a lowpass with normalized cutoff frequency no greater than min(1/InterpolationFactor, 1/DecimationFactor). All initial filter states are zero. The default is firpm(70,[0 0.28 0.32 1],[1 1 0 0]).

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set FullPrecisionOverride to true, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero` |. The default is `Floor`. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | `Wrap` | `Saturate` |. The default is `Wrap`. This property applies only if the object is not in full precision mode.

CoefficientsDataType

Word and fraction lengths of filter coefficients

Specify the filter coefficient fixed-point data type as one of | `Same word length as input` | `Custom` |. The default is `Same word length as input`.

CustomCoefficientsDataType

Word and fraction lengths of filter coefficients

Specify the filter coefficient fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `CoefficientsDataType` on page 4-827 property is `Custom`. The default is `numerictype([],16,15)`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of | `Full precision` | `Same as input` | `Custom` |. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when the ProductDataType on page 4-827 property is Custom. The default is numerictype([],32,30).

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of | Full precision | Same as product | Same as input | Custom |. The default is Full precision.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when the AccumulatorDataType on page 4-828 property is Custom. The default is numerictype([],32,30).

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of | Same as accumulator | Same as product | Same as input | Custom |. The default is Same as accumulator.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when the OutputDataType on page 4-828 property is Custom. The default is numerictype([],16,15).

Methods

reset	Reset states of FIR sample rate converter

step	Resample input with FIR rate converter

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Resample a Signal using FIR Rate Converter

This example shows how to resample a 100 Hz sine wave signal by a factor of 3:2.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

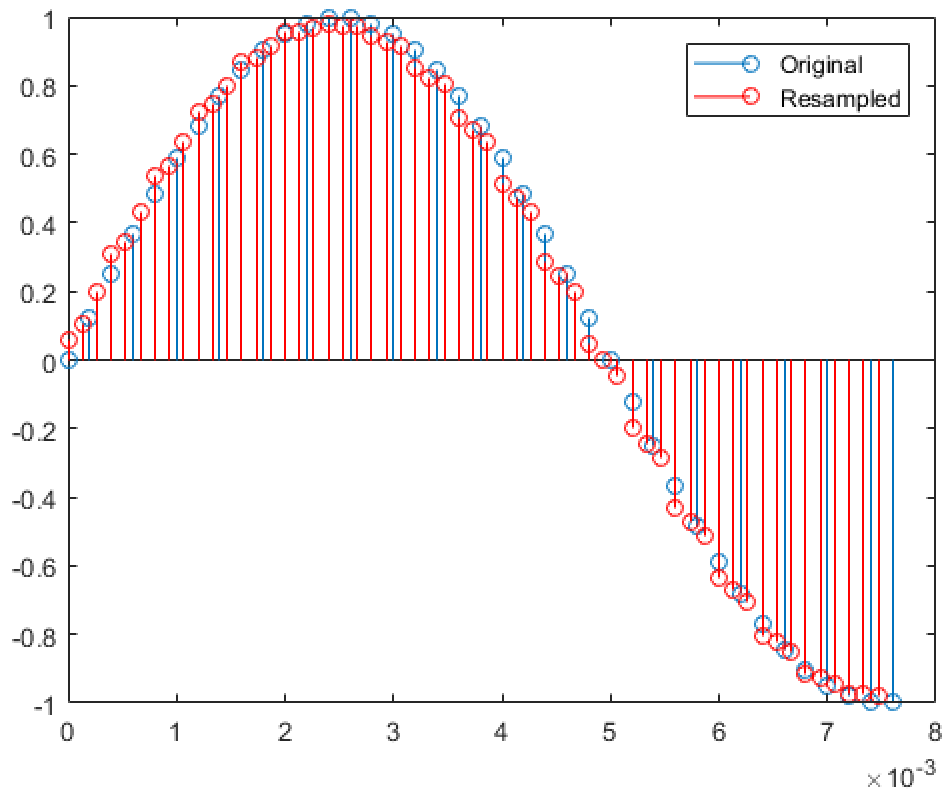
```
sine = dsp.SineWave(1, 100,'SampleRate', 5000,'SamplesPerFrame', 50);
```

Create a FIR rate converter filter. The default interpolation factor is 3 and decimation factor is 2.

```
firrc = dsp.FIRRateConverter;
input = sine();
output = firrc(input);
```

Plot the original and resampled signals.

```
ndelay = round(length(firrc.Numerator)/2/firrc.DecimationFactor);
indx = ndelay+1:length(output);
x = (0:length(indx)-1)/sine.SampleRate*firrc.DecimationFactor/firrc.InterpolationFacto
stem((0:38)/sine.SampleRate, input(1:39)); hold on;
stem(x, firrc.InterpolationFactor*output(indx),'r');
legend('Original','Resampled');
```



Resample and Play an Audio Signal

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

This example shows how to resample and play an audio signal from 48 kHz to 32 kHz on a Windows® platform.

```
afr = dsp.AudioFileReader('audio48kHz.wav', ...
    'OutputDataType', 'single', ...
    'SamplesPerFrame', 300);
adw = audioDeviceWriter(32000);
```

Create an FIRRateConverter System object with interpolation factor = 2, decimation factor = 3. Default FIR filter coefficients define a lowpass filter with normalized cutoff frequency of 1/3.

```
firrc = dsp.FIRRateConverter(2,3);

while ~isDone(afr)
    audio1 = afr();
    audio2 = firrc(audio1);
    adw(audio2);
end

release(afr);
release(adw);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `FIR Rate Conversion` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

dsp.FIRDecimator | dsp.FIRInterpolator

Introduced in R2012a

reset

System object: dsp.FIRRateConverter
Package: dsp

Reset states of FIR sample rate converter

Syntax

```
reset(firrc)
```

Description

`reset(firrc)` resets the filter states of the FIR filter in the sample rate converter, `firrc`, to their initial values of 0. The initial filter state values correspond to the initial conditions for the constant coefficient linear difference equation defining the FIR filter. After the `step` method applies the FIR rate converter to nonzero input data, the states may be nonzero. Invoking the `step` method again without first invoking the `reset` method may produce different outputs for an identical input.

Reset an FIR Rate Converter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Reset filter states to 0 to produce consistent output.

```
firrc = dsp.FIRRateConverter(2,4);
x =[1 -1]'; x = repmat(x,8,1);
y = firrc(x); % Filter states are nonzero
```

Use reset method to set states to zero.

```
reset(firrc);
```

Apply sampling rate converter to input x

```
y1 = firrc(x);
```

```
isequal(y,y1)

ans =

  logical

   1
```

step

System object: dsp.FIRRateConverter
Package: dsp

Resample input with FIR rate converter

Syntax

```
Y = step(firrc,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(firrc,X)` resamples the input `X` and returns the resampled signal `Y` . An *M*-by-*N* matrix input is treated as *N* independent channels.

dsp.HDLFIRRateConverter System object

Package: dsp

Upsample, filter, and downsample—optimized for HDL code generation

Description

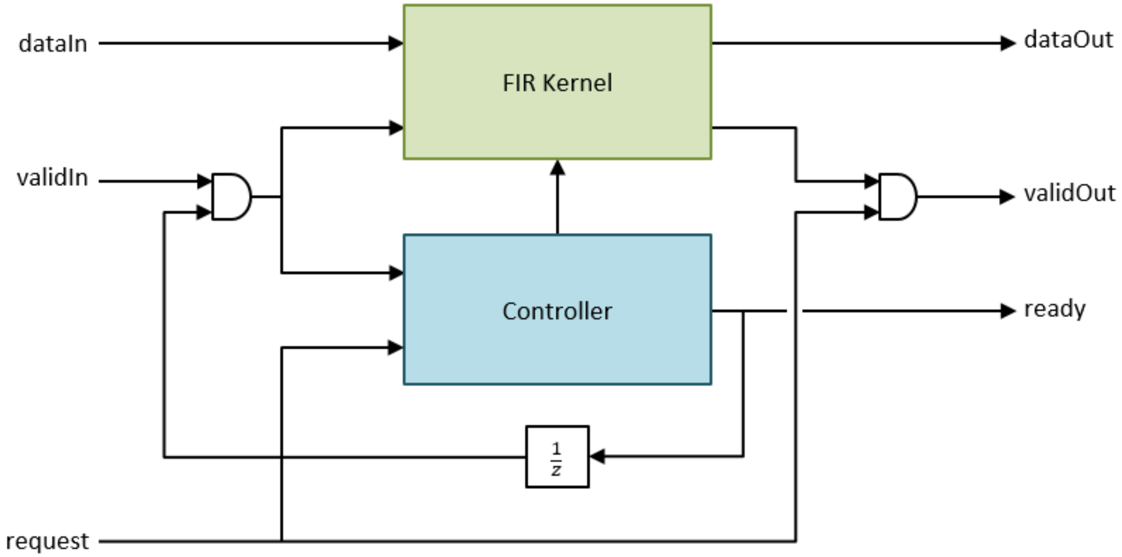
The `HDLFIRRateConverter` System object upsamples, filters, and downsamples input signals. It is optimized for HDL code generation and operates on one sample of each channel at a time. The object implements an efficient polyphase architecture to avoid unnecessary arithmetic operations and high intermediate sample rates.

The object upsamples by an integer factor of L, applies an FIR filter, and downsamples by an integer factor of M.

To resample and filter input data:

1. Define and configure a FIR rate converter System object. See "Construction" on page 4-837.
2. Call the step method to compute each output data sample according to the "Properties" on page 4-837 of the object you created. The object models HDL pipeline latency, so there is an initial delay of several `step` calls before the object returns the first valid output sample. The behavior of `step` is specific to each System object.

The `step` method accepts and returns control signal arguments for pacing the flow of samples in and out of the System object. The `step` method uses `validIn` and `validOut` control signals by default. You can also enable a `ready` output signal and a `request` input signal.



The `ready` output from the `step` method indicates that the object can accept a new input data sample on the next call to `step`. When $L \geq M$, you can use the `ready` argument to achieve continuous output data samples. If you apply a new input sample after each time the `step` method returns `ready` = `true`, each call to `step` returns a data output sample with `validOut` = `true`.

When you do not enable the `ready` argument, you can apply a valid data sample only every `ceil(L/M)` calls to `step`. For example:

- L/M = 4/5 — You can apply a new input sample on every call to `step`.
- L/M = 3/2— You can apply a new input sample on every other call to `step`.

When you enable the `request` input to the `step` method, the object returns the next output sample when `request` is `true` and a valid output sample is available. When you do not use `request`, the object returns output samples when they are available. Calls to `step` that do not return a new sample return `validOut` = `false`.

You can connect `request` to the `ready` output of a downstream object.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`HDLFIRRC = dsp.HDLFIRRateConverter` returns a System object, `HDLFIRRC`, which resamples each channel of the input. The object upsamples by an integer factor of L, applies an FIR filter, and downsamples by an integer factor of M. The default L/M is 3/2.

`HDLFIRRC = dsp.HDLFIRRateConverter(Name,Value)` returns a System object, `HDLFIRRC`, with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

`HDLFIRRC = dsp.HDLFIRRateConverter(L,M,NUM,Name,Value)` returns a System object, `HDLFIRRC`, with the `InterpolationFactor` property set to `L`, the `DecimationFactor` property set to `M`, the `Numerator` property set to `NUM`, and other specified properties set to the specified values.

Properties

`InterpolationFactor`

Upsampling factor, L, specified as a scalar positive integer.

Default: 3

`DecimationFactor`

Downsampling factor, M, specified as a scalar positive integer.

Default: 2

`Numerator`

FIR filter coefficients, specified as a vector in descending powers of z^{-1}.

You can generate filter coefficients using the Signal Processing Toolbox filter design functions (such as `fir1`). Design a lowpass filter with normalized cutoff frequency no greater than `min(1/L,1/M)`. The object initializes internal filter states to zero.

Default: `firpm(70,[0 .28 .32 1],[1 1 0 0])`

`ReadyPort`

Set `ReadyPort` to `true` to return a logical scalar value, `ready`, from the `step` method. This value indicates whether the object will be ready for a new input sample the next time you call the `step` method.

Default: `false`

`RequestPort`

Set `RequestPort` to `true` to enable a `request` argument to the `step` method. This argument is a logical scalar value that requests a new output sample on the current call to the `step` method.

Default: `false`

`RoundingMethod`

Rounding mode used for fixed-point operations. This property does not apply when the input is single or double type. '`Simplest`' mode is not supported.

Default: `Floor`

`OverflowAction`

Overflow mode used for fixed-point operations. This property does not apply when the input is single or double type.

Default: `Wrap`

CoefficientsDataType

Data type of the FIR filter coefficients, specified as a `numerictype(s,wl,fl)` object with `signedness`, `word length`, and `fractional length` properties.

Default: `numerictype(1,16,16)`

OutputDataType

Data type of the output data samples, specified as one of `'Same word length as input'`, `'Full precision'`, or as a `numerictype(s,wl,fl)` object.

Default: `'Same word length as input'`

Methods

step	Resample and filter input sample

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Downsample Signal

Convert a signal from 48 kHz to 32 kHz using the HDL FIR Rate Converter System object™.

Define the sample rate and length of the input signal, and a 2 kHz cosine waveform. Set `validIn` = `true` for every sample.

```
Fs = 48e3;
Ns = 100;
t = (0:Ns-1).'/Fs;
dataIn = cos(2*pi*2e3*t);
validIn = true(Ns,1);
```

Preallocate `dataOut` and `validOut` signals for faster simulation.

```
dataOut = zeros(Ns,1);
validOut = false(Ns,1);
```

Create the System object. Configure it to perform rate conversion by a factor of 2/3, using an equiripple filter.

```
Numerator = firpm(70, [0,.25,.32,1], [1,1,0,0]);
firrc = dsp.HDLFIRRateConverter(2,3,Numerator);
```

Call the System object to perform the rate conversion and obtain each output sample.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

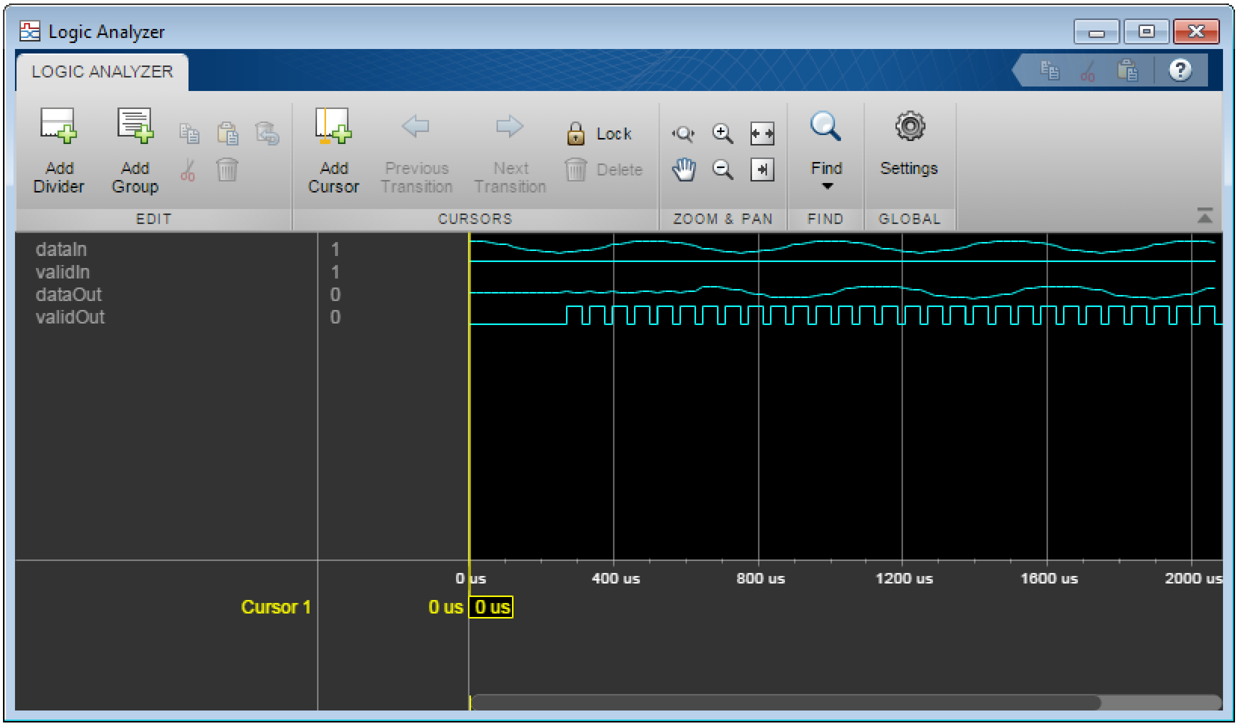
```
for k = 1:Ns
    [dataOut(k),validOut(k)] = firrc(dataIn(k),validIn(k));
end
```

Because the input sample rate is higher than the output sample rate, not every member of `dataOut` is valid. Use `validOut` to extract the valid samples from `dataOut`.

```
y = dataOut(validOut);
```

View the input and output signals with the **Logic Analyzer**.

```
la = dsp.LogicAnalyzer('NumInputPorts',4,'SampleTime',1/Fs,'TimeSpan',Ns/Fs);
modifyDisplayChannel(la,1,'Name','dataIn','Format','Analog','Height',8)
modifyDisplayChannel(la,2,'Name','validIn')
modifyDisplayChannel(la,3,'Name','dataOut','Format','Analog','Height',8)
modifyDisplayChannel(la,4,'Name','validOut')
la(dataIn,validIn,dataOut,validOut)
```



Upsample Signal

Convert a signal from 40 MHz to 100 MHz using the HDL FIR Rate Converter System object™. To avoid overrunning the object as the signal is upsampled, control the input rate manually.

Define the sample rate and length of the input signal, and a fixed-point cosine waveform.

```
Fs = 40e6;
Ns = 50;
t = (0:Ns-1).'/Fs;
x = fi(cos(2*pi*1.2e6*t), 1, 16, 14);
```

Define the rate conversion parameters. Use an interpolation factor of 5 and a decimation factor of 2. Calculate how often the object can accept a new input sample.

```
L = 5;
M = 2;
```

```
stepsPerInput = ceil(L/M);
numSteps = stepsPerInput*Ns;
```

Generate `dataIn` and `validIn` based on how often the object can accept a new sample.

```
dataIn = zeros(numSteps,1,'like',x);
dataIn(1:stepsPerInput:end) = x;
validIn = false(numSteps,1);
validIn(1:stepsPerInput:end) = true;
```

Create the System object. Configure it to perform rate conversion using the specified factors and an equiripple FIR filter.

```
Numerator = firpm(70, [0,.15,.25,1], [1,1,0,0]);
rateConverter = dsp.HDLFIRRateConverter(L,M,Numerator);
```

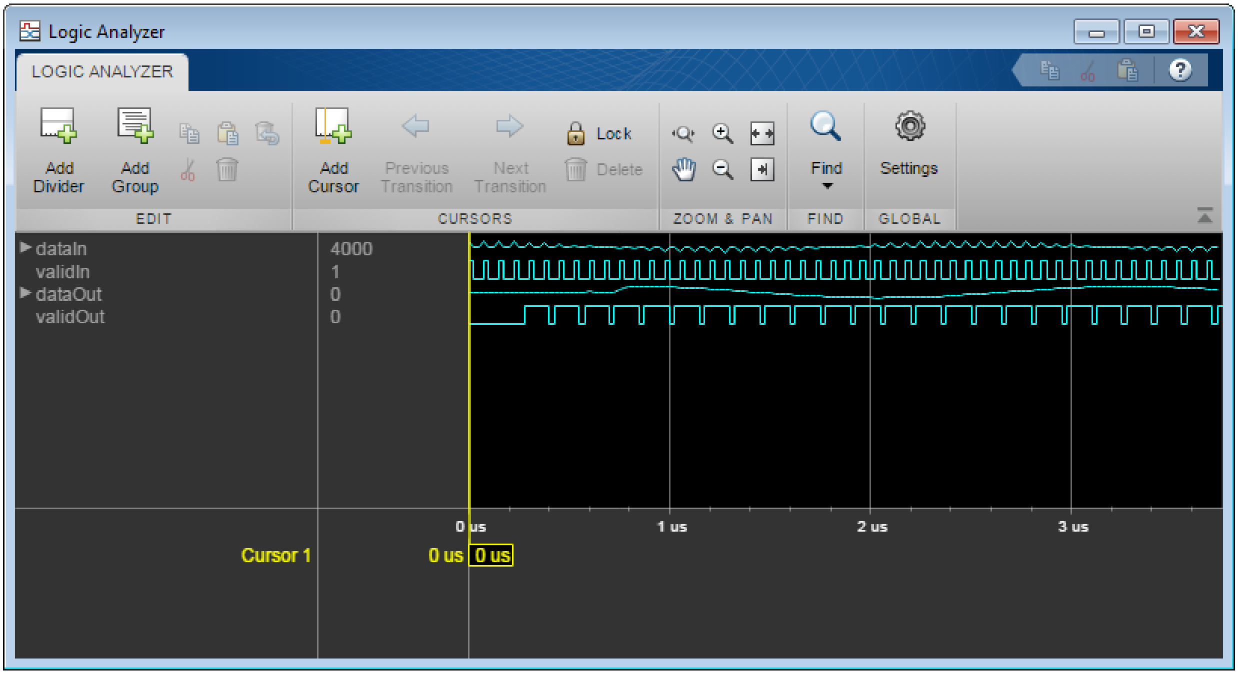
Create a **Logic Analyzer** to capture and view the input and output signals.

```
la = dsp.LogicAnalyzer('NumInputPorts',4,'SampleTime',1/Fs,'TimeSpan',numSteps/Fs);
modifyDisplayChannel(la,1,'Name','dataIn','Format','Analog','Height',8)
modifyDisplayChannel(la,2,'Name','validIn')
modifyDisplayChannel(la,3,'Name','dataOut','Format','Analog','Height',8)
modifyDisplayChannel(la,4,'Name','validOut')
```

Call the System object to perform the rate conversion and obtain each output sample. Call the Logic Analyzer to add each sample to the waveform display.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
for k = 1:numSteps
    [dataOut,validOut] = rateConverter(dataIn(k),validIn(k));
    la(dataIn(k),validIn(k),dataOut,validOut)
end
```



Control Input Rate When Upsampling

Convert a signal from 40 MHz to 100 MHz using the HDL FIR Rate Converter System object™. Use the optional `ready` output signal to avoid overrunning the object as the data is upsampled. The `ready` signal indicates the object can accept a new data sample on the next call to the object.

Define the sample rate and length of the input signal, and a fixed-point cosine waveform. Create a `SignalSource` object to provide data samples on demand.

```
Fs = 40e6;
Ns = 50;
t = (0:Ns-1).'/Fs;
x = fi(cos(2*pi*1.2e6*t),1,16,14);
inputSource = dsp.SignalSource(x);
```

Define the rate conversion parameters. Use an interpolation factor of 5 and a decimation factor of 2. Determine the number of calls to the object needed to convert *Ns* samples.

```
L = 5;
```

```
M = 2;
numSteps = floor(Ns*L/M);
```

Create the System object and configure it to perform rate conversion using the desired factors and an equiripple FIR filter. Enable the optional `ready` output port.

```
Numerator = firpm(70, [0,.15,.25,1], [1,1,0,0]);
rateConverter = dsp.HDLFIRRateConverter(L,M,Numerator,'ReadyPort',true);
```

Create a **Logic Analyzer** to capture and view the input and output signals.

```
la = dsp.LogicAnalyzer('NumInputPorts',5,'SampleTime',1/Fs,'TimeSpan',numSteps/Fs);
modifyDisplayChannel(la,1,'Name','dataIn','Format','Analog','Height',8)
modifyDisplayChannel(la,2,'Name','validIn')
modifyDisplayChannel(la,3,'Name','dataOut','Format','Analog','Height',8)
modifyDisplayChannel(la,4,'Name','validOut')
modifyDisplayChannel(la,5,'Name','ready')
```

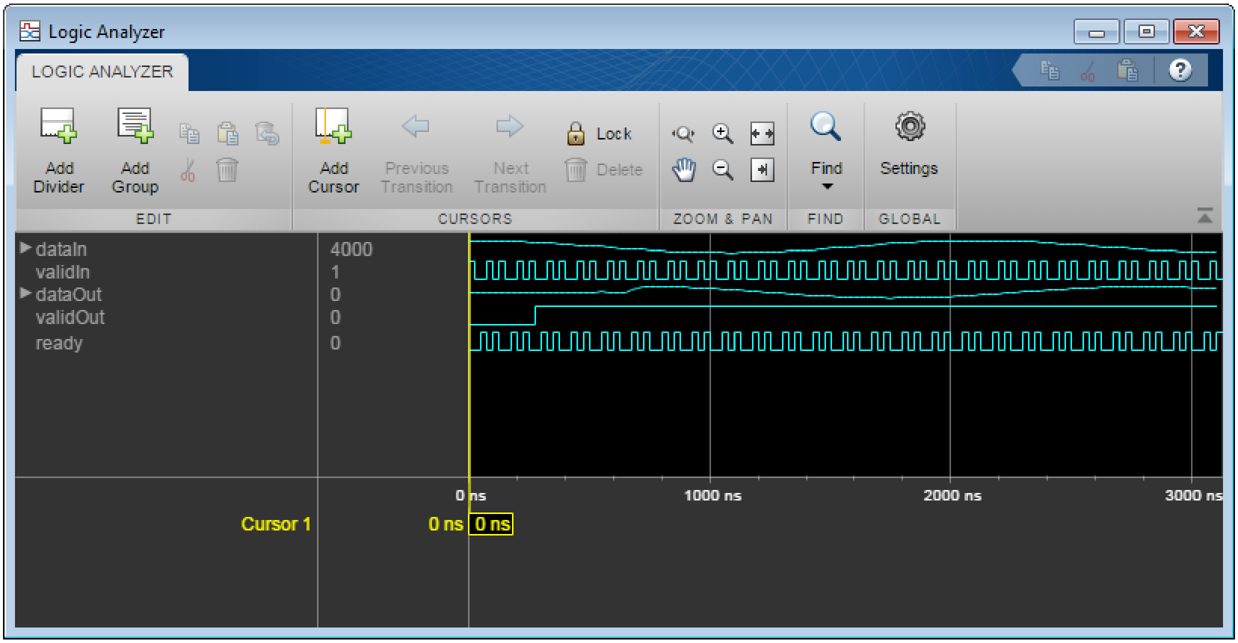
Initialize the `ready` signal. The object is always ready for input data on the first call.

```
ready = true;
```

Call the System object to perform the rate conversion and obtain each output sample. Apply a new input sample when the object indicates it is ready. Otherwise, set `validIn` to `false`.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
for k = 1:numSteps
    if ready
        dataIn = inputSource();
    end
    validIn = ready;
    [dataOut,validOut,ready] = rateConverter(dataIn,validIn);
    la(dataIn,validIn,dataOut,validOut,ready)
end
```



Control Output Rate When Upsampling

Convert a signal from 40 MHz to 100 MHz using the HDL FIR Rate Converter System object™. Use the optional `request` input signal to control the output data rate. When the object receives the `request` signal, it returns a new output data sample. This example models a clock rate of 200 MHz by requesting an output sample every second call to the object.

Define the sample rate and length of the input signal, and a fixed-point cosine waveform. Create a `SignalSource` object to provide data samples on demand.

```
Fs = 40e6;
Ns = 50;
t = (0:Ns-1).'/Fs;
x = fi(cos(2*pi*1.2e6*t),1,16,14);
inputSource = dsp.SignalSource(x);
```

Define the rate conversion parameters. Use an interpolation factor of 5 and a decimation factor of 2. Determine the number of calls to the object needed to convert *Ns* samples.

```
L = 5;
```

```
M = 2;
numSteps = floor(2*Ns*L/M);
```

Create the System object and configure it to perform rate conversion using the specified factors and an equiripple FIR filter. Enable the optional `ready` and `request` ports.

```
Numerator = firpm(70, [0,.15,.25,1], [1,1,0,0]);
rateConverter = dsp.HDLFIRRateConverter(L,M,Numerator,...
                                        'ReadyPort',true,...
                                        'RequestPort',true);
```

Create a **Logic Analyzer** to capture and view the input and output signals.

```
la = dsp.LogicAnalyzer('NumInputPorts',6,'SampleTime',1/Fs,'TimeSpan',numSteps/Fs);
modifyDisplayChannel(la,1,'Name','dataIn','Format','Analog','Height',8)
modifyDisplayChannel(la,2,'Name','validIn')
modifyDisplayChannel(la,3,'Name','request')
modifyDisplayChannel(la,4,'Name','dataOut','Format','Analog','Height',8)
modifyDisplayChannel(la,5,'Name','validOut')
modifyDisplayChannel(la,6,'Name','ready')
```

Generate a signal that requests a new output sample every second call to the object.

```
request = false(numSteps,1);
request(1:2:end) = true;
```

Initialize the `ready` signal. The object is always ready for input data on the first call.

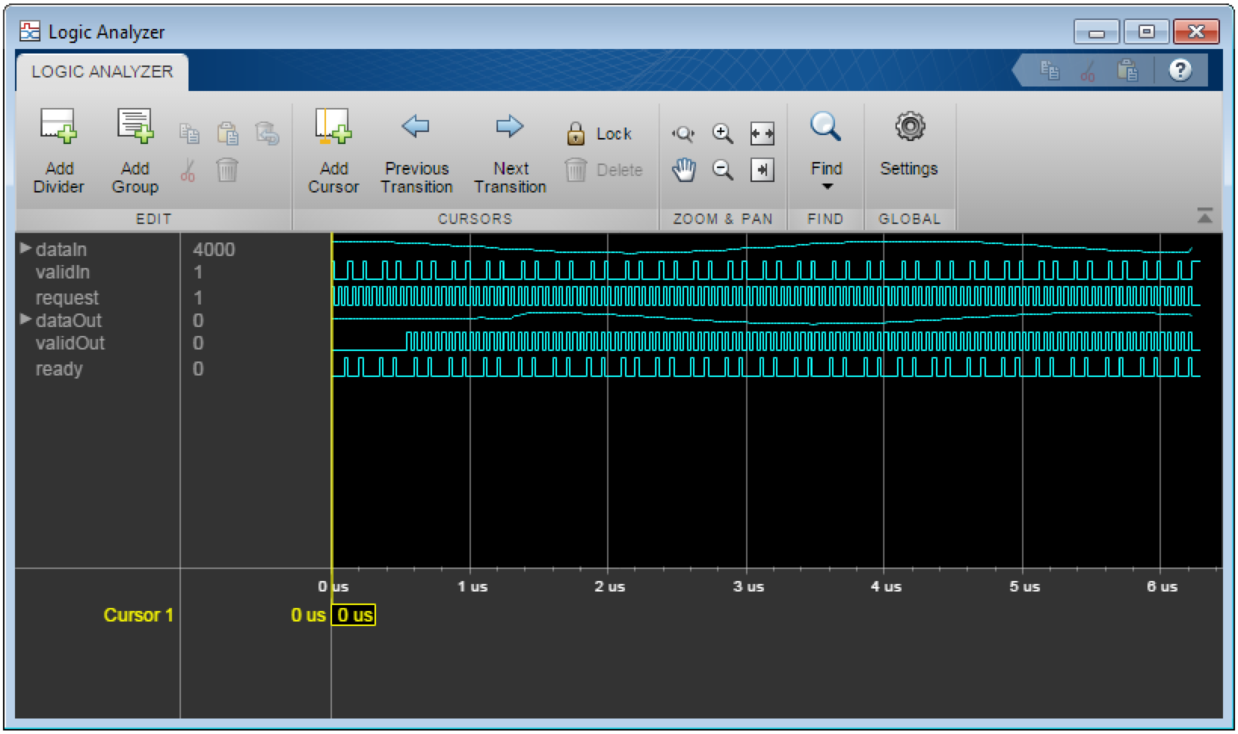
```
ready = true;
```

Call the System object to perform the rate conversion and obtain each output sample. Apply a new input sample when the object indicates it is ready. Otherwise, set `validIn` to `false`. The object returns valid output samples when `request` is set to `true`.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
for k = 1:numSteps
    if ready
        dataIn = inputSource();
    end
    validIn = ready;
    [dataOut,validOut,ready] = rateConverter(dataIn,validIn,request(k));
```

```
    la(dataIn,validIn,request(k),dataOut,validOut,ready)
end
```



Design for HDL Code Generation from HDL FIR Rate Converter

Create a rate conversion function targeted for HDL code generation, and a test bench to exercise it. The function converts a signal from 40 MHz to 100 MHz. To avoid overrunning the object, the test bench manually controls the input rate.

Define the sample rate and length of the input signal, and a fixed-point cosine waveform.

```
Fs = 40e6;
Ns = 50;
t = (0:Ns-1).'/Fs;
x = fi(cos(2*pi*1.2e6*t), 1, 16, 14);
```

Define the rate conversion parameters. Use an interpolation factor of 5 and a decimation factor of 2. Calculate how often the object can accept a new data sample.

```
L = 5;
M = 2;
stepsPerInput = ceil(L/M);
numSteps = stepsPerInput*Ns;
```

Generate `dataIn` and `validIn` based on how often the object can accept a new sample.

```
dataIn = zeros(numSteps,1,'like',x);
dataIn(1:stepsPerInput:end) = x;
validIn = false(numSteps,1);
validIn(1:stepsPerInput:end) = true;
```

Create a **Logic Analyzer** to capture and view the input and output signals.

```
la = dsp.LogicAnalyzer('NumInputPorts',4,'SampleTime',1/Fs,'TimeSpan',numSteps/Fs);
modifyDisplayChannel(la,1,'Name','dataIn','Format','Analog','Height',8)
modifyDisplayChannel(la,2,'Name','validIn')
modifyDisplayChannel(la,3,'Name','dataOut','Format','Analog','Height',8)
modifyDisplayChannel(la,4,'Name','validOut')
```

Write a function that instantiates and calls the System object.

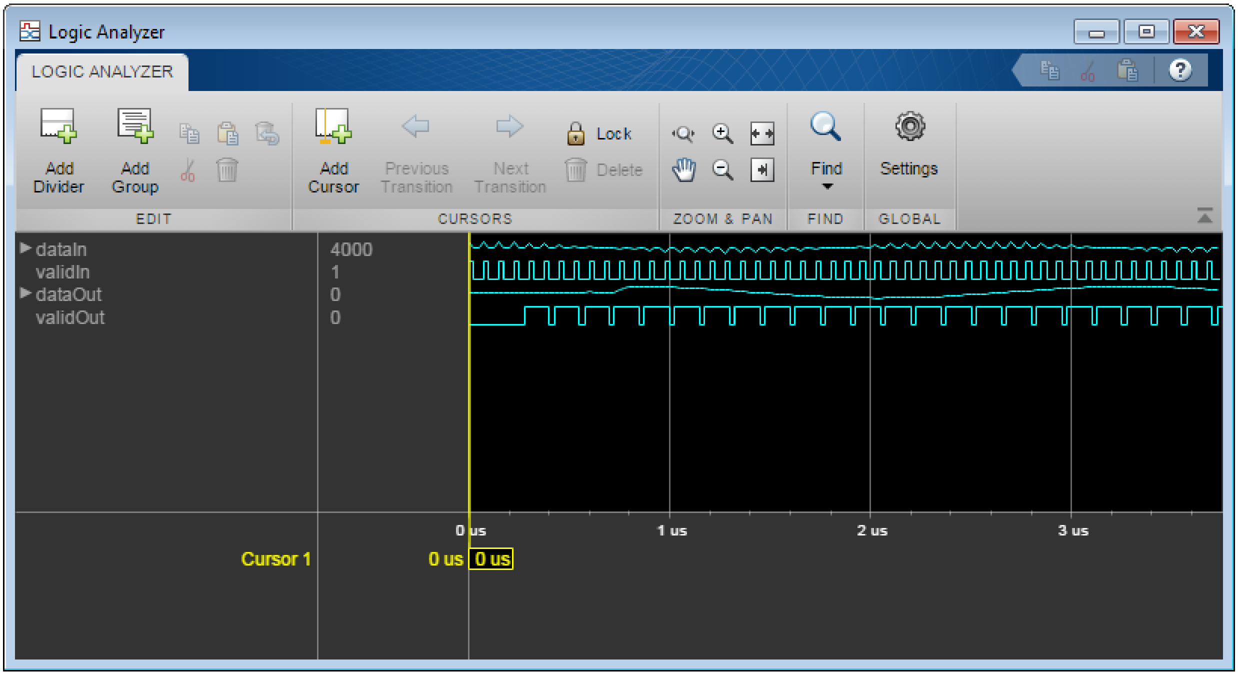
Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
function [dataOut,validOut] = HDLFIRRC5_2(dataIn,validIn)
%HDLFIRRC5_2
% Processes one sample of data using the dsp.HDLFIRRateConverter System
% object. dataIn is a fixed-point scalar value. validIn is a logical scalar value.
% You can generate HDL code from this function.

  persistent firrc5_2;
  if isempty(firrc5_2)
    Numerator = firpm(70,[0,.15,.25,1],[1,1,0,0]);
    firrc5_2 = dsp.HDLFIRRateConverter(5,2,Numerator);
  end
  [dataOut,validOut] = firrc5_2(dataIn,validIn);
end
```

Resample the signal by calling the function for each data sample.

```
for k = 1:numSteps
```

```
    [dataOut,validOut] = HDLFIRRC5_2(dataIn(k),validIn(k));
    la(dataIn(k),validIn(k),dataOut,validOut)
end
```



Algorithm

This object implements the algorithms described on the `FIR Rate Conversion HDL Optimized` block reference page.

See Also

See Also

dsp.FIRRateConverter | `FIR Rate Conversion HDL Optimized`

Introduced in R2015b

step

System object: dsp.HDLFIRRateConverter
Package: dsp

Resample and filter input sample

Syntax

```
[dataOut,validOut] = step(HDLFIRRC,dataIn,validIn)
[dataOut,ready,validOut] = step(HDLFIRRC,dataIn,validIn)
[dataOut,ready,validOut] = step(HDLFIRRC,dataIn,validIn,request)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[dataOut,validOut] = step(HDLFIRRC,dataIn,validIn)` resamples `dataIn` according to the `InterpolationFactor` (*L*) and `DecimationFactor` (*M*) properties. To avoid dropped samples when using this syntax, apply new valid input samples, with `validIn` set to `true`, only every `ceil(L/M)` calls to the `step` method. The object sets `validOut` to `true` when `dataOut` is a new valid sample.

`[dataOut,ready,validOut] = step(HDLFIRRC,dataIn,validIn)` resamples the input data and returns `ready` to indicate whether the object can accept a new sample on the next call to the `step` method. This syntax applies when you set the `ReadyPort` property to `true`.

`[dataOut,ready,validOut] = step(HDLFIRRC,dataIn,validIn,request)` resamples the input data, indicates whether the object can accept a new sample, and, if `request` is `true`, returns the next available sample . This syntax applies when you set the `RequestPort` property to `true`.

You can connect the `ready` output of a downstream object to the `request` input of an upstream object.

Note: The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Input Arguments

`dataIn` must be a scalar value or a row vector in which each element represents an independent channel. `validIn` and `validOut` are logical scalars that indicate the validity of the input and output signals respectively.

HDLFIRRC

`dsp.HDLFIRRateConverter` System object.

dataIn

Data sample, specified as a scalar, or as a row vector where each element represents an independent channel.

The data can be real or complex. Supported data types:

- `fixdt()`
- `uint`8/16/32, `int`8/16/32
- `double` and `single` are allowed for simulation but not for HDL code generation.

validIn

Qualification of the `dataIn` value, specified as a `logical` scalar.

When `validIn` is `true`, the object captures the value on `dataIn`. You can apply a valid data sample every `ceil(L/M)` calls to `step`. You can use the optional `ready` output signal to indicate when the object can accept a new sample.

request

Request for a new output sample, specified as a logical scalar.

When request is true, and a new output data sample is available, the object returns the sample with validOut set to true. If there is no sample available, the object returns validOut set to false. The object accepts this argument when you set the RequestPort property to true.

Output Arguments

dataOut

Resampled and filtered data sample, returned as a scalar, or as a vector in which each element represents an independent channel.

Supported data types:

- fixdt()
- uint8/16/32, int8/16/32
- double and single are allowed for simulation but not for HDL code generation.

validOut

Qualification of the dataOut value, returned as a logical scalar.

When validOut is true, the data output is valid. When validOut is false, the data output is not valid.

ready

Indication that the object is ready for a new input sample, returned as a logical scalar.

The object returns ready = true to indicate that the object can accept a new input data sample on the next call to step. The object returns this additional output when you set the ReadyPort property to true.

dsp.FrequencyDomainAdaptiveFilter System object

Package: dsp

Frequency Domain Adaptive filter

Description

The `dsp.FrequencyDomainAdaptiveFilter` computes output, error, and coefficients using a frequency domain FIR adaptive filter.

To implement the adaptive FIR filter object:

1. Define and set up your adaptive FIR filter object. See "Construction" on page 4-853.
2. Call step to implement the filter according to the properties of `dsp.FrequencyDomainAdaptiveFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`fdaf = dsp.FrequencyDomainAdaptiveFilter` returns a frequency domain FIR adaptive filter System object, `fdaf`. This System object is used to compute the filtered output and the filter error for a given input and desired signal.

`fdaf = dsp.FrequencyDomainAdaptiveFilter('`*`PropertyName`*`',` *`PropertyValue`*`,...)` returns an `FrequencyDomainAdaptiveFilter` System object, `fdaf`, with each specified property set to the specified value.

`fdaf =`
`dsp.FrequencyDomainAdaptiveFilter(LEN,'`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an `FrequencyDomainAdaptiveFilter` System object, `fdaf`, with the Length property set to `LEN` and other specified properties set to the specified values.

Properties

`Method`

Method to calculate filter coefficients

Specify the method used to calculate filter coefficients as one of `'Constrained FDAF'` |`'Unconstrained FDAF'`. The default is `'Constrained FDAF'`. For algorithms on how to implement this filter and its three methods, refer to [1]. This property is nontunable.

`Length`

Length of filter coefficients vector

Specify the length of the FIR filter coefficients vector as a positive integer value. This property is nontunable.

The default value is 32.

`BlockLength`

Block length for coefficient updates

Specify the block length of the coefficients updates as a positive integer value. The length of the input vectors must be divisible by the `BlockLength` property value. For faster execution, the sum of the length property value and the `BlockLength` property value should be a power of two. The default value is the value of `Length`. This property is nontunable.

`StepSize`

Adaptation step size

Specify the adaptation step size factor as a positive numeric scalar less than or equal to `1`. Setting the `StepSize` property equal to `1` provides the fastest convergence during adaptation. The default is `1`. This property is tunable.

`LeakageFactor`

Adaptation leakage factor

Specify the leakage factor used in leaky adaptive filter as a scalar numeric value between 0 and 1, both inclusive. When the value is less than 1, the System object implements a leaky adaptive algorithm. The default is `1`, providing no leakage in the adapting method. This property is tunable.

AveragingFactor

Averaging factor of energy estimator

Specify the averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates as a scalar positive numeric value less than or equal to 1. The default value is `1`. This property is tunable.

Offset

Offset for normalization terms

Specify the offset for the normalization terms in the coefficient updates as a scalar nonnegative numeric value. Use this property to avoid divide by zero or divide by very small numbers. This situation occurs if any of the FFT input signal powers becomes very small. The default value is `0`. This property is tunable.

InitialPower

Initial FFT input signal power

Specify the initial common value of all of the FFT input signal powers as a scalar positive numeric value. The default is `1`. This property is tunable.

InitialCoefficients

Time-domain initial coefficients of the filter

Specify the initial time-domain coefficients of the adaptive filter as a scalar or a vector of length equal to the `Length` property value. The adaptive filter object uses these coefficients to compute the initial frequency-domain filter coefficients. The default is `0`. This property is tunable.

LockCoefficients

Locked status of coefficient updates

Specify whether to lock the filter coefficient values. By default, the value of this property is `false`, and the object continuously updates the filter coefficients. If this property is set to `true`, the filter coefficients do not update and their values remain the same.

This property is tunable.

FFTCoefficients

Current FFT coefficients of the filter

This property stores the current discrete Fourier transform of the filter coefficients as a row vector. The length of this vector is equal to the sum of the `Length` value and the `BlockLength` value. This property is initialized to the FFT values of the `InitialCoefficients` property. To get the discrete Fourier transform of the filter coefficients, call the object, and access the `FFTCoefficients` property of the object.

Methods

reset	Reset filter states for Frequency Domain Adaptive filter
step	Apply Frequency Domain Adaptive filter to input

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

QPSK Adaptive Equalization With FIR Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Generate QPSK and noise signals, filter to obtain the received signal, and delay the QPSK signal to obtain the desired signal.

```
D = 16;
```

```
b  = exp(1i*pi/4)*[-0.7 1];
a  = [1 -0.7];
ntr= 1024;
s  = sign(randn(1,ntr+D))+1i*sign(randn(1,ntr+D));
n  = 0.1*(randn(1,ntr+D) + 1i*randn(1,ntr+D));
r  = filter(b,a,s) + n;
x  = r(1+D:ntr+D);
d  = s(1:ntr);
```

Use the Frequency Domain Adaptive Filter to compute the filtered output and the filter error for the input and desired signal. To get the discrete Fourier transform of the filter coefficients, call the `fdaf` object, and access the FFTCoefficients property of this object.

```
mu  = 0.1;
fdaf = dsp.FrequencyDomainAdaptiveFilter('Length',32,'StepSize',mu);
[y,e] = fdaf(x,d);
fftCoeffs = fdaf.FFTCoefficients
```

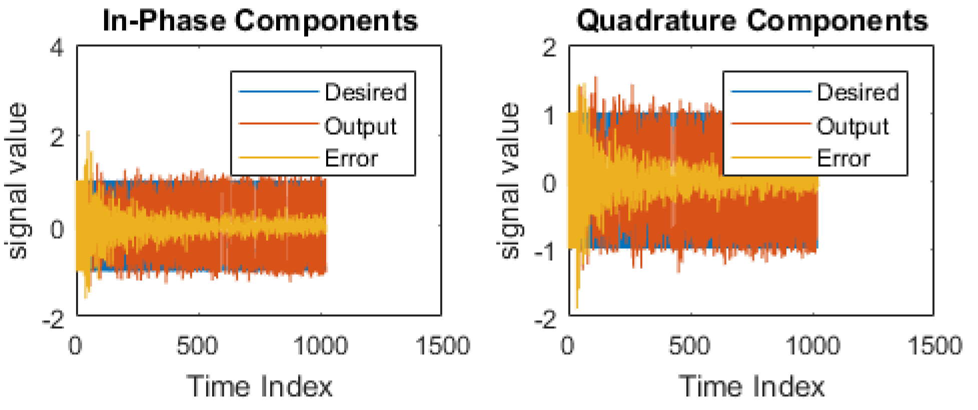
```
fftCoeffs =

  Columns 1 through 4

   0.6802 - 0.6847i  -0.2485 - 0.9427i  -0.9675 - 0.2123i  -0.5605 + 0.8002i

  Columns 5 through 8

   0.5748 + 0.7593i   0.8541 - 0.3917i  -0.2526 - 0.9022i  -0.9298 + 0.1255i

  Columns 9 through 12

   0.0181 + 0.9366i   0.9207 + 0.0511i   0.1063 - 0.8972i  -0.8919 - 0.1829i

  Columns 13 through 16

  -0.2668 + 0.9113i   0.9215 + 0.3186i   0.3417 - 0.8859i  -0.8285 - 0.3760i

  Columns 17 through 20

  -0.4317 + 0.8200i   0.8741 + 0.4765i   0.4874 - 0.9075i  -0.8517 - 0.4774i

  Columns 21 through 24

  -0.4709 + 0.7632i   0.7468 + 0.4833i   0.5193 - 0.7995i  -0.8218 - 0.5649i
```

```
  Columns 25 through 28

  -0.5908 + 0.7768i   0.7316 + 0.5866i   0.5806 - 0.7270i  -0.7148 - 0.5998i

  Columns 29 through 32

  -0.6287 + 0.6702i   0.6575 + 0.6379i   0.6332 - 0.7153i  -0.7659 - 0.6424i

  Columns 33 through 36

  -0.6678 + 0.7294i   0.6536 + 0.6891i   0.7006 - 0.6333i  -0.6594 - 0.7117i

  Columns 37 through 40

  -0.7207 + 0.6517i   0.6031 + 0.7239i   0.7362 - 0.5776i  -0.5869 - 0.7682i

  Columns 41 through 44

  -0.7975 + 0.5789i   0.5449 + 0.7992i   0.7909 - 0.5343i  -0.5512 - 0.8070i

  Columns 45 through 48

  -0.8392 + 0.5338i   0.4605 + 0.8493i   0.8358 - 0.3921i  -0.3751 - 0.8388i

  Columns 49 through 52

  -0.8739 + 0.3785i   0.3625 + 0.9048i   0.8986 - 0.3405i  -0.3285 - 0.8728i

  Columns 53 through 56

  -0.8692 + 0.2947i   0.2106 + 0.9001i   0.9391 - 0.1095i  -0.0381 - 0.9526i

  Columns 57 through 60

  -0.9339 - 0.0169i  -0.1066 + 0.9073i   0.8955 + 0.2469i   0.4160 - 0.8776i

  Columns 61 through 64

  -0.7874 - 0.6033i  -0.8034 + 0.5626i   0.1949 + 0.9546i   0.9394 + 0.2570i
```

Plot the In-Phase and the Quadrature components of the desired, output, and the error signals.

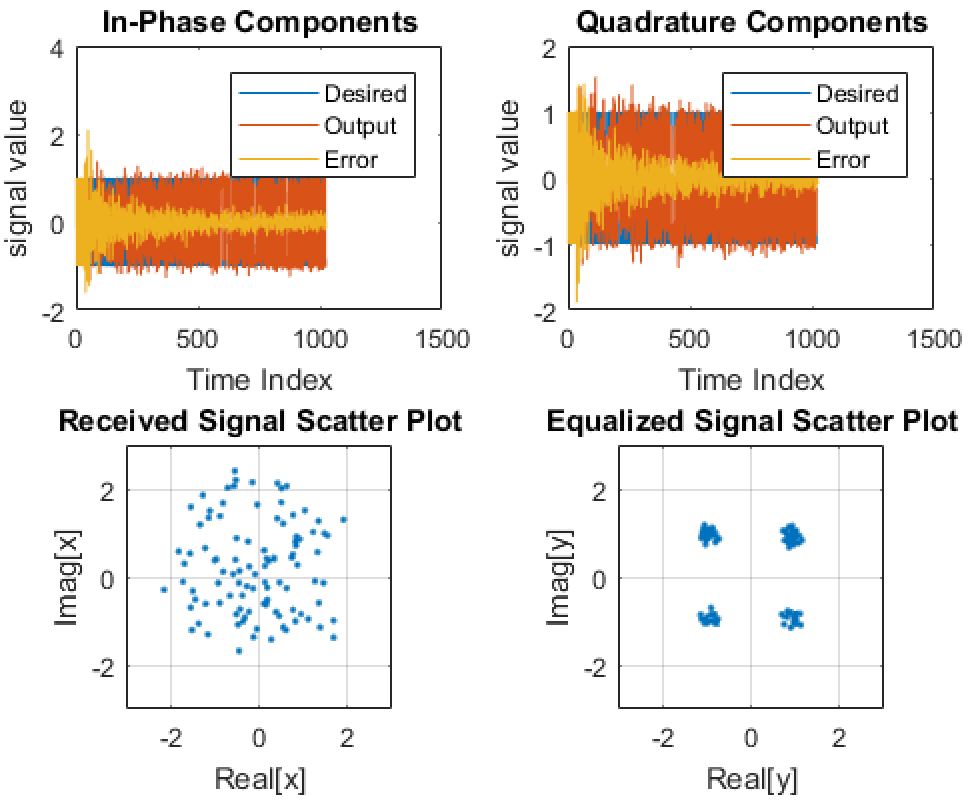
```
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
```

```
legend('Desired','Output','Error'); title('In-Phase Components');
xlabel('Time Index'); ylabel('signal value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
legend('Desired','Output','Error'); title('Quadrature Components');
xlabel('Time Index'); ylabel('signal value');
```



Plot the received and equalized signals' scatter plots.

```
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```



References

[1] Shynk, J.J.“Frequency-Domain and Multirate Adaptive Filtering.” *IEEE Signal Processing Magazine,* Vol. 9, No. 1, pp. 14–37, Jan. 1992.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder).
- When the sum of `BlockLength` and `Length` is not a power of two, the executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- When the sum of `BlockLength` and `Length` is a power of two, you can generate standalone C and C++ code from this System object.

See Also

See Also

System Objects
dsp.AffineProjectionFilter | dsp.AdaptiveLatticeFilter | dsp.FilteredXLMSFilter | dsp.FIRFilter | dsp.LMSFilter | dsp.RLSFilter

Introduced in R2013b

reset

System object: dsp.FrequencyDomainAdaptiveFilter
Package: dsp

Reset filter states for Frequency Domain Adaptive filter

Syntax

```
reset(sysObj)
```

Description

`reset(sysObj)` resets the internal states of the System object, `sysObj`, to their initial values. The reset method is always a no-op for unlocked System objects, as the states may not be allocated when the object is not locked.

step

System object: dsp.FrequencyDomainAdaptiveFilter
Package: dsp

Apply Frequency Domain Adaptive filter to input

Syntax

```
[y,err] = step(fdaf,x,d)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[y,err] = step(fdaf,x,d)` filters the input *x*, using *d* as the desired signal, and returns the filtered output in *y* and the filter error in *err*. The System object estimates the filter weights needed to minimize the error between the output signal and the desired signal. You can obtain the FFT of these filter weights by accessing the `FFTCoefficients` property after calling the `step` method by `fdaf.FFTCoefficients`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.HDLComplexToMagnitudeAngle System object

Package: dsp

Compute magnitude and phase angle of complex signal—optimized for HDL code generation

Description

The HDL Complex To Magnitude Angle System object computes the magnitude and/or phase angle of a complex signal. It provides hardware-friendly control signal arguments for the step method. The object uses a pipelined Coordinate Rotation Digital Computer (CORDIC) algorithm to achieve an efficient HDL implementation.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`HCMA = dsp.HDLComplexToMagnitudeAngle()` returns a System object, HCMA, that computes the magnitude and phase angle of a complex input.

`HCMA = dsp.HDLComplexToMagnitudeAngle(Name,Value)` returns a System object, HCMA, with additional options specified by one or more `Name,Value` pair arguments.

Properties

`OutputValue`

Specifies which outputs the `step` method returns. You can set this property to `Magnitude`, `Angle`, or `Magnitude and angle`. The default is `Magnitude and angle`.

AngleFormat

Specifies the format of the Angle output of the `step` method. You can set this property to `Normalized` or `Radians`. `Normalized` is a fixed-point format that normalizes the angles in the range [–1,1]. `Radians` returns fixed-point values between π and −π. The default is `Normalized`.

ScaleOutput

Scales output by the inverse of the CORDIC gain factor when this property is `true`. The default is `true`.

Note: If you turn off output scaling, and apply the CORDIC gain elsewhere in your design, you must exclude the π/4 term. The quadrant mapping algorithm replaces the first CORDIC iteration by mapping inputs onto the angle range [0,π/4]. Therefore, the initial rotation does not contribute a gain term.

NumIterationsSource

Specifies the source of the `NumIterations` property for the CORDIC algorithm. You can set `NumIterations` to `Property` or `Auto`. `Property` uses the number of iterations from `NumIterations`. `Auto` sets the number of iterations to input vector word length – 1. If the input is `double` or `single`, `Auto` sets the number of iterations to 16. The default is `Auto`.

NumIterations

Specifies the number of CORDIC iterations the object executes. This value is used when `NumIterationsSource` is set to `Property`. The number of iterations must be less than or equal to the input data word length – 1.

Methods

step	Process complex input to produce magnitude and/or phase angle

Common to All System Objects	
`clone`	Create System object with same property values

Common to All System Objects	
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

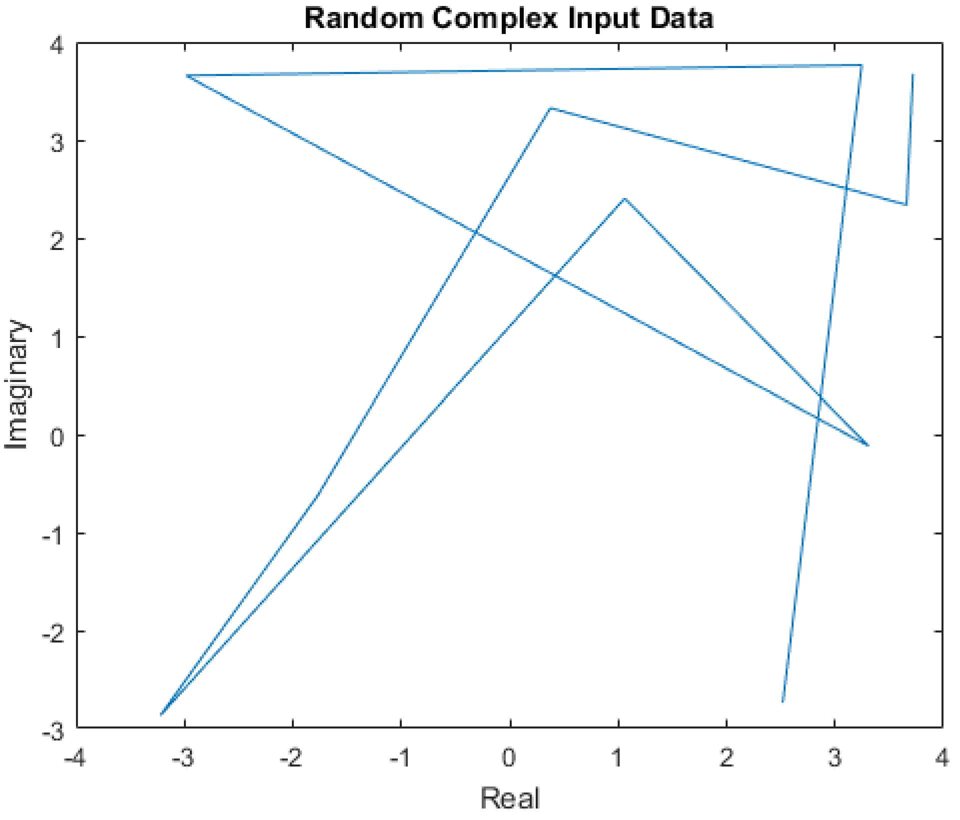
Examples

Convert Complex Data to Magnitude-Angle Data

This example shows how to use the `dsp.HDLComplextoMagnitudeAngle` object to compute magnitude and angle of a complex signal. The object uses a CORDIC algorithm for an efficient hardware implementation.

Choose word lengths and create random complex input data. Than, convert the input to fixed-point.

```
a = -4;
b = 4;
inputWL = 16;
inputFL = 12;
numSamples = 10;
reData = ((b-a).*rand(numSamples,1)+a);
imData = ((b-a).*rand(numSamples,1)+a);
dataIn = (fi(reData+imData*1i,1,inputWL,inputFL));
figure
plot(dataIn)
title('Random Complex Input Data')
xlabel('Real')
ylabel('Imaginary')
```



Write a function that creates and calls the System object™. You can generate HDL from this function.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
function [mag,angle,validOut] = Complex2MagAngle(yIn,validIn)
%Complex2MagAngle
% Converts one sample of complex data to magnitude and angle data.
% yIn is a fixed-point complex number.
% validIn is a logical scalar value.
% You can generate HDL code from this function.
```

```
  persistent cma;
  if isempty(cma)
    cma = dsp.HDLComplexToMagnitudeAngle('AngleFormat','Radians')
  end
  [mag,angle,validOut] = cma(yIn,validIn);
end
```

The object takes some latency to computer the answer for each input sample, depending on the number of CORDIC iterations. When you set `NumIterationsSource` to 'Auto', `NumIterations=inputWL-1`. The latency is `NumIterations+2`, so `latency=inputWL+1`.

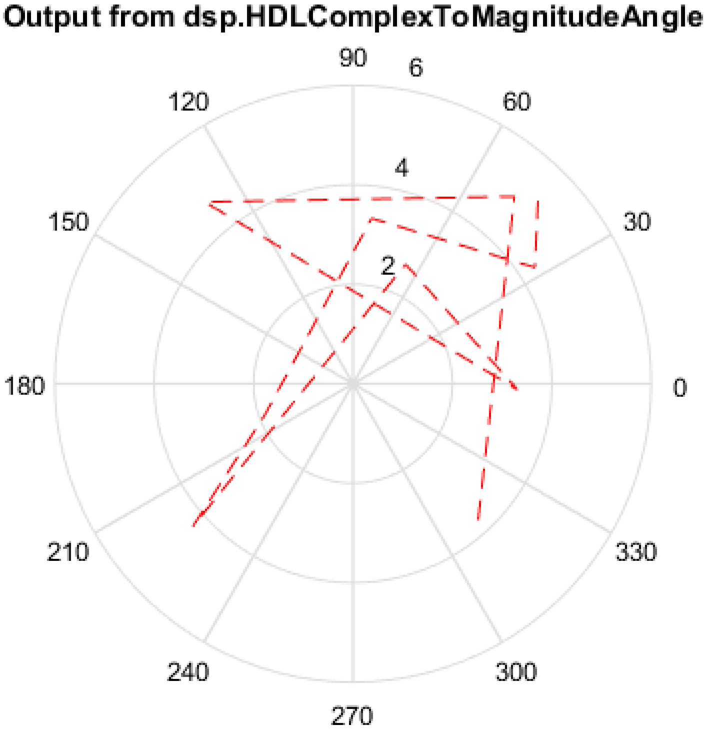
```
latency = inputWL+1;
mag = zeros(1,numSamples+latency);
ang = zeros(1,numSamples+latency);
validOut = false(1,numSamples+latency);
```

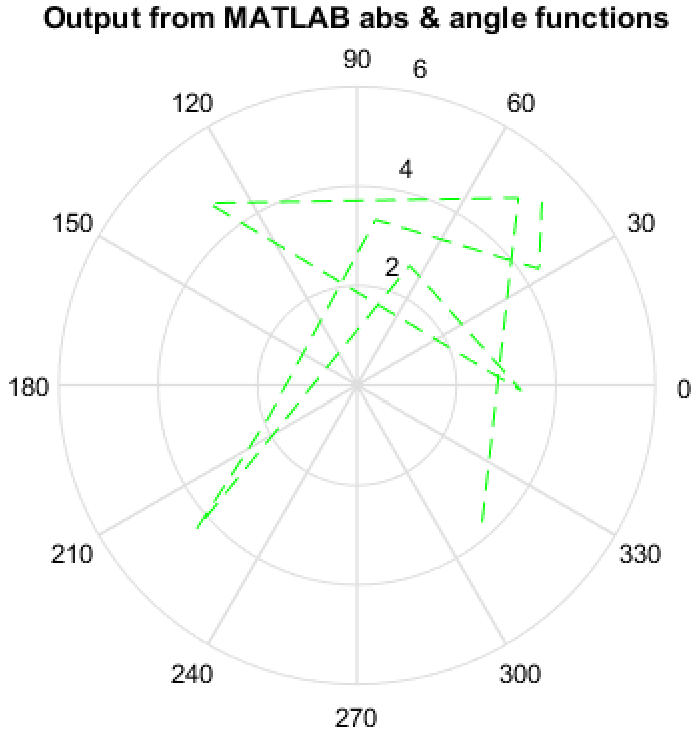
Call the function to convert each sample. After you apply all input samples, continue calling the function with invalid input to flush remaining output samples.

```
for ii = 1:1:numSamples
   [mag(ii),ang(ii),validOut] = Complex2MagAngle(dataIn(ii),true);
end
for ii = (numSamples+1):1:(numSamples+latency)
   [mag(ii),ang(ii),validOut(ii)] = Complex2MagAngle(fi(0+0*1i,1,inputWL,inputFL),false
end
% Strip non-valid out
mag = mag(validOut == 1);
ang = ang(validOut == 1);
figure
polar(ang,mag,'--r')   % Red is output from HDL-Optimized object
title('Output from dsp.HDLComplexToMagnitudeAngle')
magD = abs(dataIn);
angD = angle(dataIn);
figure
polar(angD,magD,'--g') % Green is output from MATLAB abs & angle
title('Output from MATLAB abs & angle functions')
```

```
cma =
```

```
dsp.HDLComplexToMagnitudeAngle with properties:

  NumIterationsSource: 'Auto'
         OutputFormat: 'Magnitude and angle'
          AngleFormat: 'Radians'
          ScaleOutput: true
```





Algorithm

This object implements the algorithm described on the `Complex to Magnitude-Angle HDL Optimized` block reference page.

Delay

The latency is `NumIterations` + 2 cycles from input to output. Each call to the step method models one clock cycle.

When you set `Number of Iterations Source` to `Auto`, the number of iterations is input word length − 1, and the latency is input word length + 1. If the input is `double` or `single` type, the number of iterations is 16, and the latency is 18.

Performance

When generated HDL code for the default configuration, with output scaling disabled and `fixdt(1,16,12)` input, is synthesized into a Xilinx Virtex–6 (XC6VLX240T-1FFG1156) FPGA, the design achieves 260 MHz clock frequency. It uses the following resources.

Resource	Uses
LUT	882
FFS	792
Xilinx LogiCORE DSP48	0
Block RAM (16K)	0

Performance of the synthesized HDL code varies depending on your target and synthesis options.

See Also

See Also

cordicangle (Fixed-Point Designer) | cordiccart2pol (Fixed-Point Designer) | cordicabs (Fixed-Point Designer) | angle (MATLAB) | `Complex to Magnitude-Angle HDL Optimized`

Introduced in R2014b

step

System object: dsp.HDLComplexToMagnitudeAngle
Package: dsp

Process complex input to produce magnitude and/or phase angle

Syntax

```
[Mag,Angle,validOut] = step(H,X,validIn)
[Mag,validOut] = step(H,X,validIn)
[Angle,validOut] = step(H,X,validIn)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[Mag,Angle,validOut] = step(H,X,validIn)` converts complex scalar X into component magnitude and angle. `validIn` and `validOut` are logical scalars that indicate the validity of the input and output signals respectively. If `validOut` is high, the output is valid.

`[Mag,validOut] = step(H,X,validIn)` converts complex scalar X into component magnitude. Use this syntax when you set `OutputValue` to `'Magnitude'`. `validOut` is a logical scalar that indicates the validity of output signal. If `validOut` is high, the output is valid.

`[Angle,validOut] = step(H,X,validIn)` converts complex scalar X into component magnitude. Use this syntax when you set `OutputValue` to `'Angle'`. `validOut` is a logical scalar that indicates the validity of output signal. If `validOut` is high, the output is valid.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.HDLFFT System object

Package: dsp

Fast Fourier transform—optimized for HDL code generation

Description

The HDL FFT System object implements a pipelined Radix 2^2 FFT algorithm which provides hardware speed and area optimization for streaming data applications. The object accepts scalar or vector real or complex input, provides hardware-friendly control signals, and has optional output frame control signals. You can achieve giga-sample-per-second (GSPS) throughput using vector input.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`HFFT = dsp.HDLFFT` returns an HDL FFT System object, `HFFT`, that performs a fast Fourier transform.

`HFFT = dsp.HDLFFT(Name,Value)` returns an HDL FFT System object, `HFFT`, with additional options specified by one or more `Name,Value` pair arguments.

Properties

`Architecture`

`'Streaming Radix 2^2'` (default) — An efficient architecture that has lower latency and uses fewer resources than the previous versions.

`'Streaming Radix 2 (this choice will be removed -see release notes).'` — Versions before R2016a use this architecture. Use only for backward-compatibility purposes.

`ComplexMultiplication`

Select the HDL implementation of complex multipliers. Choose to implement each multiplier with `'Use 4 multipliers and 2 adders'` or `'Use 3 multipliers and 5 adders'`. Which option is faster or smaller depends on your synthesis tool and target device. This option applies only if you use the Radix 2^2 architecture.

`BitReversedOutput`

Select the order of the output data.

- `true` (default) — The output channel elements are bit reversed relative to the input order.
- `false` — The output channel elements are in linear order.

The FFT algorithm calculates output in the reverse order to the input, and performs an extra reversal operation when required to provide output in the same order as the input. For vector data, input and output data must be in opposite orders so choose only one of `BitReversedOutput` or `BitReversedInput`. For more information on ordering of the output, see "Linear and Bit-Reversed Output Order".

`BitReversedInput`

Select the expected order of the input data.

- `true` — The input channel elements are in bit-reversed order.
- `false` (default) — The input channel elements are in linear order.

The FFT algorithm calculates output in the reverse order to the input, and performs an extra reversal operation when required to provide output in the same order as the input. For vector data, input and output data must be in opposite orders so choose only one of `BitReversedOutput` or `BitReversedInput`. For more information on ordering of the output, see "Linear and Bit-Reversed Output Order".

`Normalize`

Enable output scaling.

- `true` — The object implements an overall 1/N scale factor by scaling the result of each pipeline stage by 2. This adjustment keeps the output of the IFFT in the same amplitude range as its input.

- `false` (default) — The object avoids overflow by increasing the word length by one bit at each stage.

FFTLength

Number of data points used for one FFT calculation. The default value is 1024. The FFT length must be a power of 2 between 2^3 and 2^{16} for HDL code generation.

ResetInputPort

When you set `ResetInputPort` to `true`, the `step` method expects an additional argument, `reset`. When `reset` is `true`, the object stops calculation and clears all internal state. The default value of `ResetInputPort` is `false`.

StartOutputPort

If you set `StartOutputPort` to `true`, the `step` method returns an additional output signal that is `true` on the first cycle of each valid output frame. The default value of `StartOutputPort` is `false`.

EndOutputPort

If you set `EndOutputPort` to `true`, the step method returns an additional output signal that is `true` on the last cycle of each valid output frame. The default value of `EndOutputPort` is `false`.

RoundingMethod

Rounding mode used for fixed-point operations. The FFT object uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input is single or double type. The default `RoundingMethod` is `'Floor'`. Each stage rounds after the twiddle factor multiplication but before the butterflies.

Methods

getLatency	Latency of FFT operation
step	Process inputs using HDL optimized FFT (fast Fourier transform)

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Create an FFT for HDL Generation

Create the specifications and input signal.

```
N = 128;
Fs = 40;
t = (0:N-1)'/Fs;
x = sin(2*pi*15*t) + 0.75*cos(2*pi*10*t);
y = x + .25*randn(size(x));
y_fixed = sfi(y,32,24);
```

Write a function that creates and calls the System object™. You can generate HDL from this function.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
function [yOut,validOut] = HDLFFT128(yIn,validIn)
%HDLFFT128
% Processes one sample of FFT data using the dsp.HDLFFT System object(TM)
% yIn is a fixed-point scalar or column vector.
% validIn is a logical scalar value.
% You can generate HDL code from this function.

  persistent fft128;
  if isempty(fft128)
    fft128 = dsp.HDLFFT('FFTLength',128)
  end
```

```
  [yOut,validOut] = fft128(yIn,validIn);
end
```

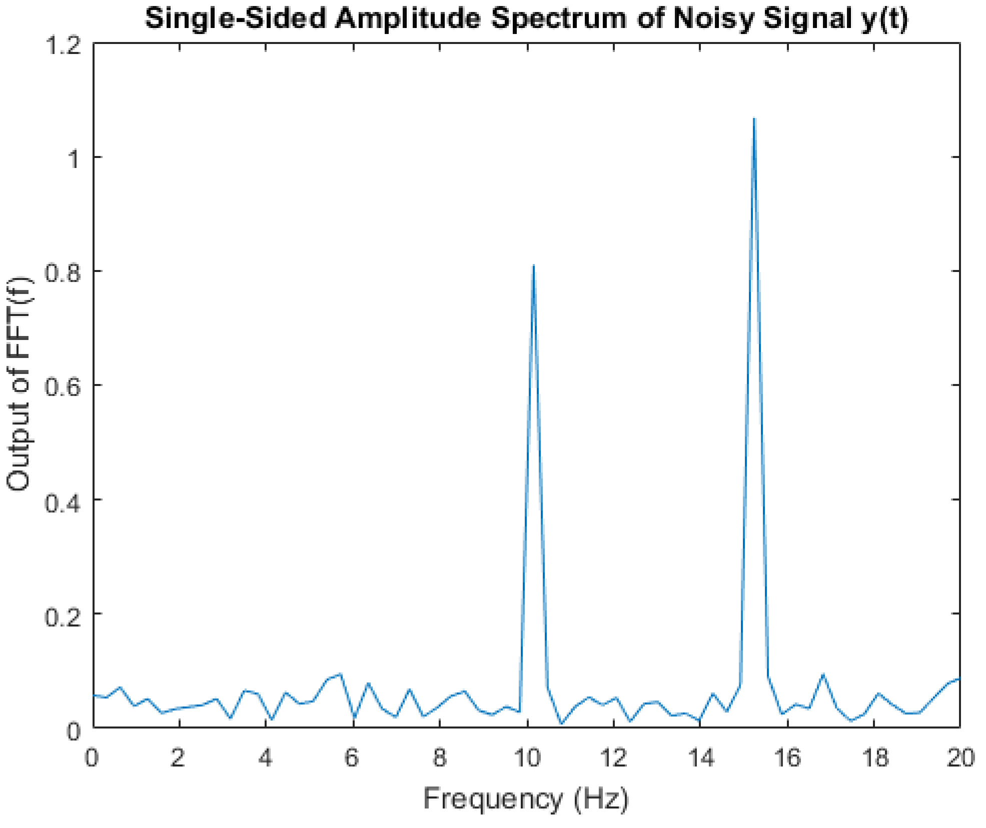
Compute the FFT by calling the function for each data sample.

```
Yf = zeros(1,3*N);
validOut = false(1,3*N);
for loop = 1:1:3*N
    if (mod(loop, N) == 0)
        i = N;
    else
        i = mod(loop, N);
    end
    [Yf(loop),validOut(loop)] = HDLFFT128(complex(y_fixed(i)),(loop <= N));
end
```

```
fft128 =

  dsp.HDLFFT with properties:

              FFTLength: 128
           Architecture: 'Streaming Radix 2^2'
    ComplexMultiplication: 'Use 4 multipliers and 2 adders'
       BitReversedOutput: true
        BitReversedInput: false
               Normalize: false

  Use get to show all properties
```

Discard invalid data samples. Then plot the frequency channel results from the FFT.

```
Yf = Yf(validOut == 1);
Yr =  bitrevorder(Yf);
plot(Fs/2*linspace(0,1,N/2), 2*abs(Yr(1:N/2)/N))
title('Single-Sided Amplitude Spectrum of Noisy Signal y(t)')
xlabel('Frequency (Hz)')
ylabel('Output of FFT(f)')
```



Create a Vector-Input FFT for HDL Generation

Create specifications and input signal. This example uses a 128-point FFT and computes the transform over 16 samples at a time.

```
N = 128;
V = 16;
Fs = 40;
t = (0:N-1)'/Fs;
x = sin(2*pi*15*t) + 0.75*cos(2*pi*10*t);
y = x + .25*randn(size(x));
y_fixed = sfi(y,32,24);
y_vect = reshape(y_fixed,V,N/V);
```

Write a function that instantiates and calls the System object™. The function does not need to know the vector size. The object saves the size of the input signal the first time you call it.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
function [yOut,validOut] = HDLFFT128V16(yIn,validIn)
%HDLFFT128V16
% Processes 16-sample vectors of FFT data
% yIn is a fixed-point column vector.
% validIn is a logical scalar value.
% You can generate HDL code from this function.

  persistent fft128v16;
  if isempty(fft128v16)
    fft128v16 = dsp.HDLFFT('FFTLength',128)
  end
  [yOut,validOut] = fft128v16(yIn,validIn);
end
```

Compute the FFT by passing 16-element vectors to the object. Use the `getLatency` method to find out when the first output data sample will be ready. Then, add the frame length to determine how many times to call the object. Because the object instance is in the function, use a dummy instance of the object to call the method. Use the loop counter to flip `validIn` to false after N input samples.

```
tempfft = dsp.HDLFFT;
loopCount = getLatency(tempfft,N,V)+N/V;
Yf = zeros(V,loopCount);
validOut = false(V,loopCount);
for loop = 1:1:loopCount
    if ( mod(loop,N/V) == 0 )
        i = N/V;
    else
        i = mod(loop,N/V);
    end
    [Yf(:,loop),validOut(loop)] = HDLFFT128V16(complex(y_vect(:,i)),(loop<=N/V));
end
```

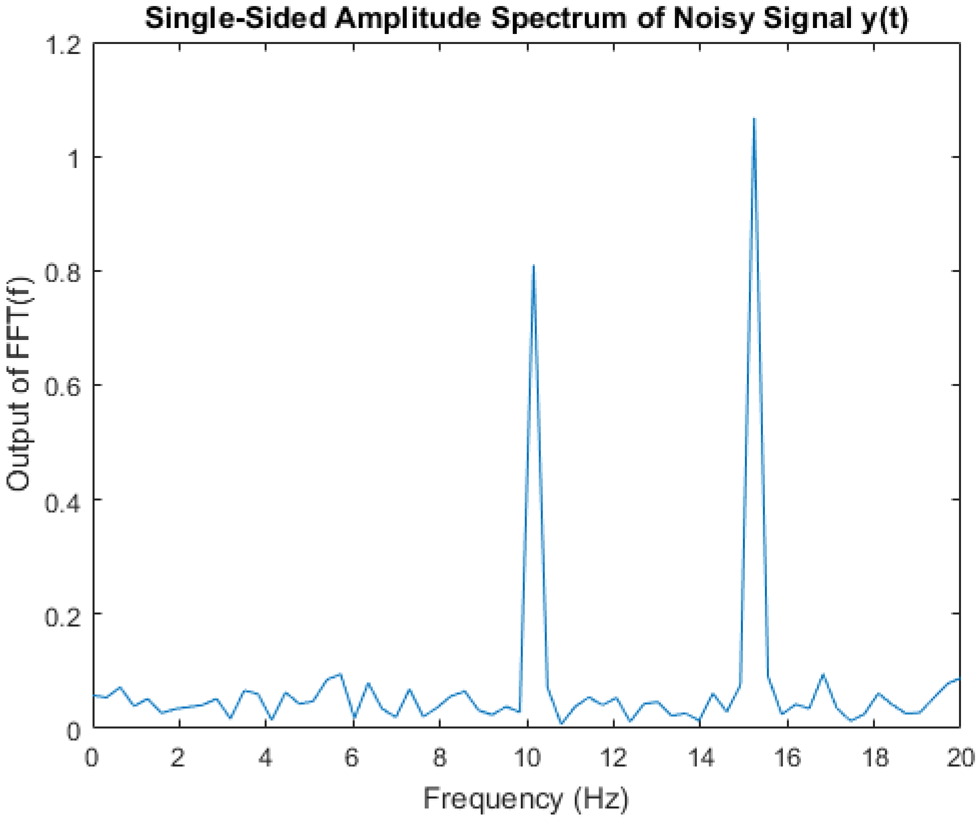
```
fft128v16 =

  dsp.HDLFFT with properties:

                FFTLength: 128
             Architecture: 'Streaming Radix 2^2'
    ComplexMultiplication: 'Use 4 multipliers and 2 adders'
        BitReversedOutput: true
         BitReversedInput: false
                Normalize: false

  Use get to show all properties
```

Discard invalid output samples.

```
C = Yf(:,validOut==1);
Yf_flat = C(:);
```

Plot the frequency channel data from the FFT. The FFT output is in bit-reversed order. Reorder it before plotting.

```
Yr =  bitrevorder(Yf_flat);
plot(Fs/2*linspace(0,1,N/2),2*abs(Yr(1:N/2)/N))
title('Single-Sided Amplitude Spectrum of Noisy Signal y(t)')
xlabel('Frequency (Hz)')
ylabel('Output of FFT(f)')
```



Explore Latency of HDL FFT Object

The latency of the object varies with the FFT length and the vector size. Use the `getLatency` method to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous.

Create a new `dsp.HDLFFT` object and request the latency.

```
hdlfft = dsp.HDLFFT('FFTLength',512);
L512 = getLatency(hdlfft)
```

```
L512 =
```

```
   599
```

Request hypothetical latency information about a similar object with a different FFT length. The properties of the original object do not change.

```
L256 = getLatency(hdlfft,256)
N = hdlfft.FFTLength
```

```
L256 =

   329


N =

   512
```

Request hypothetical latency information of a similar object that accepts eight-sample vector input.

```
L256v8 = getLatency(hdlfft,256,8)
```

```
L256v8 =

    93
```

Set `Normalize` to `true`. The latency does not change.

```
hdlfft.Normalize = true;
L512n = getLatency(hdlfft)
```

```
L512n =

   599
```

Set `BitReversedOutput` to `false`. The latency increases because the object must collect the output before reordering.

```
hdlfft.BitReversedOutput = false;
L512r = getLatency(hdlfft)
```

```
L512r =

        1078
```

Algorithm

This object implements the algorithms described on the `FFT HDL Optimized` block reference page.

Latency

The latency varies with the FFT length and the vector size. Use the getLatency method to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous.

Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a Xilinx Virtex-6 (XC6VLX75T-1FF484) FPGA. The three examples in the tables have this configuration:

- FFT length (default) — 1024
- Complex multiplication — 4 multipliers, 2 adders
- Output scaling — Enabled
- 16-bit complex input data
- Minimize clock enables (HDL Coder parameter)

Performance of the synthesized HDL code varies with your target and synthesis options. For instance, natural order output uses more RAM than bit-reversed output, and real input uses less RAM than complex input.

For a scalar input Radix 2^2 configuration, the design achieves 326 MHz clock frequency. The latency is 1116 cycles. The design uses these resources.

Resource	Uses
LUT	4597
FFS	5353
Xilinx LogiCORE DSP48	12
Block RAM (16K)	6

When you vectorize the same Radix 2^2 implementation to process two 16-bit input samples in parallel, the design achieves 316 MHz clock frequency. The latency is 600 cycles. The design uses these resources.

Resource	Uses
LUT	7653
FFS	9322
Xilinx LogiCORE DSP48	24
Block RAM (16K)	8

The Radix 2 implementation is supported with scalar input data only. The Radix 2 design achieves 295 MHz clock frequency. The latency is 1148 cycles. The design uses these resources.

Resource	Uses
LUT	4060
FFS	5160
Xilinx LogiCORE DSP48	16
Block RAM (16K)	6

See Also

See Also

dsp.FFT | dsp.HDLIFFT | dsp.HDLChannelizer | `FFT HDL Optimized`

Introduced in R2014b

getLatency

System object: dsp.HDLFFT
Package: dsp

Latency of FFT operation

Syntax

```
Y = getLatency(H)
Y = getLatency(H,N)
Y = getLatency(H,N,V)
```

Description

`Y = getLatency(H)` returns the number of cycles, `Y`, that the object takes to calculate the FFT of an input. The latency changes depending on the FFT length.

`Y = getLatency(H,N)` returns the latency, `Y`, that a hypothetical object with FFT Length `N`, and scalar input, would take to calculate the FFT of an input. This method does not change the properties of `H`.

`Y = getLatency(H,N,V)` returns the latency, `Y`, that a hypothetical object with FFT Length `N`, and vector input of size `V`, would take to calculate the FFT of an input. This method does not change the properties of `H`.

Examples

Explore Latency of HDL FFT Object

The latency of the object varies with the FFT length and the vector size. Use the `getLatency` method to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous.

Create a new `dsp.HDLFFT` object and request the latency.

```
hdlfft = dsp.HDLFFT('FFTLength',512);
L512 = getLatency(hdlfft)
```

```
L512 =

   599
```

Request hypothetical latency information about a similar object with a different FFT length. The properties of the original object do not change.

```
L256 = getLatency(hdlfft,256)
N = hdlfft.FFTLength
```

```
L256 =

   329


N =

   512
```

Request hypothetical latency information of a similar object that accepts eight-sample vector input.

```
L256v8 = getLatency(hdlfft,256,8)
```

```
L256v8 =

    93
```

Set `Normalize` to `true`. The latency does not change.

```
hdlfft.Normalize = true;
L512n = getLatency(hdlfft)
```

```
L512n =
```

```
   599
```

Set `BitReversedOutput` to `false`. The latency increases because the object must collect the output before reordering.

```
hdlfft.BitReversedOutput = false;
L512r = getLatency(hdlfft)
```

```
L512r =

        1078
```

Input Arguments

H

HDLFFT System object

N

(optional) FFT length. Use this argument to request the latency of a similar object with FFT length `N`.

V

(optional) Vector size. Use this argument to request the latency of a similar object with `V`-sample vector input. When you do not specify this argument, the function assumes scalar input.

Output Arguments

Y

Cycles of latency that the object takes to calculate the FFT of an input. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous. Each call to the step method simulates one cycle.

step

System object: dsp.HDLFFT
Package: dsp

Process inputs using HDL optimized FFT (fast Fourier transform)

Syntax

```
[Y,validOut] = step(H,X,validIn)
[Y,startOut,endOut,validOut] = step(H,X,validIn)
[Y,validOut] = step(H,X,validIn,resetIn)
[Y,startOut,endOut,validOut] = step(H,X,validIn,resetIn)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[Y,validOut] = step(H,X,validIn)` returns the fast Fourier transform (FFT), `Y`, of the input, `X`, when `validIn` is `true`. `validIn` and `validOut` are logical scalars that indicate the validity of the input and output signals, respectively.

`[Y,startOut,endOut,validOut] = step(H,X,validIn)` returns the FFT, `Y`. `startOut` is `true` on the first sample of a frame of output data. `endOut` is `true` for the last sample of a frame of output data. To use this syntax, set the `StartOutputPort` and `EndOutputPort` properties to `true`.

`[Y,validOut] = step(H,X,validIn,resetIn)` returns the FFT, `Y`, when `validIn` is `true` and `resetIn` is `false`. When `resetIn` is `true`, the object stops the current calculation and clears all internal state. To use this syntax set the `ResetInputPort` property to `true`.

`[Y,startOut,endOut,validOut] = step(H,X,validIn,resetIn)` returns the FFT, `Y`, using all optional control signals. You can use any combination of the optional port syntaxes.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Input Arguments

`H`

HDLFFT System object.

`X`

Scalar or vector of real or complex input data, in fixed-point or integer format. The vector size must be a power of 2 between 1 and 64, and not greater than the FFT length. `double`/`single` is supported for simulation but not for HDL code generation.

`validIn`

Logical scalar that indicates the validity of the input data.

`resetIn`

(optional) Logical scalar to reset the internal state.

Output Arguments

`Y`

Scalar or vector of real or complex output data, in the same format as the input data.

`startOut`

(optional) Logical scalar that is `true` for the first sample of a frame of output data.

`endOut`

(optional) Logical scalar that is `true` for the last sample of a frame of output data.

`validOut`

Logical scalar that indicates the validity of the output data.

dsp.HDLIFFT System object

Package: dsp

Inverse fast Fourier transform—optimized for HDL code generation

Description

The HDL IFFT System object implements a pipelined Radix 2^2 IFFT algorithm which provides hardware speed and area optimization for streaming data applications. The object accepts scalar or vector real or complex input, provides hardware-friendly control signals, and has optional output frame control signals. You can achieve giga-sample-per-second (GSPS) throughput using vector input.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`HIFFT = dsp.HDLIFFT` returns an HDL IFFT System object, `HIFFT`, that performs an inverse fast Fourier transform.

`HIFFT = dsp.HDLIFFT(Name,Value)` returns an HDL IFFT System object, `HIFFT`, with additional options specified by one or more `Name,Value` pair arguments.

Properties

`Architecture`

`'Streaming Radix 2^2'` (default) — An efficient architecture that has lower latency and uses fewer resources than the previous versions.

`'Streaming Radix 2 (this choice will be removed -see release notes).'` — Versions before R2016a use this architecture. Use only for backward-compatibility purposes.

ComplexMultiplication

Select the HDL implementation of complex multipliers. Choose to implement each multiplier with `'Use 4 multipliers and 2 adders'` or `Use 3 multipliers and 5 adders`. Which option is faster or smaller depends on your synthesis tool and target device. This option applies only if you use the Radix 2^2 architecture.

BitReversedOutput

Select the order of the output data.

- `true` (default) — The output channel elements are bit reversed relative to the input order.
- `false` — The output channel elements are in linear order.

The FFT algorithm calculates output in the reverse order to the input, and performs an extra reversal operation when required to provide output in the same order as the input. For vector data, input and output data must be in opposite orders so choose only one of `BitReversedOutput` or `BitReversedInput`. For more information on ordering of the output, see "Linear and Bit-Reversed Output Order".

BitReversedInput

Select the expected order of the input data.

- `true` — The input channel elements are in bit-reversed order.
- `false` (default) — The input channel elements are in linear order.

The FFT algorithm calculates output in the reverse order to the input, and performs an extra reversal operation when required to provide output in the same order as the input. For vector data, input and output data must be in opposite orders so choose only one of `BitReversedOutput` or `BitReversedInput`. For more information on ordering of the output, see "Linear and Bit-Reversed Output Order".

Normalize

Enable output scaling.

- `true` (default) — The object implements an overall 1/N scale factor by scaling the result of each pipeline stage by 2. This adjustment keeps the output of the IFFT in the same amplitude range as its input.

- `false` — The object avoids overflow by increasing the word length by one bit at each stage.

FFTLength

Number of data points used for one FFT calculation. The default value is 1024. The FFT length must be a power of 2 between 2^3 and 2^{16} for HDL code generation.

ResetInputPort

When you set `ResetInputPort` to `true`, the `step` method expects an additional argument, `reset`. When `reset` is `true`, the object stops calculation and clears all internal state. The default value of `ResetInputPort` is `false`.

StartOutputPort

If you set `StartOutputPort` to `true`, the `step` method returns an additional output signal that is `true` on the first cycle of each valid output frame. The default value of `StartOutputPort` is `false`.

EndOutputPort

If you set `EndOutputPort` to `true`, the step method returns an additional output signal that is `true` on the last cycle of each valid output frame. The default value of `EndOutputPort` is `false`.

RoundingMethod

Rounding mode used for fixed-point operations. The IFFT object uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input is single or double type. The default `RoundingMethod` is `Floor`. Each stage rounds after the twiddle factor multiplication but before the butterflies.

Methods

getLatency	Latency of IFFT operation
step	Process inputs using the HDL optimized IFFT (Inverse Fast Fourier Transform)

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Create an IFFT for HDL Code Generation

Create the specifications and input signal. This example uses a 128-point FFT.

```
N = 128;
Fs = 40;
t = (0:N-1)'/Fs;
x = sin(2*pi*15*t) + 0.75*cos(2*pi*10*t);
y = x + .25*randn(size(x));
y_fixed = sfi(y,32,16);
noOp = zeros(1,'like',y_fixed);
```

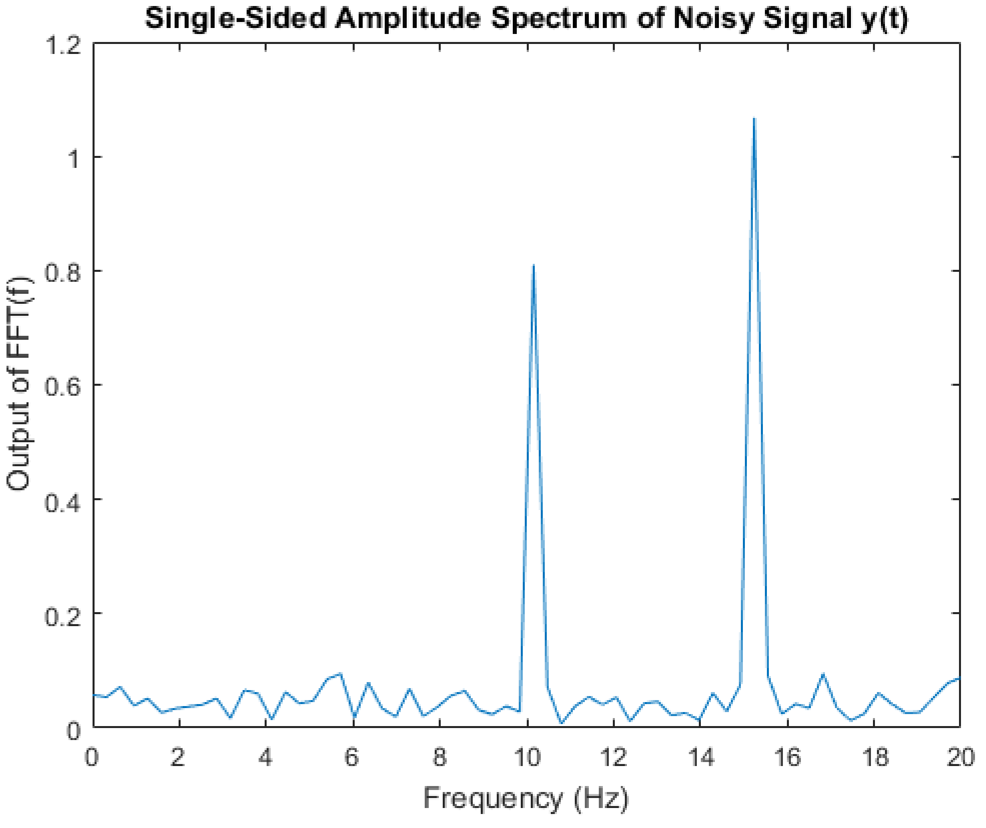
Compute the FFT of the signal to use as the input to the IFFT object.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
hdlfft = dsp.HDLFFT('FFTLength',N,'BitReversedOutput',false);
Yf = zeros(1,4*N);
validOut = false(1,4*N);
for loop = 1:1:N
      [Yf(loop),validOut(loop)] = hdlfft(complex(y_fixed(loop)),true);
end
for loop = N+1:1:4*N
   [Yf(loop),validOut(loop)] = hdlfft(complex(noOp),false);
end
Yf = Yf(validOut == 1);
```

Plot the single-sided amplitude spectrum.

```
plot(Fs/2*linspace(0,1,N/2),2*abs(Yf(1:N/2)/N))
title('Single-Sided Amplitude Spectrum of Noisy Signal y(t)')
xlabel('Frequency (Hz)')
ylabel('Output of FFT(f)')
```



Select frequencies that hold the majority of the energy in the signal. The `cumsum` function doesn't accept fixed-point arguments, so convert the data back to `double`.

```
[Ysort,i] = sort(abs(double(transpose(Yf(1:N)))),1,'descend');
Ysort_d = double(Ysort);
CumEnergy = sqrt(cumsum(Ysort_d.^2))/norm(Ysort_d);
j = find(CumEnergy > 0.9, 1);
     disp(['Number of FFT coefficients that represent 90% of the ', ...
```

```
     'total energy in the sequence: ', num2str(j)])
Yin = zeros(N,1);
Yin(i(1:j)) = Yf(i(1:j));
```

```
Number of FFT coefficients that represent 90% of the total energy in the sequence: 4
```

Write a function that instantiates and calls the IFFT System object™. You can generate HDL from this function.

```
function [yOut,validOut] = HDLIFFT128(yIn,validIn)
%HDLIFFT128
% Processes one sample of data using the dsp.HDLIFFT System object(TM)
% yIn is a fixed-point scalar or column vector.
% validIn is a logical scalar.
% You can generate HDL code from this function.

  persistent ifft128;
  if isempty(ifft128)
    ifft128 = dsp.HDLIFFT('FFTLength',128)
  end
  [yOut,validOut] = ifft128(yIn,validIn);
end
```

Compute the IFFT by calling the function for each data sample.

```
Xt = zeros(1,3*N);
validOut = false(1,3*N);
for loop = 1:1:N
    [Xt(loop),validOut(loop)] = HDLIFFT128(complex(Yin(loop)),true);
end
for loop = N+1:1:3*N
    [Xt(loop),validOut(loop)] = HDLIFFT128(complex(0),false);
end
```

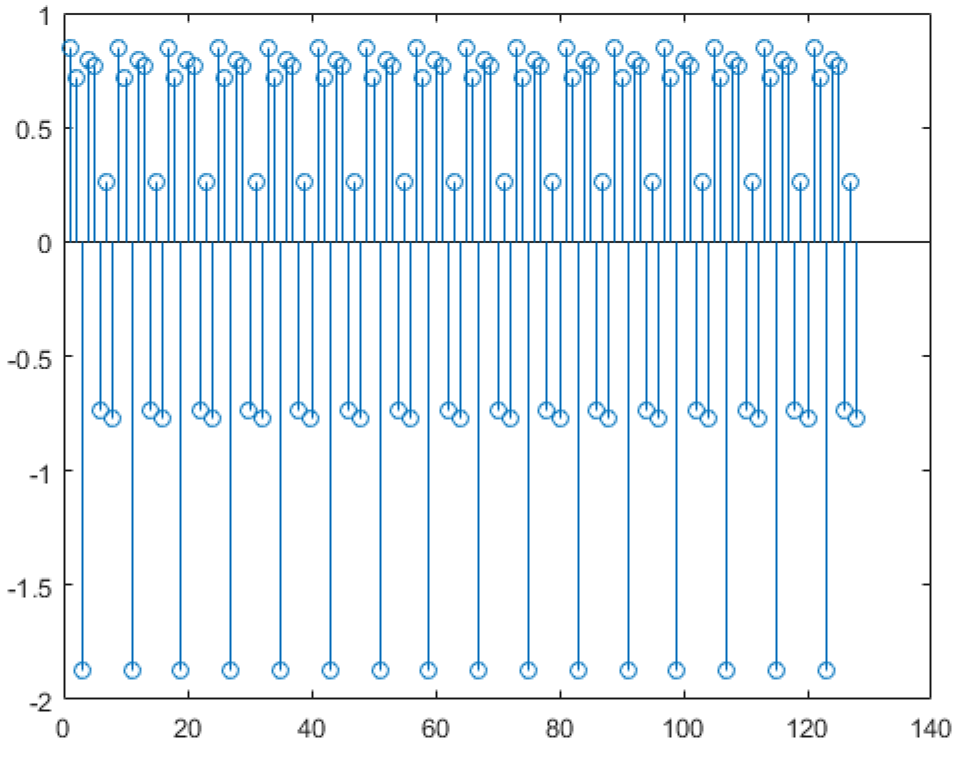
```
ifft128 =

  dsp.HDLIFFT with properties:

              FFTLength: 128
           Architecture: 'Streaming Radix 2^2'
  ComplexMultiplication: 'Use 4 multipliers and 2 adders'
```

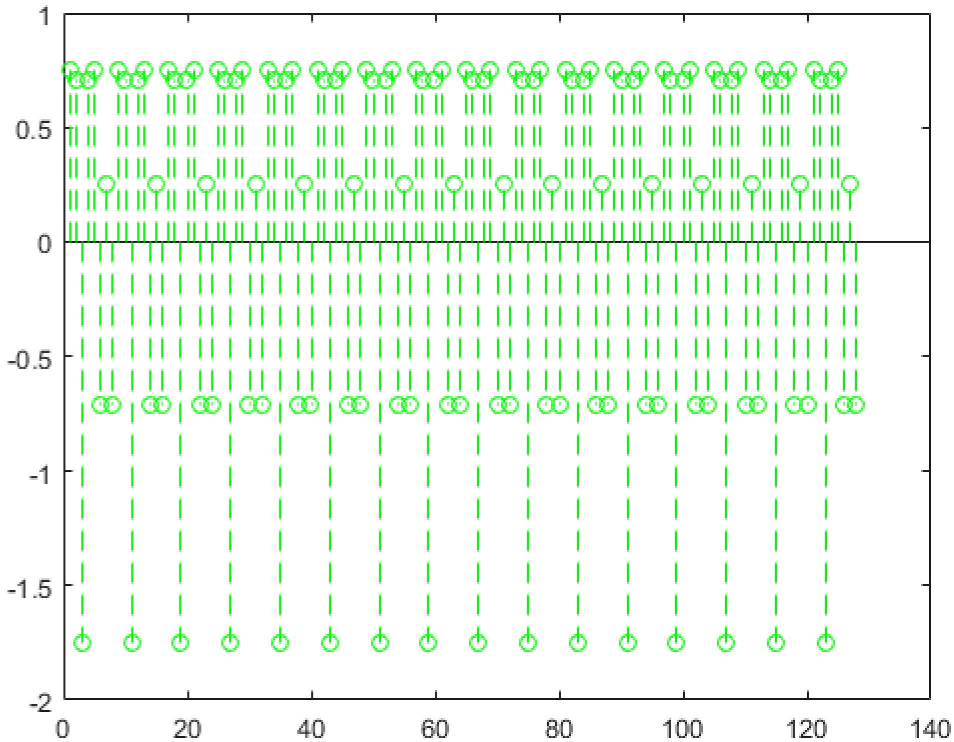
```
        BitReversedOutput: true
         BitReversedInput: false
                Normalize: true

  Use get to show all properties
```

Discard invalid output samples. Then inspect the output and compare it with the input signal. The original input is in green.

```
Xt = Xt(validOut==1);
Xt = bitrevorder(Xt);
norm(x-transpose(Xt(1:N)))
figure
stem(real(Xt))
figure
stem(real(x),'--g')
```

```
ans =

    0.7863
```


1
0.5
0
-0.5
-1
-1.5
-2
0
20
40
60
80
100
120
140



Create a Vector-Input IFFT for HDL Code Generation

Create the specifications and input signal. This example uses a 128-point FFT and computes the transform over 16 samples at a time.

```
N = 128;
V = 16;
Fs = 40;
t = (0:N-1)'/Fs;
x = sin(2*pi*15*t) + 0.75*cos(2*pi*10*t);
y = x + .25*randn(size(x));
y_fixed = sfi(y,32,24);
y_vect = reshape(y_fixed,V,N/V);
```

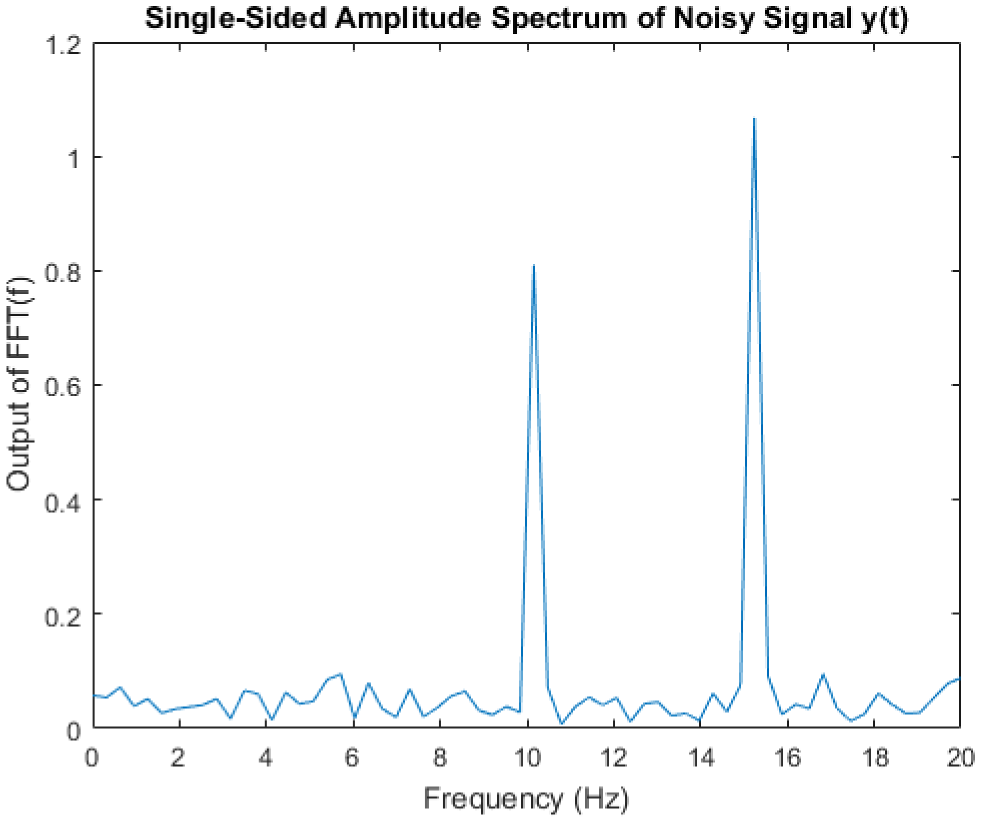
Compute the FFT of the signal, to use as the input to the IFFT object.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
hdlfft = dsp.HDLFFT('FFTLength',N);
loopCount = getLatency(hdlfft,N,V)+N/V;
Yf = zeros(V,loopCount);
validOut = false(V,loopCount);
for loop = 1:1:loopCount
    if ( mod(loop,N/V) == 0 )
        i = N/V;
    else
        i = mod(loop,N/V);
    end
    [Yf(:,loop),validOut(loop)] = hdlfft(complex(y_vect(:,i)),(loop<=N/V));
end
```

Plot the single-sided amplitude spectrum.

```
C = Yf(:,validOut==1);
Yf_flat = C(:);
Yr = bitrevorder(Yf_flat);
plot(Fs/2*linspace(0,1,N/2),2*abs(Yr(1:N/2)/N))
title('Single-Sided Amplitude Spectrum of Noisy Signal y(t)')
xlabel('Frequency (Hz)')
ylabel('Output of FFT(f)')
```



Select frequencies that hold the majority of the energy in the signal. The `cumsum` function doesn't accept fixed-point arguments, so convert the data back to `double`.

```
[Ysort,i] = sort(abs(double(Yr(1:N))),1,'descend');
CumEnergy = sqrt(cumsum(Ysort.^2))/norm(Ysort);
j = find(CumEnergy > 0.9, 1);
     disp(['Number of FFT coefficients that represent 90% of the ', ...
     'total energy in the sequence: ', num2str(j)])
Yin = zeros(N,1);
Yin(i(1:j)) = Yr(i(1:j));
YinVect = reshape(Yin,V,N/V);
```

```
Number of FFT coefficients that represent 90% of the total energy in the sequence: 4
```

Write a function that creates and calls the IFFT System object™. You can generate HDL from this function.

```
function [yOut,validOut] = HDLIFFT128V16(yIn,validIn)
%HDLFFT128V16
% Processes 16-sample vectors of FFT data
% yIn is a fixed-point column vector.
% validIn is a logical scalar value.
% You can generate HDL code from this function.

  persistent ifft128v16;
  if isempty(ifft128v16)
    ifft128v16 = dsp.HDLIFFT('FFTLength',128)
  end
  [yOut,validOut] = ifft128v16(yIn,validIn);
end
```

Compute the IFFT by calling the function for each data sample.

```
Xt = zeros(V,loopCount);
validOut = false(V,loopCount);
for loop = 1:1:loopCount
    if ( mod(loop,N/V) == 0 )
        i = N/V;
    else
        i = mod(loop,N/V);
    end
    [Xt(:,loop),validOut(loop)] = HDLIFFT128V16(complex(YinVect(:,i)),(loop<=N/V));
end
```

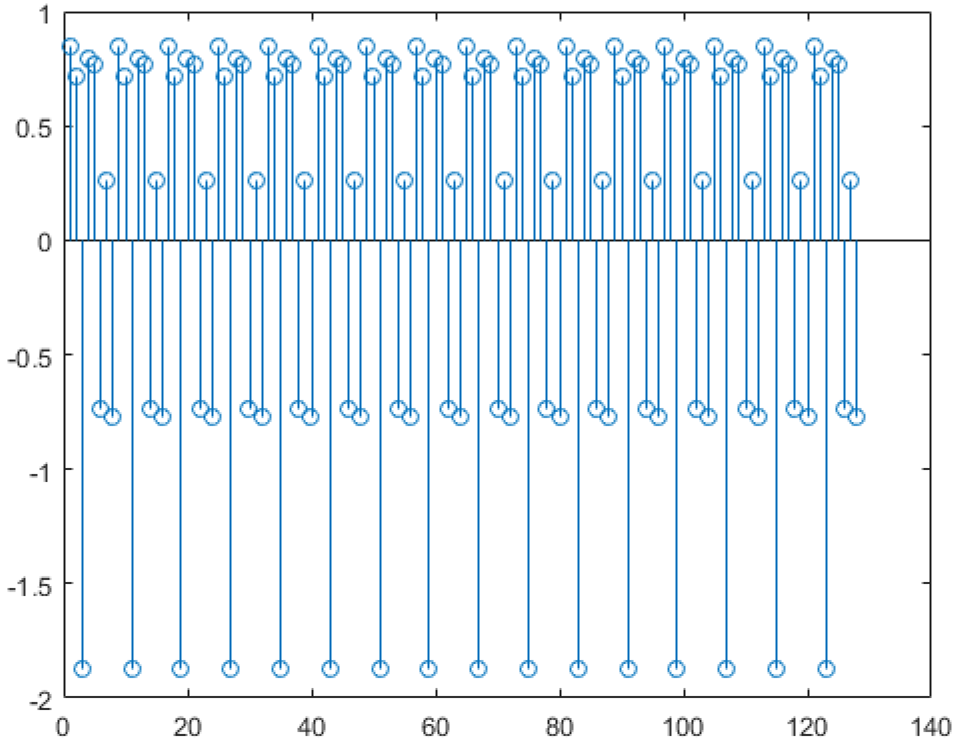
```
ifft128v16 =

  dsp.HDLIFFT with properties:

                FFTLength: 128
             Architecture: 'Streaming Radix 2^2'
    ComplexMultiplication: 'Use 4 multipliers and 2 adders'
        BitReversedOutput: true
         BitReversedInput: false
                Normalize: true
```

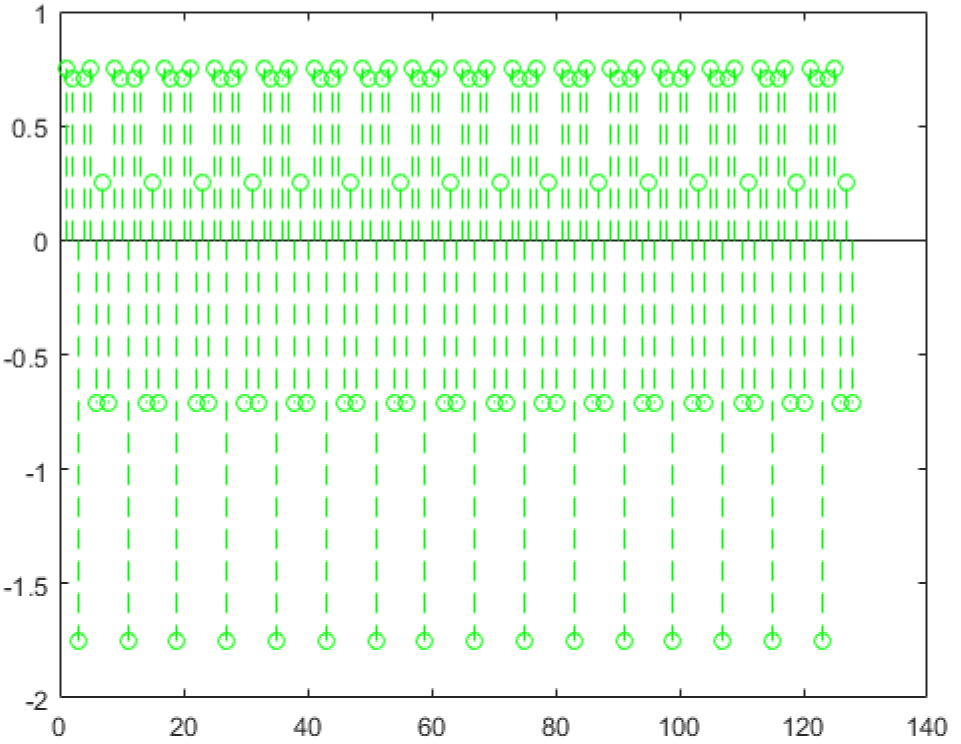
```
  Use get to show all properties
```

Discard invalid output samples. Then inspect the output and compare it with the input signal. The original input is in green.

```
C = Xt(:,validOut==1);
Xt = C(:);
Xt = bitrevorder(Xt);
norm(x-Xt(1:N))
figure
stem(real(Xt))
figure
stem(real(x),'--g')
```

```
ans =

    0.7863
```


1
0.5
0
-0.5
-1
-1.5
-2
0
20
40
60
80
100
120
140



Explore Latency of HDL IFFT Object

The latency of the object varies with the FFT length and the vector size. Use the `getLatency` method to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous.

Create a new `dsp.HDLIFFT` object and request the latency.

```
hdlifft = dsp.HDLIFFT('FFTLength',512);
L512 = getLatency(hdlifft)
```

```
L512 =
```

```
   599
```

Request hypothetical latency information about a similar object with a different FFT length. The properties of the original object do not change. When you do not specify a vector length, the function assumes scalar input data.

```
L256 = getLatency(hdlifft,256)
N = hdlifft.FFTLength
```

```
L256 =

   329


N =

   512
```

Request hypothetical latency information of a similar object that accepts eight-sample vector input.

```
L256v8 = getLatency(hdlifft,256,8)
```

```
L256v8 =

    93
```

Set `Normalize` to `true`. The latency does not change.

```
hdlifft.Normalize = true;
L512n = getLatency(hdlifft)
```

```
L512n =

   599
```

Set `BitReversedOutput` to `false`. This setting increases the latency because the object must collect the output before reordering.

```
hdlifft.BitReversedOutput = false;
L512r = getLatency(hdlifft)


L512r =

        1078
```

Algorithm

This object implements the inverse function of the HDL FFT System object. The algorithm is described on the `FFT HDL Optimized` block reference page.

Latency

The latency varies with the FFT length and the vector size. Use the getLatency method to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous.

Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a Xilinx Virtex-6 (XC6VLX75T-1FF484) FPGA. The three examples in the tables have this configuration:

- FFT length (default) — 1024
- Complex multiplication — 4 multipliers, 2 adders
- Output scaling — Enabled
- 16-bit complex input data
- Minimize clock enables (HDL Coder parameter)

Performance of the synthesized HDL code varies with your target and synthesis options. For instance, natural order output uses more RAM than bit-reversed output, and real input uses less RAM than complex input.

For a scalar input Radix 2^2 configuration, the design achieves 326 MHz clock frequency. The latency is 1116 cycles. The design uses these resources.

Resource	Uses
LUT	4597
FFS	5353
Xilinx LogiCORE DSP48	12
Block RAM (16K)	6

When you vectorize the same Radix 2^2 implementation to process two 16-bit input samples in parallel, the design achieves 316 MHz clock frequency. The latency is 600 cycles. The design uses these resources.

Resource	Uses
LUT	7653
FFS	9322
Xilinx LogiCORE DSP48	24
Block RAM (16K)	8

The Radix 2 implementation is supported with scalar input data only. The Radix 2 design achieves 295 MHz clock frequency. The latency is 1148 cycles. The design uses these resources.

Resource	Uses
LUT	4060
FFS	5160
Xilinx LogiCORE DSP48	16
Block RAM (16K)	6

See Also

See Also

dsp.IFFT | dsp.HDLFFT | `IFFT HDL Optimized`

Introduced in R2014b

getLatency

System object: dsp.HDLIFFT
Package: dsp

Latency of IFFT operation

Syntax

```
Y = getLatency(H)
Y = getLatency(H,N)
Y = getLatency(H,N,V)
```

Description

`Y = getLatency(H)` returns the number of cycles, `Y`, that the object takes to calculate the IFFT of an input. The latency depends on the FFT length.

`Y = getLatency(H,N)` returns the latency, `Y`, that a hypothetical object with FFT Length `N`, and scalar input data, would take to calculate the IFFT of an input. This method does not change the properties of `H`.

`Y = getLatency(H,N,V)` returns the latency, `Y`, that a hypothetical object with FFT Length `N`, and vector input of size `V`, would take to calculate the IFFT of an input. This method does not change the properties of `H`.

Examples

Explore Latency of HDL IFFT Object

The latency of the object varies with the FFT length and the vector size. Use the `getLatency` method to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous.

Create a new `dsp.HDLIFFT` object and request the latency.

```
hdlifft = dsp.HDLIFFT('FFTLength',512);
L512 = getLatency(hdlifft)
```

```
L512 =

   599
```

Request hypothetical latency information about a similar object with a different FFT length. The properties of the original object do not change. When you do not specify a vector length, the function assumes scalar input data.

```
L256 = getLatency(hdlifft,256)
N = hdlifft.FFTLength
```

```
L256 =

   329


N =

   512
```

Request hypothetical latency information of a similar object that accepts eight-sample vector input.

```
L256v8 = getLatency(hdlifft,256,8)
```

```
L256v8 =

    93
```

Set `Normalize` to `true`. The latency does not change.

```
hdlifft.Normalize = true;
L512n = getLatency(hdlifft)
```

```
L512n =
```

```
   599
```

Set `BitReversedOutput` to `false`. This setting increases the latency because the object must collect the output before reordering.

```
hdlifft.BitReversedOutput = false;
L512r = getLatency(hdlifft)
```

```
L512r =

        1078
```

Input Arguments

H

HDLIFFT System object

N

(optional) FFT length. Use this argument to request the latency of a similar object with FFT length `N`.

V

(optional) Vector size. Use this argument to request the latency of a similar object with `V`-sample vector input. When you do not specify this argument, the function assumes scalar input.

Output Arguments

Y

Cycles of latency that the object takes to calculate the IFFT of an input. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous. Each call to the step method simulates one cycle.

step

System object: dsp.HDLIFFT
Package: dsp

Process inputs using the HDL optimized IFFT (Inverse Fast Fourier Transform)

Syntax

```
[Y,validOut] = step(H,X,validIn)
[Y,startOut,endOut,validOut] = step(H,X,validIn)
[Y,validOut] = step(H,X,validIn,resetIn)
[Y,startOut,endOut,validOut] = step(H,X,validIn,resetIn)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[Y,validOut] = step(H,X,validIn)` returns the inverse fast Fourier transform (IFFT), `Y`, of the input, `X`, when `validIn` is `true`. `validIn` and `validOut` are logical scalars that indicate the validity of the input and output signals, respectively.

`[Y,startOut,endOut,validOut] = step(H,X,validIn)` returns the IFFT, `Y`. `startOut` is `true` on the first sample of a frame of output data. `endOut` is `true` for the last sample of a frame of output data. To use this syntax, set the `StartOutputPort` and `EndOutputPort` properties to `true`.

`[Y,validOut] = step(H,X,validIn,resetIn)` returns the IFFT, `Y`, when `validIn` is `true` and `resetIn` is `false`. When `resetIn` is `true`, the object stops the current calculation and clears all internal state. To use this syntax set the `ResetInputPort` property to `true`.

`[Y,startOut,endOut,validOut] = step(H,X,validIn,resetIn)` returns the IFFT, `Y`, using all optional control signals. You can use any combination of the optional port syntaxes.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Input Arguments

`H`

HDLIFFT System object.

`X`

Scalar or vector of real or complex input data, in fixed-point or integer format. The vector size must be a power of 2 between 1 and 64, and not greater than the FFT length. `double`/`single` is supported for simulation but not for HDL code generation.

`validIn`

Logical scalar that indicates the validity of the input data.

`resetIn`

(optional) Logical scalar to reset the internal state.

Output Arguments

`Y`

Scalar or vector of real or complex output data, in the same format as the input data.

`startOut`

(optional) Logical scalar that is `true` for the first sample of a frame of output data.

`endOut`

(optional) Logical scalar that is `true` for the last sample of a frame of output data.

`validOut`

Logical scalar that indicates the validity of the output data.

dsp.HDLNCO System object

Package: dsp

Generate real or complex sinusoidal signals—optimized for HDL code generation

Description

The HDL NCO System object generates real or complex sinusoidal signals. In addition, the HDL NCO System object provides hardware-friendly control signals, optional reset signal, optional valid input signal, optional phase output signal, and an optional external dither input signal. It uses the same phase accumulation and lookup table technology as implemented in the NCO System object. You can use the lookup table compression option to significantly reduce the lookup table size with less than one LSB loss in precision. The System object does not support the property that allows the block to synthesize the LUT to a ROM on an FPGA.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`HDLNCO = dsp.HDLNCO` returns a numerically controlled oscillator (NCO) System object, `HDLNCO`, that generates a real or complex sinusoidal signal. The amplitude of the generated signal is always 1.

`HDLNCO = dsp.HDLNCO(Name,Value)` returns an HDL NCO System object, `HDLNCO`, with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

`HDLNCO = dsp.HDLNCO(Inc,'PhaseIncrementSource','Property')` returns an HDL NCO System object, `HDLNCO`, with the `PhaseIncrement` property set to `Inc`. `Inc` is an integer scalar. To use the `PhaseIncrement` property, set the `PhaseIncrementSource` property to `Property`. You can add other `Name,Value` pairs before or after `PhaseIncrementSource`.

Input Arguments

`Inc`

An integer scalar value for the `PhaseIncrement` property. To use this value, set the `PhaseIncrementSource` property to `Property`.

Default: 100

Properties

This object supports `double` for simulation but not for HDL code generation. When a data input is fixed point, or when no data input ports are enabled, the object computes the output waveform based on the fixed-point property settings. When a data input is floating-point, the object ignores `NumDitherBits`, `PhaseQuantization`, `NumQuantizerAccumulatorBits`, `LUTCompress`, and the fixed-point data type properties, and computes a double-precision output waveform.

`PhaseIncrementSource`

Select which phase increment the object uses. Set `PhaseIncrementSource` to `Property` to use the `PhaseIncrement` property. Set `PhaseIncrementSource` to `Input port` to use an input argument of the `step` method. The default is `Input port`.

```
hnco = dsp.HDLNCO(...,'PhaseIncrementSource','Property','PhaseIncrement',phIncr, ...)
```

`PhaseIncrement`

Specify the phase increment as an integer scalar. This property is applicable when you set the `PhaseIncrementSource` property to `Property`. The default value of this property is 100.

`PhaseOffsetSource`

Select which phase offset the object uses. Set `PhaseOffsetSource` property to `Property` to use the `PhaseOffset` property. Set `PhaseOffsetSource` property to `Input port` to use an input argument of the `step` method. The default is `Property`.

```
hnco = dsp.HDLNCO(...,'PhaseOffsetSource','Property','PhaseOffset',phOffset,...)
```

PhaseOffset

Specify the phase offset as an integer scalar. This property is applicable when you set the `PhaseOffsetSource` property to `Property`. The default value of `PhaseOffset` is 0.

DitherSource

Select which dither size the object uses. Set `DitherSource` property to `Property` to use the `NumDitherBits` property. Other options are `Input port` to use an input argument of the `step` method, or `None` to disable dithering. The default is `Property`.

```
hnco = dsp.HDLNCO(...,'DitherSource','Property','NumDitherBits',ditherBits,...)
```

NumDitherBits

Specify the number of dither bits as a positive integer. This property is applicable when you set the `DitherSource` property to `Property`. The default value of `NumDitherBits` is 4.

PhaseQuantization

Set this property to `true` to enable quantization of the accumulated phase. The default value of `PhaseQuantization` is `true`.

```
hnco = dsp.HDLNCO(...,'PhaseQuantization',true,'NumQuantizerAccumulatorBits',accumBits,...)
```

NumQuantizerAccumulatorBits

Specify the number of quantizer accumulator bits as an integer scalar greater than 1 and less than the accumulator word length. `NumQuantizerAccumulatorBits` determines the number of entries in the lookup table of sine values. This property is applicable when you set `PhaseQuantization` to `true`. The default value of this property is 12.

LUTCompress

Set this property to `true` to enable lookup table compression. The object uses the Sunderland compression method to reduce the size of the lookup table. The default value of this property is `false`.

Waveform

Choose whether the output of the object is `Sine`, `Cosine`, `Complex exponential`, or `Sine and cosine` signals. If you select `Complex exponential`, the output is of the form `sine + j*cosine`. If you select `Sine and cosine`, the `step` method returns an additional output. The default is `Sine`.

PhasePort

Set `PhasePort` to `true` to return the current phase along with the output from the `step` method. The default value of this property is `false`.

ResetAction

Set `ResetAction` to `true` to enable a reset argument to the `step` method. The default value of this property is `false`.

ValidInputPort

Set `ValidInputPort` to `false` to disable the `ValidIn` input argument to the `step` method. The default value is `true`.

OverflowAction

Overflow mode for fixed-point operations. `OverflowAction` is a constant property with value `Wrap`.

RoundingMethod

Rounding mode for fixed-point operations. `RoundingMethod` is a constant property with value `Floor`.

AccumulatorDataType

Accumulator data type description. This property is a constant with value `Binary point scaling`.

AccumulatorSigned

Select signed or unsigned accumulator data format. This property is a constant. All output is signed format.

AccumulatorWL

Accumulator word length. Default is 16 bits.

AccumulatorFL

Accumulator fraction length. This property is a constant with value 0 bits.

OutputDataType

Specify the output signal data type. Options are: `double`, `single`, and `Binary point scaling`. If this property is set to `Binary point scaling`, the output sign, word length, and fraction length are taken from the following three properties. The default is `Binary point scaling`.

OutputSigned

Select signed or unsigned output data. This property is a constant. All output is signed format.

OutputWL

Output data word length. The default is 16 bits.

OutputFL

Output data fraction length. The default is 14 bits.

Methods

step	Process inputs using the HDL optimized NCO (Numerically Controlled Oscillator)

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object

Common to All System Objects	
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Design a HDL-Compatible NCO Source

This example shows how to design an HDL-compatible NCO source.

```
F0 = 510;     % Output frequency = 510 Hz
df = 0.05;    % Frequency resolution = 0.05 Hz
minSFDR = 96; % Spurious free dynamic range >= 96 dB
Ts = 1/4000;  % Sample period = 1/4000 seconds
dphi = pi/2;  % Desired phase offset = pi/2;
```

Calculate the number of accumulator bits required for the frequency resolution, and the number of quantized accumulator bits to satisfy the SFDR requirement.

```
Nacc = ceil(log2(1/(df*Ts)));
actdf = 1/(Ts*2^Nacc); % Actual frequency resolution achieved
Nqacc = ceil((minSFDR-12)/6);
```

Calculate the phase increment and offset.

```
phIncr = round(F0*Ts*2^Nacc);
phOffset = 2^Nacc*dphi/(2*pi);
```

Construct a NCO HDL System object™. Set the properties to the values calculated. Call the object to generate data points in a sine wave. The input to the object is a valid signal.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

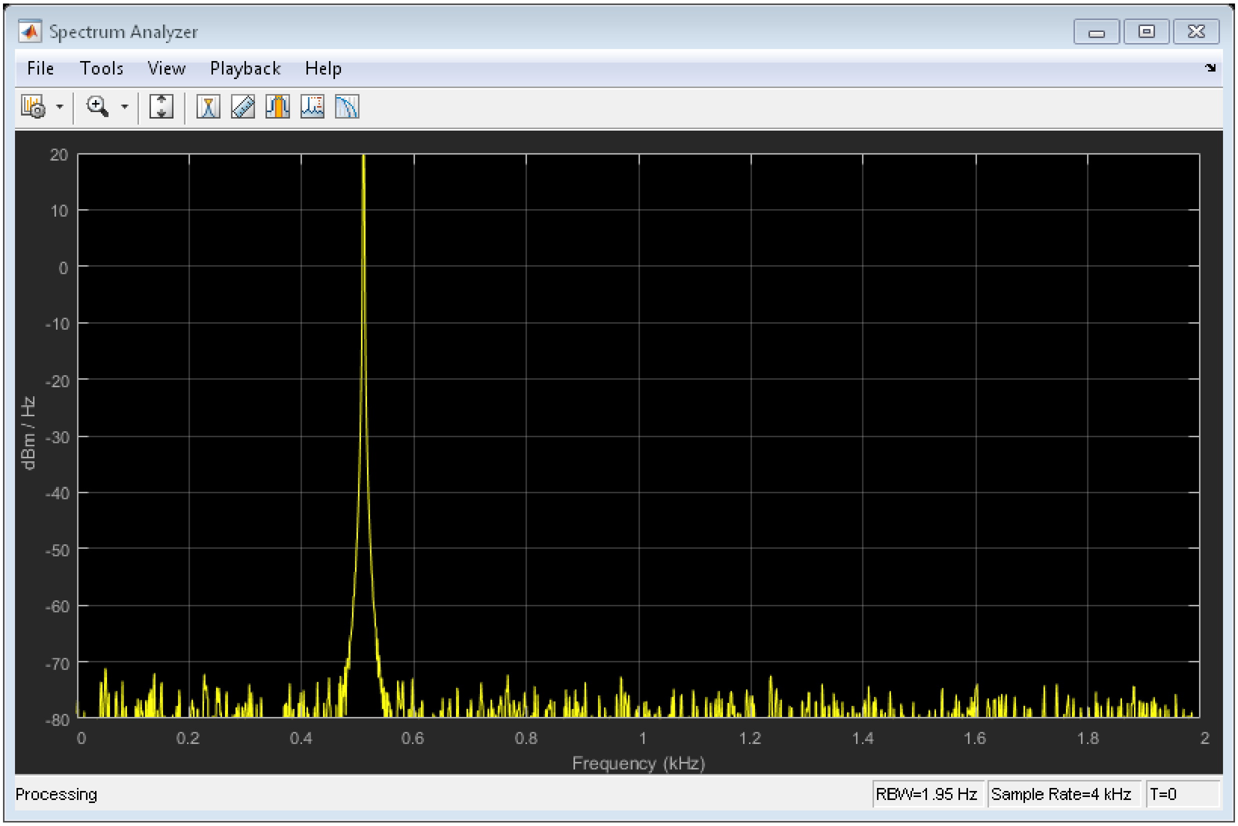
```
hdlnco = dsp.HDLNCO('PhaseIncrementSource','Property', ...
       'PhaseIncrement',phIncr,...
       'PhaseOffset',phOffset,...
```

```
        'NumDitherBits',4, ...
        'NumQuantizerAccumulatorBits',Nqacc,...
        'AccumulatorWL',Nacc)
for k = 1:1/Ts
   y(k) = hdlnco(true);
end
```

```
hdlnco =

  dsp.HDLNCO with properties:

           PhaseIncrementSource: 'Property'
                 PhaseIncrement: 16712
              PhaseOffsetSource: 'Property'
                    PhaseOffset: 32768
                   DitherSource: 'Property'
                  NumDitherBits: 4
              PhaseQuantization: true
    NumQuantizerAccumulatorBits: 14
                    LUTCompress: false
                    ResetAction: false
                 ValidInputPort: true
                       Waveform: 'Sine'
                      PhasePort: false

  Use get to show all properties
```

Plot the mean-square spectrum of the 510 Hz sine wave generated by the NCO.

```
sa = dsp.SpectrumAnalyzer('SampleRate',1/Ts);
sa.SpectrumType = 'Power density';
sa.PlotAsTwoSidedSpectrum = false;
sa(y');
```



Algorithms

Lookup Table Algorithm

When you enable lookup table (LUT) compression, the HDL NCO System object applies the Sunderland compression method. Sunderland techniques use trigonometric identities to divide each phase of the quarter sine wave into three components and express it as:

$$\sin(A+B+C) = \sin(A+B)\cos C + \cos A \cos B \cos C - \sin A \sin B \sin C$$

If the phase has 12 bits, the components are defined as:

- A, the four most significant bits

  $(0 \le A \le \frac{\pi}{2})$

- B, the following four bits

  $(0 \le B \le \frac{\pi}{2} \times 2^{-4})$

- C, the four least significant bits

  $(0 \le C \le \frac{\pi}{2} \times 2^{-8})$

Because C is small enough that sin(C)≅1 and cos(C)≅0, the equation is approximated by:

$$\sin(A+B+C) \approx \sin(A+B) + \cos A \sin C$$

The HDL NCO System object implements this equation with one LUT for sin(A+B) and one LUT for cos(A)sin(C). The second term is a fine correction factor and can be truncated to fewer bits without losing precision. With the default accumulator size of 16 bits, and the example phase width of 12 bits, the LUTs use only 2^8×16 plus 2^8×4 bits (5kb). A quarter sine lookup table would use 2^{12}×16 bits (65kb). This approximation is accurate within 1 LSB which gives an SNR of at least 60 dB on the output. See L. Cordesses, "Direct Digital Synthesis: A Tool for Periodic Wave Generation (Part 1)", IEEE Signal Processing Magazine, DSP Tips & Tricks column, pp. 50–54, Vol. 21, No. 4 July 2004.

See Also

See Also

dsp.NCO | `NCO HDL Optimized`

Introduced in R2013a

step

System object: dsp.HDLNCO
Package: dsp

Process inputs using the HDL optimized NCO (Numerically Controlled Oscillator)

Syntax

```
[Y,ValidOut] = step(HDLNCO,Inc,ValidIn)
[Y,ValidOut] = step(HDLNCO)
[Y,ValidOut] = step(HDLNCO,Inc,Offset,Dither,ValidIn)
[Y,Phase,ValidOut]= step(HDLNCO,___)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`[Y,ValidOut] = step(HDLNCO,Inc,ValidIn)` returns a sinusoidal signal, `Y`, generated by the HDLNCO System object, with the specified phase increment, `Inc`. When `ValidIn` is `true`, `Inc` is added to the accumulator.

`[Y,ValidOut] = step(HDLNCO)` returns a waveform, `Y`, when you set the `PhaseIncrementSource`, `PhaseOffsetSource`, and `DitherSource` properties to `Property`, and `ValidInputPort` to `false`. These properties are independent of each other. You can mix and match the activation of these arguments.

`[Y,ValidOut] = step(HDLNCO,Inc,Offset,Dither,ValidIn)` returns a waveform, `Y`, with phase increment, `Inc`, phase offset, `Offset`, and dither, `Dither`. This syntax applies when you set the `PhaseIncrementSource`, `PhaseOffsetSource`, and `DitherSource` properties to `Input port`. These properties are independent of each other. You can mix and match the activation of these arguments.

`[Y,Phase,ValidOut]= step(HDLNCO,___)` returns a waveform, `Y`, and output `Phase`, when the `PhasePort` property is `true`. The object returns the waveform value, Y, as a sine value, a cosine value, a complex exponential value, or a [Sine,Cosine] pair of values, depending on the Waveform property. This syntax can include any of the arguments from other syntaxes.

Note: obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Input Arguments

HDLNCO

dsp.HDLNCO System object.

Inc

Phase increment, specified as a scalar integer.

The object accepts this argument when you set the PhaseIncrementSource property to Input port.

Supported data types:

- fixdt([],N,0)
- int, uint
- double and single are allowed for simulation but not for HDL code generation.

ValidIn

Enable signal, specified as a logical scalar.

The object increments the phase when ValidIn input is high. When ValidIn is low, the phase is held. The object accepts this argument when you set the ValidInputPort property to true.

Offset

Phase offset, specified as a scalar integer.

The object accepts this argument when you set the `PhaseSource` property to `Input port`.

Supported data types:

- `fixdt([],N,0)`
- `int`, `uint`
- `double` and `single` are allowed for simulation but not for HDL code generation.

Dither

Dither, specified as a scalar integer.

The object accepts this argument when you set the `DitherSource` property to `Input port`.

Supported data types:

- `fixdt([],N,0)`
- `int`, `uint`
- `double` and `single` are allowed for simulation but not for HDL code generation.

Reset

Request reset of the accumulator, specified as a `logical` scalar value.

The object resets the accumulator to zero when `Reset` is high. The object accepts this argument when you set the `ResetAction` property to `true`.

Output Arguments

Y

Waveform, returned as a scalar fixed-point or floating-point value, or as a [`Sine,Cosine`] pair of values.

The type of the output waveform depends on the `Waveform` property.

- `Sine` (default) — returns a generated sine wave
- `Cosine` — returns a generated cosine wave
- `Complex exponential` — returns a wave in the form `sine + j*cosine`
- `Sine and cosine` — returns a pair of values representing corresponding sine and cosine waves

Supported data types:

- `fixdt()`
- `double` and `single` are allowed for simulation but not for HDL code generation.

ValidOut

Validity of the `Y` output, returned as a `logical` scalar.

When `ValidOut` is high, the data output is valid. When `ValidOut` is low, the data output is not valid.

Phase

Current phase of the NCO, returned as a scalar.

The object returns this additional output when you set the `PhasePort` property to `true`.

Supported data types:

- `fixdt(1,M,-Z)`, where *M* is the number of quantized accumulator bits, and *Z* is the accumulator word length.
- `double` and `single` are allowed for simulation but not for HDL code generation. The object returns `Phase` as floating point if the input to the `step` method is floating point.

dsp.HampelFilter System object

Package: dsp

Filter outliers using Hampel identifier

Description

The `dsp.HampelFilter` System object detects and removes the outliers of the input signal by using the Hampel identifier. The Hampel identifier is a variation of the three-sigma rule of statistics that is robust against outliers. For each sample of the input signal, the object computes the median of a window composed of the current sample and $\frac{Len-1}{2}$ adjacent samples on each side of current sample. *Len* is the window length you specify through the `WindowLength` property. The object also estimates the standard deviation of each sample about its window median by using the median absolute deviation. If a sample differs from the median by more than the threshold multiplied by the standard deviation, the filter replaces the sample with the median. For more information, see "Algorithms" on page 2-859.

The object accepts multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$, and $n > 1$. m is the number of samples in each frame (channel), and n is the number of channels. The object also accepts variable-size inputs. After the object is locked, you can change the size of each input channel. However, the number of channels cannot change.

To filter the input signal using a Hampel identifier:

1. Create a `dsp.HampelFilter` object and set the properties of the object.
2. Call step to filter the signal using the Hampel identifier.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`hampFilt = dsp.HampelFilter` returns a Hampel filter object, `hampFilt`, using the default properties.

`hampFilt = dsp.HampelFilter(Len)` sets the `WindowLength` property to `Len`.

`hampFilt = dsp.HampelFilter(Len, Lim)` sets the `WindowLength` property to `Len` and the `Threshold` property to `Lim`.

`hampFilt = dsp.HampelFilter(Name,Value)` specifies properties using `Name,Value` pairs. Unspecified properties have default values.

Example:

```
hampFilt = dsp.HampelFilter(11,2);
```

Properties

If a property is listed as tunable, then you can change its value even when the object is locked.

`WindowLength` — Length of the sliding window

11 (default) | positive odd scalar integer

Length of the sliding window, specified as a positive odd scalar integer. The window of finite length slides over the data, and the object computes the median and median absolute deviation of the data in the window.

`Threshold` — Threshold for outlier detection

3 (default) | positive real scalar

Threshold for outlier detection, specified as a positive real scalar. For information on how this property is used to detect the outlier, see “Algorithms” on page 2-859.

Tunable: Yes

Methods

reset	Reset internal states of System object

step	Remove outliers from input

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Remove High-Frequency Noise Using Hampel Filter

Filter high-frequency noise from a noisy sine wave signal using a Hampel filter. Compare the performance of the Hampel filter with a median filter.

Initialization

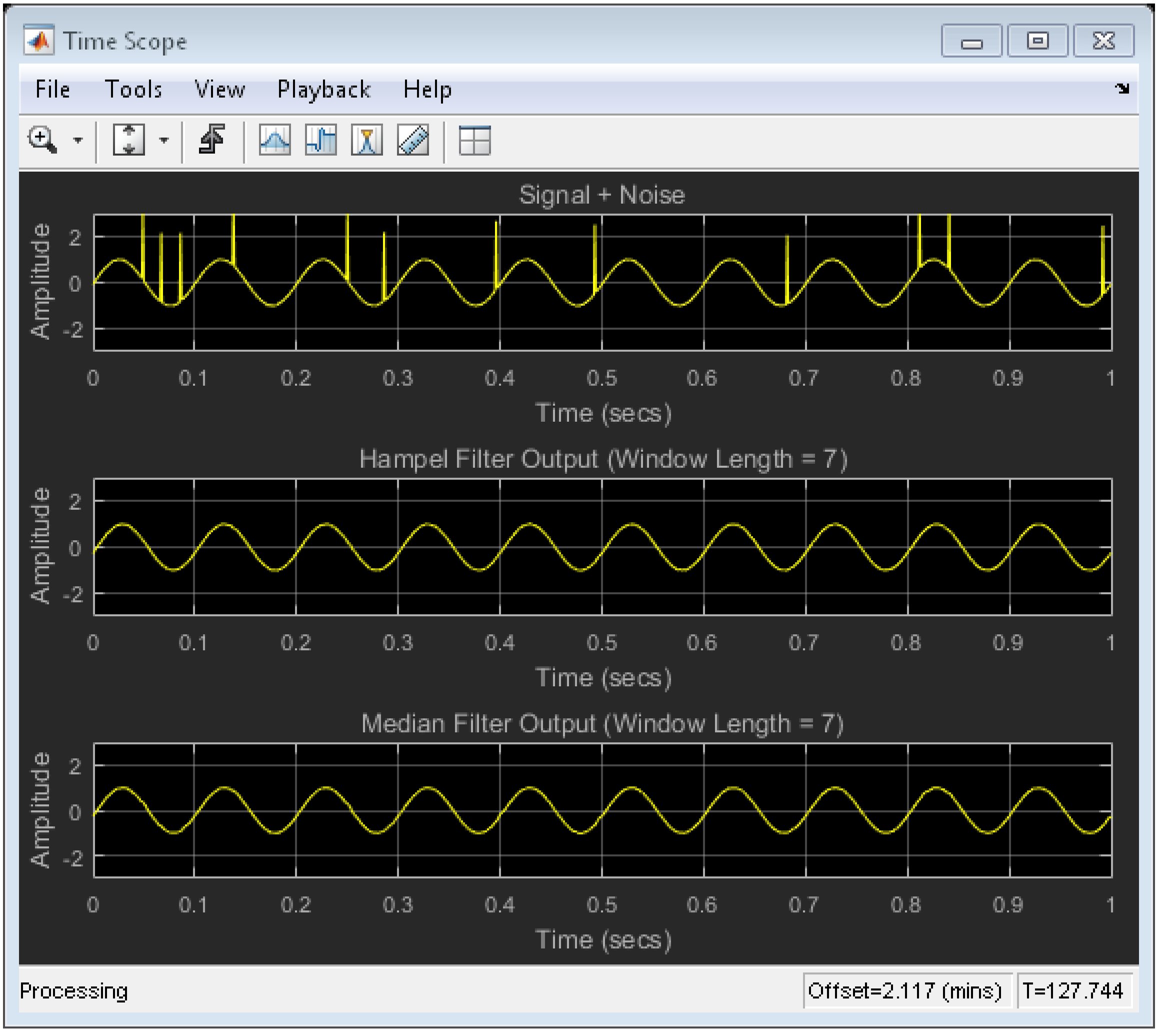
Set up a `dsp.HampelFilter` and a `dsp.MedianFilter` object. These objects use the sliding window method with a window length of 7. Create a time scope for viewing the output.

```
Fs = 1000;
hampFilt = dsp.HampelFilter(7);
medFilt = dsp.MedianFilter(7);
scope  = dsp.TimeScope('SampleRate',Fs, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpan',1,'ShowGrid',true, ...
    'YLimits',[-3 3], ...
    'LayoutDimensions',[3 1], ...
    'NumInputPorts',3);
scope.ActiveDisplay = 1;
scope.Title = 'Signal + Noise';
scope.ActiveDisplay = 2;
scope.Title = 'Hampel Filter Output (Window Length = 7)';
scope.ActiveDisplay = 3;
scope.Title = 'Median Filter Output (Window Length = 7)';
```

Filter the Noisy Sine Wave Using a Window of Length 7

Generate a noisy sine wave signal with a frequency of 10 Hz. Apply the Hampel filter and the median filter object to the signal. View the output on the time scope.

```
FrameLength = 256;
sine = dsp.SineWave('SampleRate',Fs,'Frequency',10,...
    'SamplesPerFrame',FrameLength);

for i = 1:500
    hfn = 3 * (rand(FrameLength,1) < 0.02);
    x = sine() + 1e-2 * randn(FrameLength,1) + hfn;
    y1 = hampFilt(x);
    y2 = medFilt(x);
    scope(x,y1,y2);
end
```

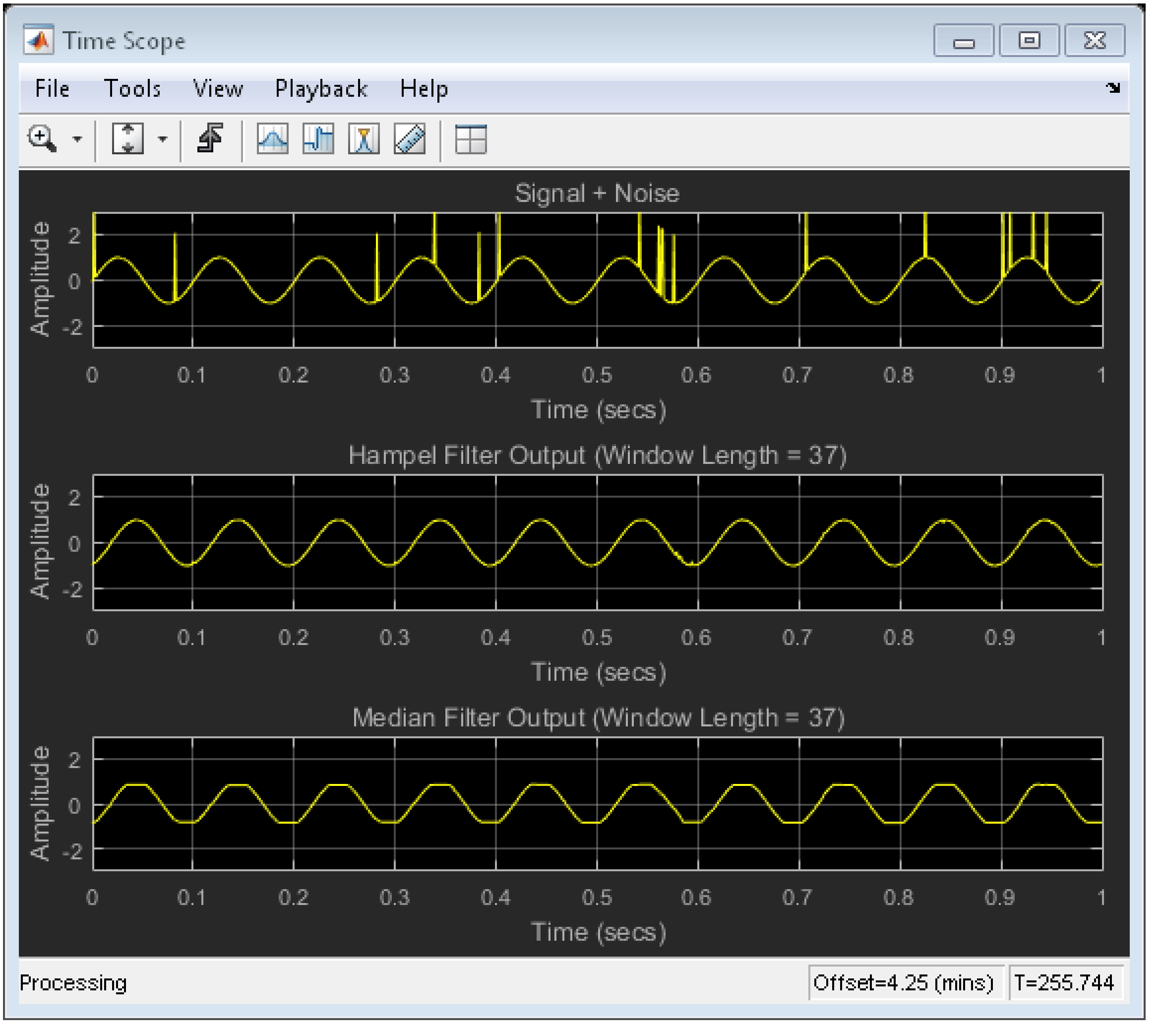


Both filters remove the high-frequency noise. However, when you increase the window length, the Hampel filter is preferred. Unlike the median filter, the Hampel filter preserves the shape of the sine wave even with large window lengths.

Filter the Noisy Sine Wave Using a Window of Length 37

Increase the window length of both the filters to 37. Filter the noisy sine wave and view the filtered output on the time scope. To change the window length of the filters, you must release the filter objects at the start of the processing loop.

```
release(hampFilt);
release(medFilt);
hampFilt.WindowLength = 37;
medFilt.WindowLength = 37;
scope.ActiveDisplay = 1;
scope.Title = 'Signal + Noise';
scope.ActiveDisplay = 2;
scope.Title = 'Hampel Filter Output (Window Length = 37)';
scope.ActiveDisplay = 3;
scope.Title = 'Median Filter Output (Window Length = 37)';
for i = 1:500
    hfn = 3 * (rand(FrameLength,1) < 0.02);
    x = sine() + 1e-2 * randn(FrameLength,1) + hfn;
    y1 = hampFilt(x);
    y2 = medFilt(x);
    scope(x,y1,y2);
end
```



Time Scope
File Tools View Playback Help
Signal + Noise
Amplitude
2
0
-2
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
Time (secs)
Hampel Filter Output (Window Length = 37)
Amplitude
Time (secs)
Median Filter Output (Window Length = 37)
Amplitude
Time (secs)
Processing
Offset=4.25 (mins)
T=255.744

The median filter flattens the crests and troughs of the sine wave due to the median operation over a large window of data. The Hampel filter preserves the shape of the signal, in addition to removing the outliers.

Remove High-Frequency Noise from Gyroscope Data Using Hampel Filter

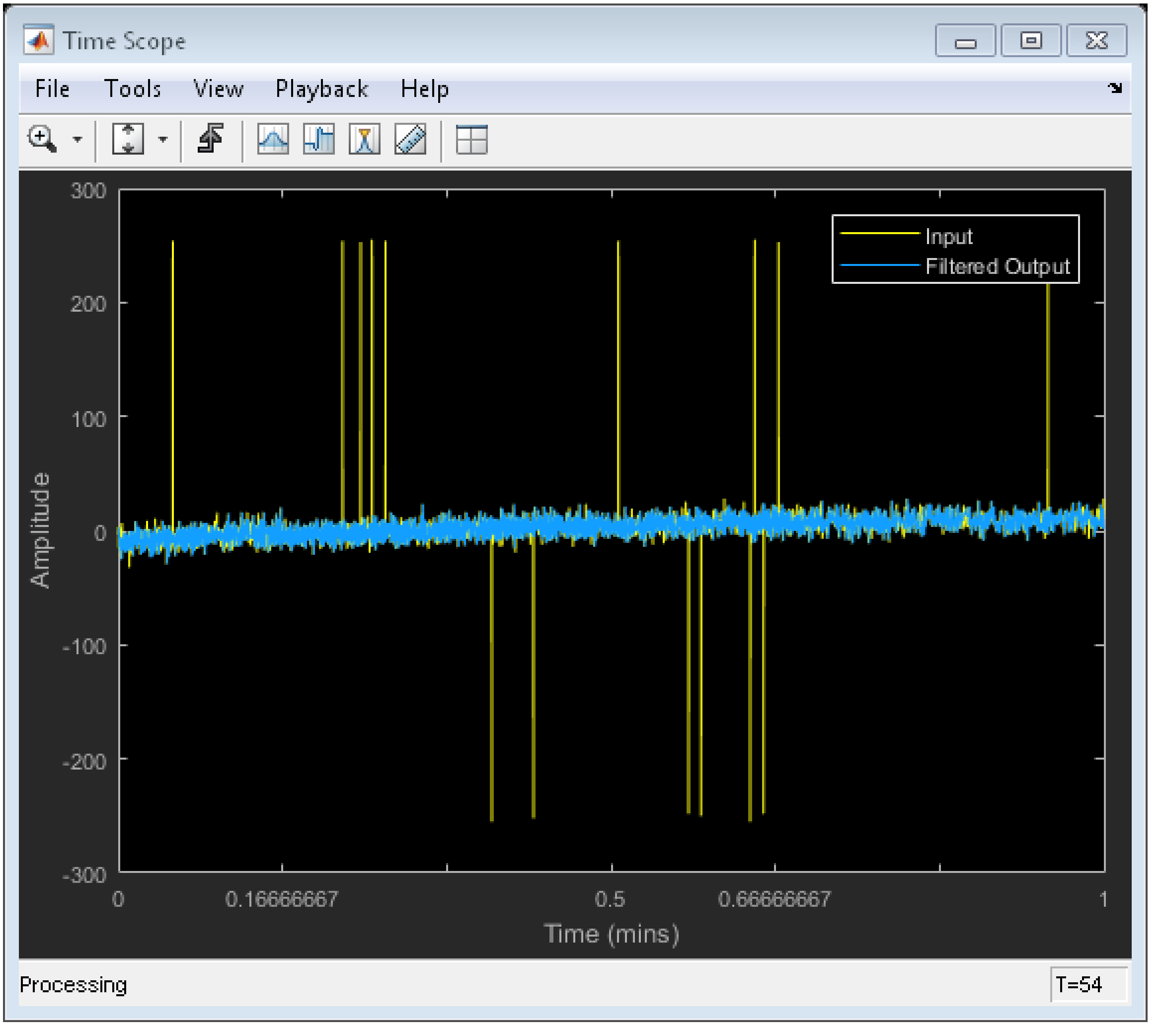
Remove the high-frequency outliers from a streaming signal using the `dsp.HampelFilter` System object™.

Use the `dsp.MatFileReader` System object to read the gyroscope MAT file. The file contains three columns of data, with each column containing 7140 samples. The three columns represent the *x*-axis, *y*-axis, and *z*-axis data from the gyroscope motion sensor. Choose a frame size of 714 samples so that each column of the data contains 10 frames. The `dsp.HampelFilter` System object uses a window length of 11. Create a `dsp.TimeScope` object to view the filtered output.

```
reader = dsp.MatFileReader('SamplesPerFrame',714,'Filename','LSM9DSHampelgyroData73.ma
    'VariableName','data');
hampFilt = dsp.HampelFilter(11);
scope = dsp.TimeScope('NumInputPorts',1,'SampleRate',119,'YLimits',[-300 300], ...
    'ChannelNames',{'Input','Filtered Output'},'TimeSpan',60,'ShowLegend',true);
```

Filter the gyroscope data using the `dsp.HampelFilter` System object. View the filtered *z*-axis data in the Time Scope.

```
for i = 1:10
    gyroData = reader();
    filteredData = hampFilt(gyroData);
    scope([gyroData(:,3),filteredData(:,3)]);
end
```


Time Scope
File
Tools
View
Playback
Help
Input
Filtered Output
300
200
100
0
-100
-200
-300
Amplitude
0
0.16666667
0.5
0.66666667
1
Time (mins)
Processing
T=54

The Hampel filter removes all the outliers and preserves the shape of the signal.

Definitions

Hampel Identifier

The Hampel identifier is a variation of the three-sigma rule of statistics that is robust against outliers.

Given a sequence x_1, x_2, x_3, …, x_n and a sliding window of length k, define point-to-point median and standard-deviation estimates using:

- Local median — $m_i = \text{median}\left(x_{i-k}, x_{i-k+1}, x_{i-k+2}, \ldots, x_i, \ldots, x_{i+k-2}, x_{i+k-1}, x_{i+k}\right)$
- Standard deviation — $\sigma_i = \kappa\, \text{median}\left(\left|x_{i-k} - m_i\right|, \ldots, \left|x_{i+k} - m_i\right|\right)$, where

$$\kappa = \frac{1}{\sqrt{2}\,\text{erfc}^{-1}\,1/2} \approx 1.4826$$

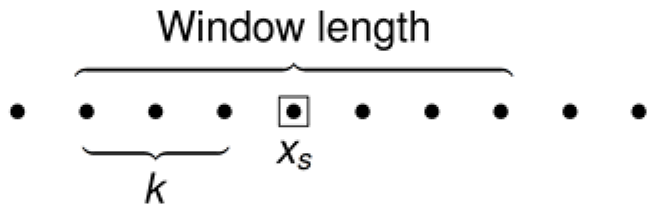
The quantity σ_i / κ is known as the median absolute deviation (MAD).

If a sample x_i is such that

$$\left|x_i - m_i\right| > n_\sigma \sigma_i$$

for a given threshold n_σ, then the Hampel identifier declares x_i an outlier and replaces it with m_i. If n_σ is 0, then the Hampel filter behaves as a regular median filter.

Algorithms



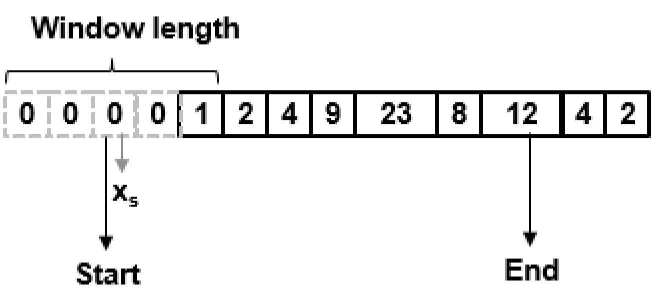
For a given sample of data, x_s, the algorithm:

- Centers the window of odd length at the current sample.
- Computes the local median, m_i, and standard deviation, σ_i, over the current window of data.
- Compares the current sample with $n_\sigma \times \sigma_i$, where n_σ is the threshold value. If $|x_s - m_i| > n_\sigma \times \sigma_i$, the filter identifies the current sample, x_s, as an outlier and replaces it with the median value, m_i.

Consider a frame of data that is passed into the Hampel filter.

1	2	4	9	23	8	12	4	2

In this example, the Hampel filter slides a window of length 5 (Len) over the data. The filter has a threshold value of 2 (n_σ). To have a complete window at the beginning of the frame, the filter algorithm prepends the frame with $Len - 1$ zeros. To compute the first sample of the output, the window centers on the $\left[\frac{Len-1}{2}+1\right]^{th}$ sample in the appended frame, the third zero in this case. The filter computes the median, median absolute deviation, and the standard deviation over the data in the local window.



- Current sample: $x_s = 0$.
- Window of data: win = [0 0 0 0 1].
- Local median: m_i = median([0 0 0 0 1]) = 0.

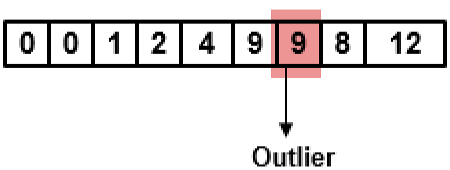
- Median absolute deviation: $mad_i = \text{median}\left(|x_{i-k} - m_i|, \ldots, |x_{i+k} - m_i|\right)$. For this window of data, $mad = \text{median}\left(|0-0|, \ldots, |1-0|\right) = 0$.
- Standard deviation: $\sigma_i = \kappa \times mad_i = 0$, where $\kappa = \frac{1}{\sqrt{2}\,\text{erfc}^{-1}\,1/2} \approx 1.4826$.
- The current sample, $x_s = 0$, does not obey the relation for outlier detection.

  $$\left[|x_s - m_i| = 0\right] > \left[(n_\sigma \times \sigma_i) = 0\right]$$

  Therefore, the Hampel filter outputs the current input sample, $x_s = 0$.

Repeat this procedure for every succeeding sample until the algorithm centers the window on the $\left[End - \frac{Len-1}{2}\right]^{th}$ sample, marked as `End`. Because the window centered on the last $\frac{Len-1}{2}$ samples cannot be full, these samples are processed with the next frame of input data.

Here is the first output frame the Hampel filter generates:



The seventh sample of the appended input frame, 23, is an outlier. The Hampel filter replaces this sample with the median over the local window [4 9 23 8 12].

References

[1] Bodenham, Dean. "Adaptive Filtering and Change Detection for Streaming Data." PH.D. Thesis. Imperial College, London, 2012.

[2] Liu, Hancong, Sirish Shah, and Wei Jiang. "On-line outlier detection and data cleaning." *Computers and Chemical Engineering*. Vol. 28, March 2004, pp. 1635–1647.

[3] Suomela, Jukka. Median Filtering Is Equivalent to Sorting, 2014.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

Functions
`hampel`

System Objects
dsp.MedianFilter | dsp.MovingAverage

Blocks
`Hampel Filter` | `Median Filter` | `Moving Average`

Introduced in R2017a

reset

System object: dsp.HampelFilter
Package: dsp

Reset internal states of System object

Syntax

`reset(hampFilt)`

Description

`reset(hampFilt)` resets the states of the input object to their initial values. The initial state values correspond to the initial conditions for the difference equation defining the object. After you call the object using nonzero input data, the filter might have nonzero states. If you continue to pass nonzero input data to the object without first calling `reset`, the object might produce different outputs for an identical input.

Introduced in R2017a

step

System object: dsp.HampelFilter
Package: dsp

Remove outliers from input

Syntax

```
y = step(hampFilt,x)
[y,isOutlier] = step(hampFilt,x)
```

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(hampFilt,x)` detects and removes the outliers of each channel of the input signal, `x`, independently over time using the input `dsp.HampelFilter` System object, `hampFilt`. The input `x` must be real, and it supports single and double data types.

`[y,isOutlier] = step(hampFilt,x)` returns a logical array, `IsOutlier`, where each element is `true` when the corresponding element in the input is an outlier. `isOutlier` is the same size as the input and output vectors.

Introduced in R2017a

dsp.HighpassFilter System object

Package: dsp

FIR or IIR highpass filter

Description

`dsp.HighpassFilter` independently filters each channel of the input over time using the given design specifications. You can set the `FilterType` property of `dsp.HighpassFilter` to `'FIR'` or `'IIR'` to implement the object as a FIR or IIR highpass filter. This object supports fixed-point operations, HDL code generation, and ARM Cortex code generation.

To filter each channel of your input:

1. Define and set up your highpass filter. See “Construction” on page 4-944.
2. Call step to filter each channel of the input signal according to the properties of `dsp.HighpassFilter`. The input signal can be a real- or complex-valued column vector or matrix, with floating point or fixed-point precision. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The number of rows in the input signal denote the channel length.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`HPF = dsp.HighpassFilter` returns a minimum order FIR highpass filter, `HPF`, with the default filter settings. Calling `step` with the default property settings filters the input data with a stopband frequency of `8` kHz, a passband frequency of `12` kHz, a stopband attenuation of `80` dB, and a passband ripple of `0.1` dB.

`HPF = dsp.HighpassFilter(Name,Value)` returns a highpass filter, with additional properties specified by one or more `Name,Value` pair arguments. `Name` is the property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Properties

`SampleRate` — Input sample rate

`44100` (default) | positive real scalar

Input sample rate in Hz, specified as the comma-separated pair consisting of `'SampleRate'` and a positive real scalar.

`FilterType` — Filter type

`'FIR'` (default) | `'IIR'`

Filter type, specified as one of the following options:

- `'FIR'` — The object designs an FIR highpass filter.
- `'IIR'` — The object designs an IIR highpass (biquad) filter.

`DesignForMinimumOrder` — Minimum order filter design

`true` (default) | `false`

Minimum order filter design, specified as the comma-separated pair consisting of `'DesignForMinimumOrder'` and a logical value. If this property is `true`, then `dsp.HighpassFilter` designs filters with the minimum order that meets the passband frequency, stopband frequency, passband ripple, and stopband attenuation specifications. Set these specifications using the corresponding properties. If this property is `false`, then the object designs filters with the order that you specify in the `FilterOrder` property. This filter design meets the passband frequency, passband ripple, and stopband attenuation specifications that you set using the respective properties.

`FilterOrder` — Order of the FIR or IIR filter

`50` (default) | positive integer scalar

Order of the FIR or IIR filter, specified as the comma-separated pair consisting of `'FilterOrder'` and a positive integer scalar. Specifying a filter order is only valid when the value of `'DesignForMinimumOrder'` is `false`.

`StopbandFrequency` — Filter stopband edge frequency

`8000` (default) | real positive scalar

Filter stopband edge frequency in Hz, specified as the comma-separated pair consisting of `'StopbandFrequency'` and a real positive scalar. The value of the stopband edge frequency in Hz must be less than the passband frequency. You can specify the stopband edge frequency only when `'DesignForMinimumOrder'` is `true`.

`PassbandFrequency` — Filter passband edge frequency
`12000` (default) | real positive scalar

Filter passband edge frequency in Hz, specified as the comma-separated pair consisting of `'PassbandFrequency'` and a real positive scalar. The value of the passband edge frequency in Hz must be less than half the `SampleRate` and greater than `StopbandFrequency`.

`StopbandAttenuation` — Minimum attenuation in the stopband
`80` (default) | real positive scalar

Minimum attenuation in the stopband in dB, specified as the comma-separated pair consisting of `'StopbandAttenuation'` and a real positive scalar. Minimum attenuation in the stopband defaults to `80` dB.

`PassbandRipple` — Maximum ripple of filter response in the passband
`0.1` (default) | real positive scalar

Maximum ripple of filter response in the passband, in dB, specified as the comma-separated pair consisting of `'PassbandRipple'` and a real positive scalar. Maximum ripple of filter response defaults to `0.1` dB.

Fixed-Point Properties

`RoundingMethod` — Rounding method for output fixed-point operations
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Simplest'` | `'Zero'`

Rounding method for output fixed-point operations, specified as a character vector. For more information on the rounding modes, see “Precision and Range”.

`CoefficientsDataType` — Word and fraction lengths of coefficients
`numerictype([],16)` (default) | `numerictype` object

Word and fraction lengths of coefficients, specified as a `numerictype` object. The default, `numerictype(1,16)` corresponds to a signed numeric type object with 16-bit coefficients and a fraction length determined based on the coefficient values, to give the best possible precision.

This property is not tunable.

Word length of the output is same as the word length of the input. Fraction length of the output is computed such that the entire dynamic range of the output can be represented without overflow. For details on how the fraction length of the output is computed, see "Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters".

Methods

measure	Measure frequency response characteristics of `dsp.HighpassFilter` System object.
reset	Reset internal states of highpass filter
step	Filter input using FIR or IIR highpass filter

For additional methods, see "Analysis Methods for Filter System Objects" on page 3-2.

For a complete list of analysis methods supported for the `dsp.HighpassFilter` object, enter `dsp.HighpassFilter.helpFilterAnalysis` at the MATLAB command prompt.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Impulse and Frequency Response of FIR and IIR Highpass Filters

Create a minimum order FIR highpass filter for data sampled at 44.1 kHz. Specify a passband frequency of 12 kHz, a stopband frequency of 8 kHz, a passband ripple of 0.1 dB, and a stopband attenuation of 80 dB.

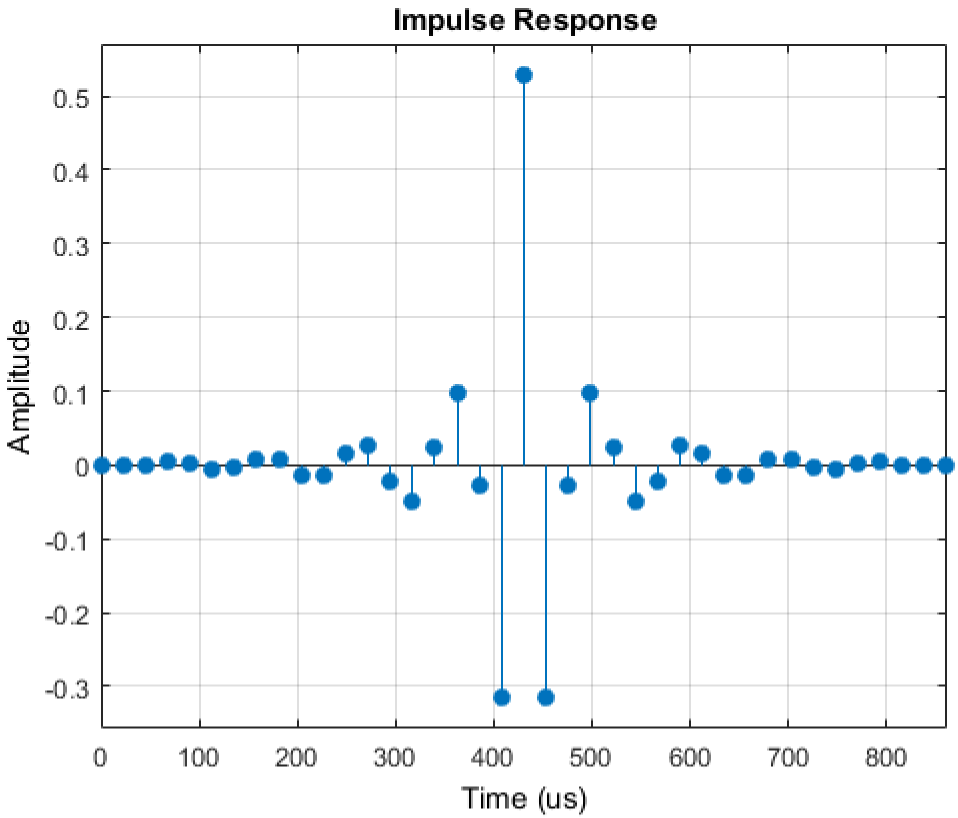
```
Fs = 44.1e3;
filtertype = 'FIR';
Fpass = 12e3;
Fstop = 8e3;
Rp = 0.1;
Astop = 80;
FIRHPF = dsp.HighpassFilter('SampleRate',Fs,...
                             'FilterType',filtertype,...
                             'PassbandFrequency',Fpass,...
                             'StopbandFrequency',Fstop,...
                             'PassbandRipple',Rp,...
                             'StopbandAttenuation',Astop);
```

Design a minimum order IIR highpass filter with the same properties as the FIR highpass filter. Use `clone` to create a system object with the same properties as the FIR Highpass filter. Change the `FilterType` property of the cloned filter to `IIR`.

```
IIRHPF = clone(FIRHPF);
IIRHPF.FilterType = 'IIR';
```

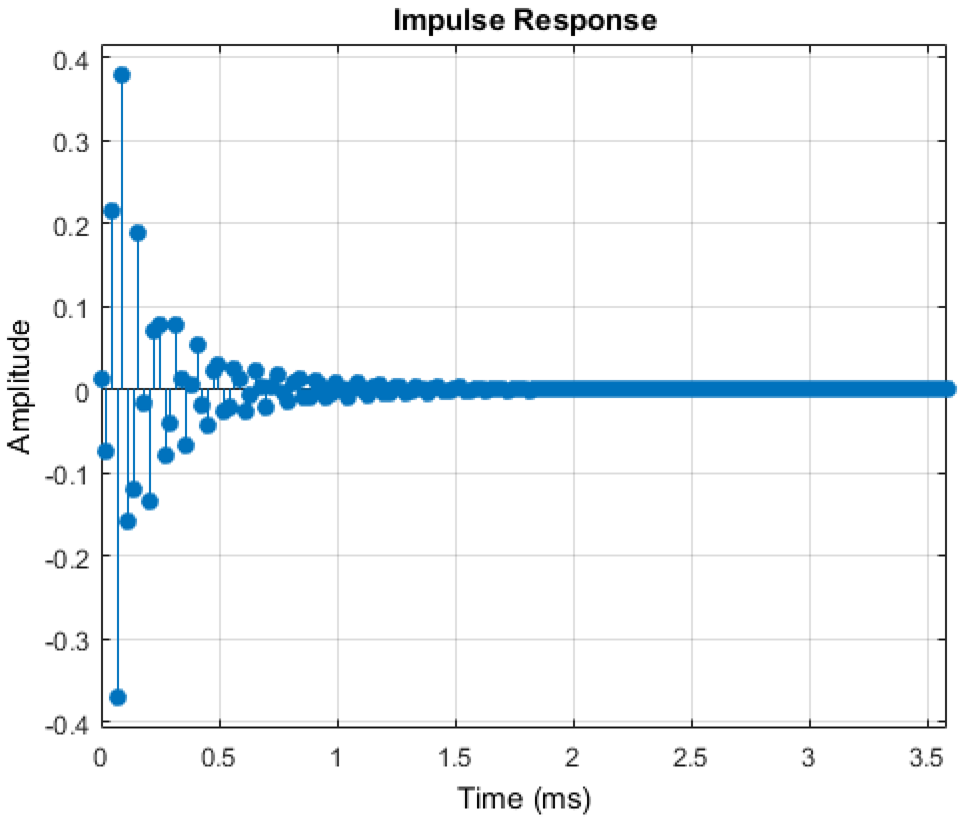
Plot the impulse response of the FIR highpass filter. The zeroth order coefficient is delayed by 19 samples, which is equal to the group delay of the filter. The FIR highpass filter is a causal FIR filter

```
fvtool(FIRHPF,'Analysis','impulse')
```



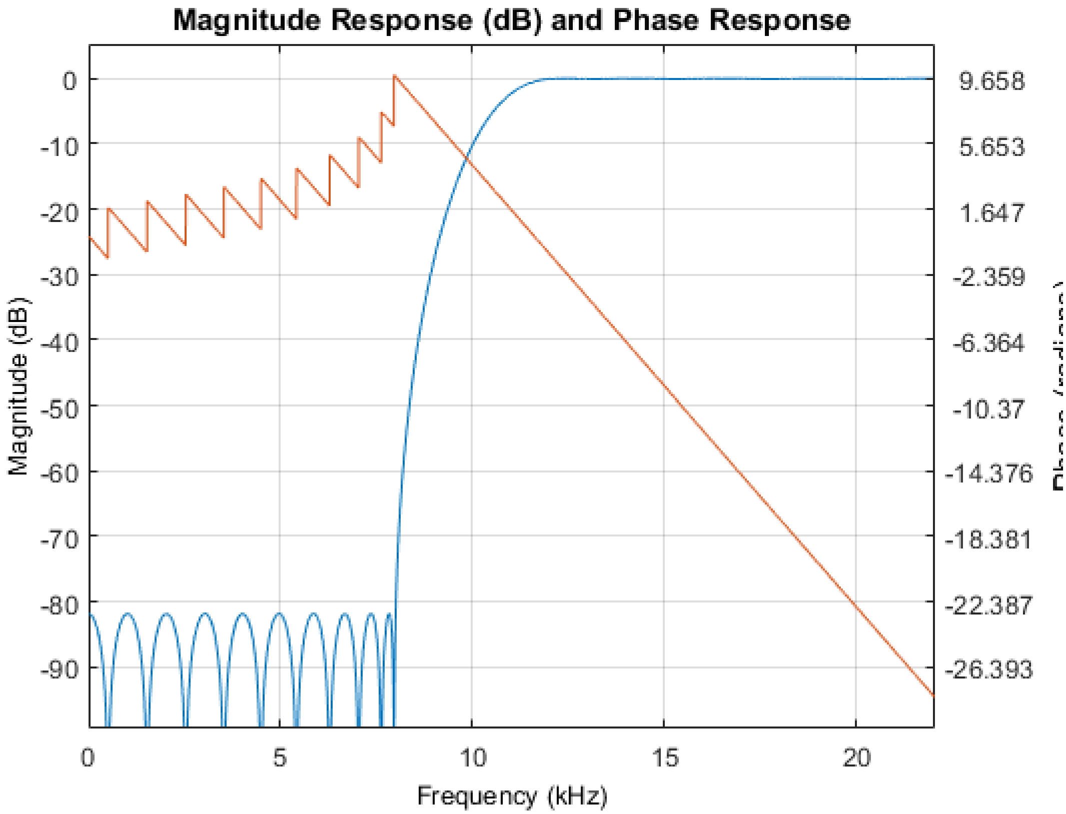
Plot the impulse response of the IIR highpass filter.

```
fvtool(IIRHPF,'Analysis','impulse')
```



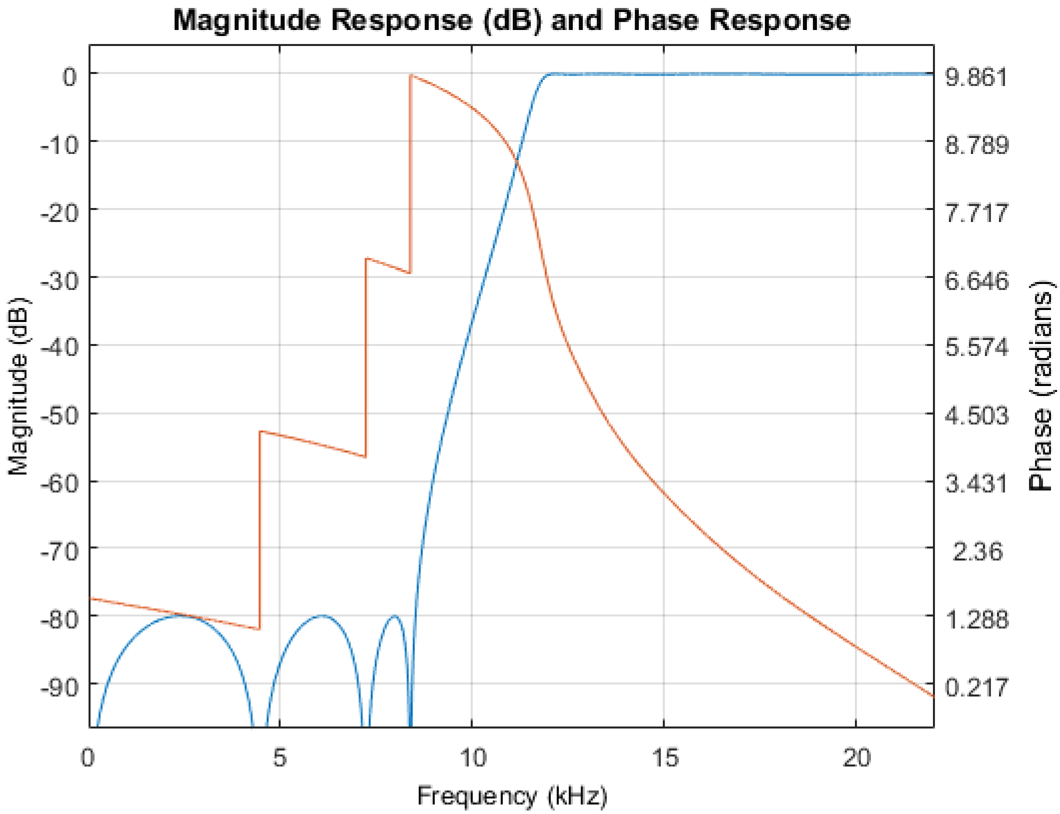
Plot the magnitude and phase response of the FIR highpass filter.

```
fvtool(FIRHPF,'Analysis','freq')
```



Plot the magnitude and phase response of the IIR highpass filter.

```
fvtool(IIRHPF,'Analysis','freq')
```



Calculate the cost of implementing the FIR highpass filter.

```
cost(FIRHPF)
```

```
ans = 

  struct with fields:

                NumCoefficients: 39
                      NumStates: 38
    MultiplicationsPerInputSample: 39
          AdditionsPerInputSample: 38
```

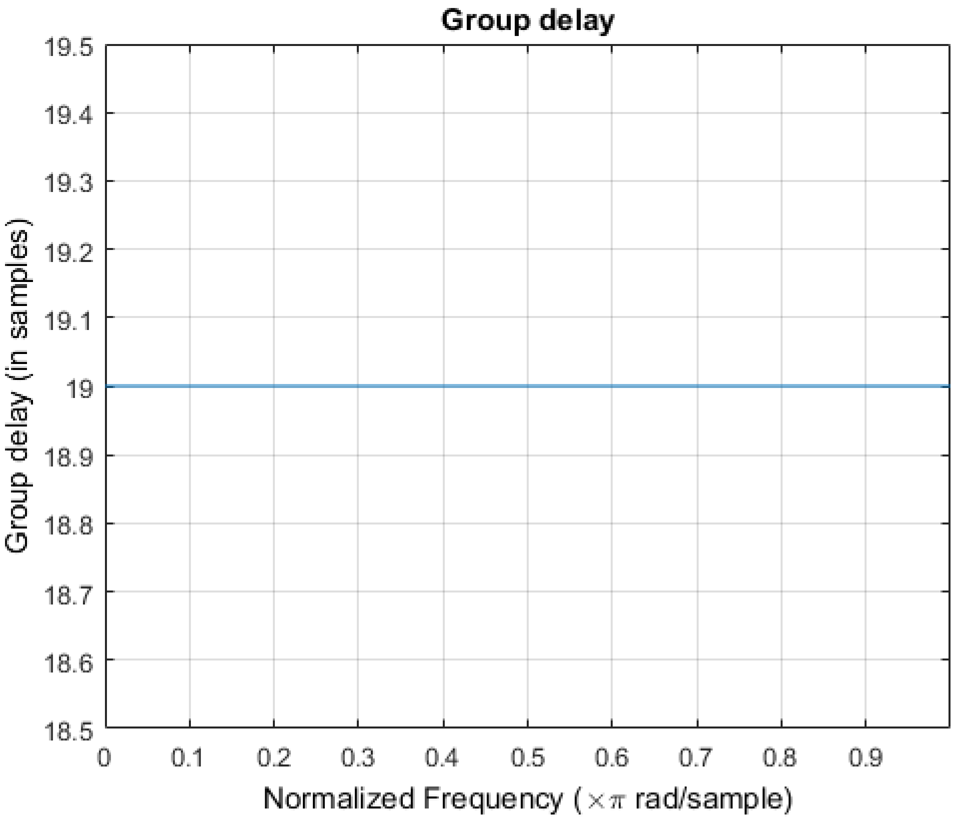
Calculate the cost of implementing the IIR highpass filter. The IIR filter is more efficient to implement than its FIR counterpart.

```
cost(IIRHPF)
```

```
ans = 

  struct with fields:

                  NumCoefficients: 18
                        NumStates: 14
    MultiplicationsPerInputSample: 18
          AdditionsPerInputSample: 14
```

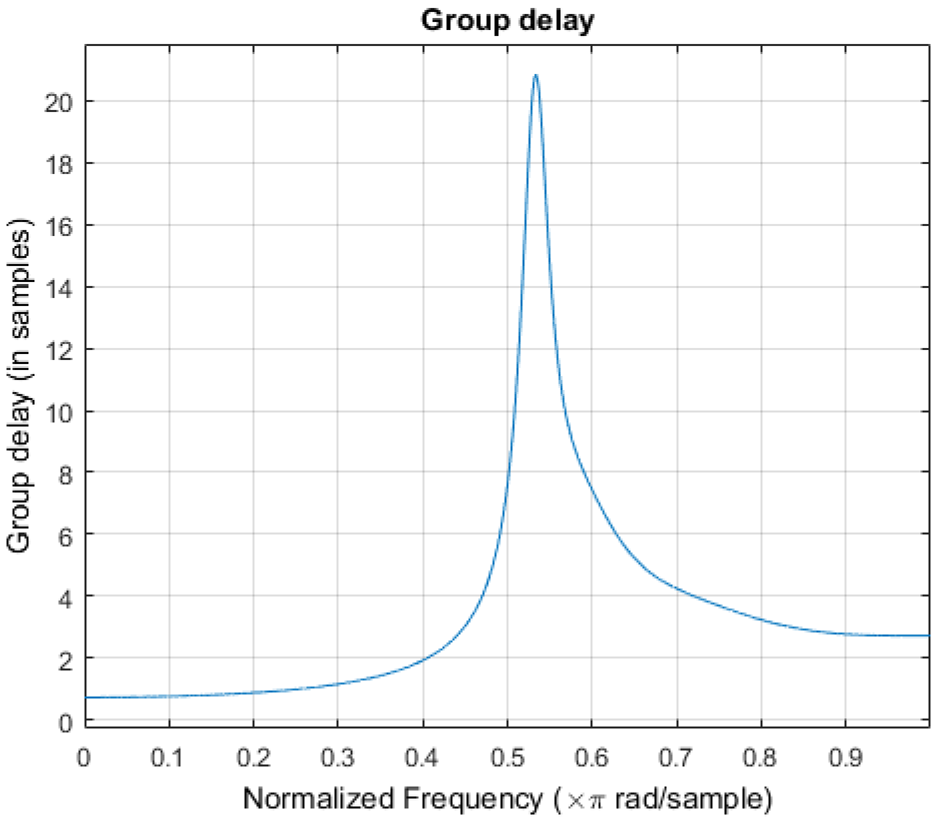
Calculate the group delay of the FIR highpass filter.

```
grpdelay(FIRHPF)
```



Calculate the group delay of the IIR highpass filter. The FIR filter has a constant group delay (linear phase) while its IIR counterpart does not.

```
grpdelay(IIRHPF)
```



Filter White Gaussian Noise with an IIR Highpass filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

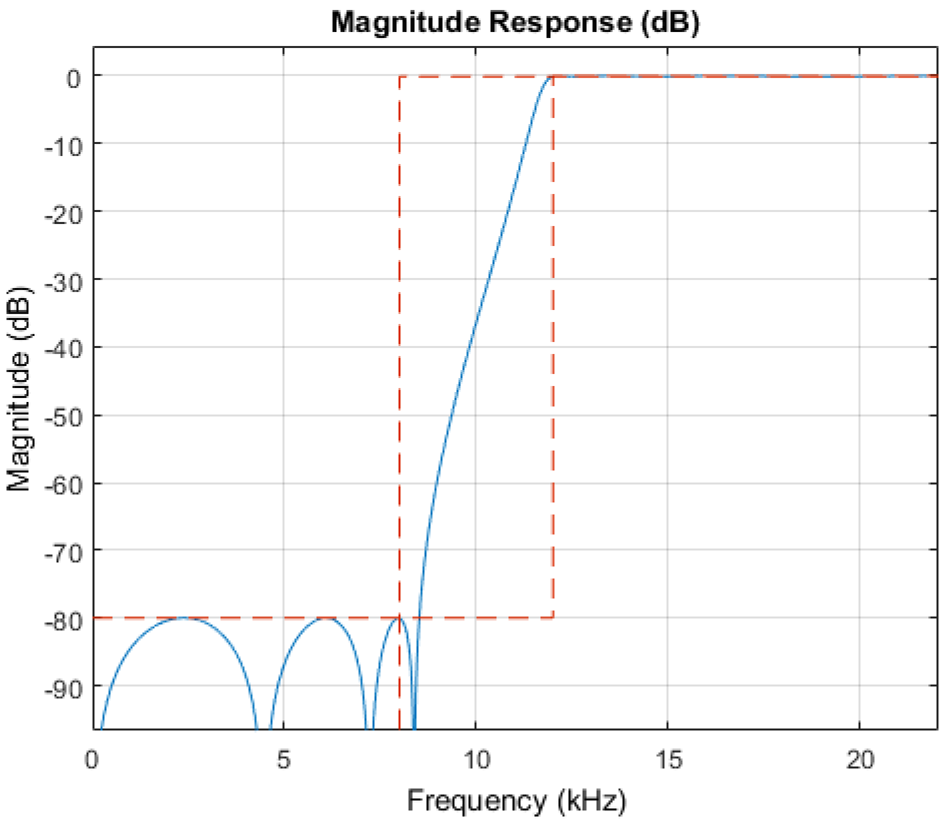
Set up the IIR highpass filter. The sampling rate of the white Gaussian noise is 44,100 Hz. The passband frequency of the filter is 12 kHz, the stopband frequency is 8 kHz, the passband ripple is 0.1 dB, and the stopband attenuation is 80 dB.

```
Fs = 44.1e3;
filtertype = 'IIR';
Fpass = 12e3;
Fstop = 8e3;
```

```
Rp = 0.1;
Astop = 80;
hpf = dsp.HighpassFilter('SampleRate',Fs,...
                             'FilterType',filtertype,...
                             'PassbandFrequency',Fpass,...
                             'StopbandFrequency',Fstop,...
                             'PassbandRipple',Rp,...
                             'StopbandAttenuation',Astop);
```

View the magnitude response of the highpass filter.

```
fvtool(hpf)
```

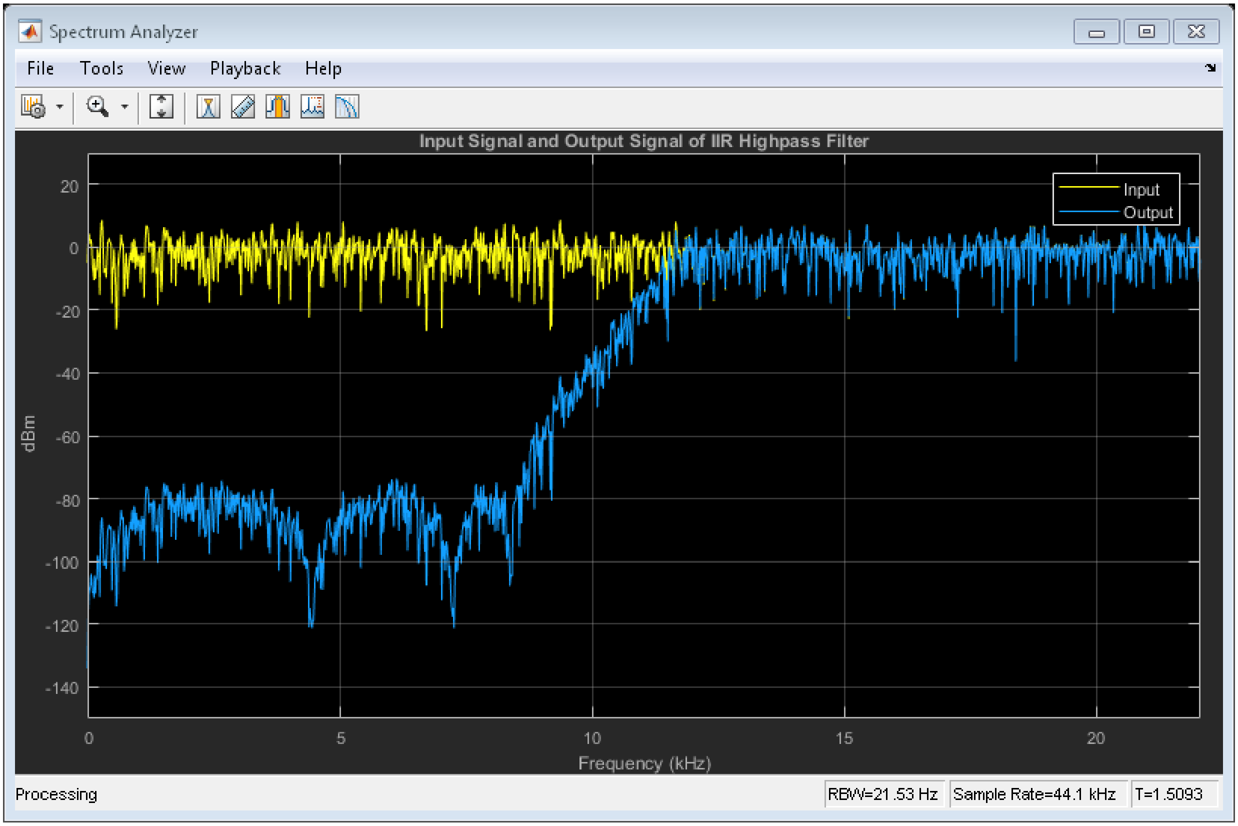


Create a spectrum analyzer object.

```
sa = dsp.SpectrumAnalyzer('SampleRate',44.1e3,...
    'PlotAsTwoSidedSpectrum',false,'ShowLegend',true,'YLimits',...
           [-150 30],...
           'Title',...
           'Input Signal and Output Signal of IIR Highpass Filter');
sa.ChannelNames = {'Input','Output'};
```

Filter the white Gaussian noisy input signal. View the input and output signals using the spectrum analyzer.

```
for k = 1:100
    Input = randn(1024,1);
    Output = hpf(Input);
    sa([Input,Output]);
end
```



Supported Data Types

Argument	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point
Output	• Double-precision floating point • Single-precision floating point • Fixed point

Algorithms

FIR Highpass Filter

When the `FilterType` property is set to `'FIR'`, the `dsp.HighpassFilter` object acts as a FIR highpass filter. In this configuration, `dsp.HighpassFilter` is an alternative to using `firceqrip` and `firgr` with `dsp.FIRFilter`. This object condenses the two-step process into one. For the minimum order design, the object uses generalized Remez FIR filter design algorithm. For the specified order design, the object uses constrained equiripple FIR filter design algorithm. The designed filter is then implemented as a linear phase Type-1 filter with a `Direct form` structure. You can use dsp.HighpassFilter.measure to verify that the design meets the prescribed specifications.

IIR Highpass Filter

When the `FilterType` property is set to `'IIR'`, the `dsp.HighpassFilter` object acts as an IIR highpass filter. In this configuration, this object uses the elliptic design method to compute the SOS and scale values required to meet the filter design specifications. The object uses the SOS and scale values to set up a `Direct form I` biquadratic IIR filter, which forms the basis of the IIR version of the `dsp.HighpassFilter` System object. You can use dsp.HighpassFilter.measure to verify that the design meets the prescribed specifications.

References

[1] Shpak, D.J., and A. Antoniou. "A generalized Remez method for the design of FIR digital filters." *IEEE Transactions on Circuits and Systems*. Vol. 37, Issue 2, Feb. 1990, pp. 161–174.

[2] Selesnick, I.W., and C. S. Burrus. "Exchange algorithms that complement the Parks-McClellan algorithm for linear-phase FIR filter design." *IEEE Transactions on Circuits and Systems*. Vol. 44, Issue 2, Feb. 1997, pp. 137–143.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

dsp.LowpassFilter

Introduced in R2015a

measure

System object: dsp.HighpassFilter
Package: dsp

Measure frequency response characteristics of `dsp.HighpassFilter` System object.

Syntax

```
HPFMeas = measure(HPF)
```

Description

`HPFMeas = measure(HPF)` measures the frequency response characteristics of the `dsp.HighpassFilter` System object, `HPF`.

Input Arguments

`HPF` — FIR or IIR highpass filter

`HighpassFilter` System object

FIR or IIR highpass filter, specified as a `HighpassFilter` System object.

Output Arguments

`HPFMeas` — Measurements of frequency response characteristics of the highpass filter

`HighpassFilter` System object

Measurements of frequency response characteristics of the highpass filter, returned as an `fdesign.highpassmeas` object.

Examples

Measure Frequency Response Characteristics of Highpass Filter

Measure the frequency response characteristics of a highpass filter. Create a `dsp.HighpassFilter` System object with default properties. Measure the frequency response characteristics of the filter.

```
HPF = dsp.HighpassFilter
HPFMeas = measure(HPF)
```

```
HPF =

  dsp.HighpassFilter with properties:

               FilterType: 'FIR'
    DesignForMinimumOrder: true
        StopbandFrequency: 8000
        PassbandFrequency: 12000
      StopbandAttenuation: 80
           PassbandRipple: 0.1000
               SampleRate: 44100

  Use get to show all properties


HPFMeas =

Sample Rate      : 44.1 kHz
Stopband Edge    : 8 kHz
6-dB Point       : 10.418 kHz
3-dB Point       : 10.8594 kHz
Passband Edge    : 12 kHz
Stopband Atten.  : 81.8558 dB
Passband Ripple  : 0.08066 dB
Transition Width : 4 kHz
```

Introduced in R2015a

reset

System object: dsp.HighpassFilter
Package: dsp

Reset internal states of highpass filter

Syntax

`reset(HPF)`

Description

`reset(HPF)` resets the filter states of the highpass filter, `HPF`, to their initial values. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the highpass filter to nonzero input data, the filter might have nonzero states. If you Invoke the `step` method again without first invoking the `reset` method, the object might produce different outputs for an identical input.

Input Arguments

`HPF` — FIR or IIR highpass filter

`HighpassFilter` System object

FIR or IIR highpass filter, specified as a `HighpassFilter` System object.

Examples

Reset a Highpass Filter

Create a highpass filter with default properties.

```
HPF = dsp.HighpassFilter;
```

Create a two-channel random signal. Apply the `step` method twice on the signal.

```
x = randn(10,2);

y1 = step(HPF,x);
y2 = step(HPF,x);

no = all(y2==y1)


no =

  1×2 logical array

   0   0
```

The output is different because the internal states of HPF have changed. Reset the highpass filter and apply step again. Verify that the output is unchanged.

```
reset(HPF)

y3 = step(HPF,x);

yes = all(y3==y1)


yes =

  1×2 logical array

   1   1
```

step

System object: dsp.HighpassFilter
Package: dsp

Filter input using FIR or IIR highpass filter

Syntax

`Y = step(HPF,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(HPF,X)` filters the real or complex input signal, `X`, using the highpass filter, `HPF`. `Y` is a highpass-filtered version of `X`. The input `X` must be a column vector or matrix. If `X` is a matrix, each column is treated as an independent channel. `X` can be a floating point or a fixed-point input.

Input Arguments

`HPF` — FIR or IIR highpass filter

`HighpassFilter` System object

FIR or IIR highpass filter, specified as a `HighpassFilter` System object.

`X` — Input signal

vector | matrix

Input signal, specified as a vector or matrix with floating point or fixed-point precision. The row length of `X` is the frame size, that is, channel length. Each column of `X` is treated as a separate channel. The column length of `X` is the number of channels.

Output Arguments

Y — Highpass-filtered signal

vector | matrix

Highpass-filtered signal, returned as a vector or matrix of the same size, data type, and complexity as the input signal, X.

Examples

Filter White Gaussian Noise Signal With a Highpass Filter

Create a highpass filter with default properties.

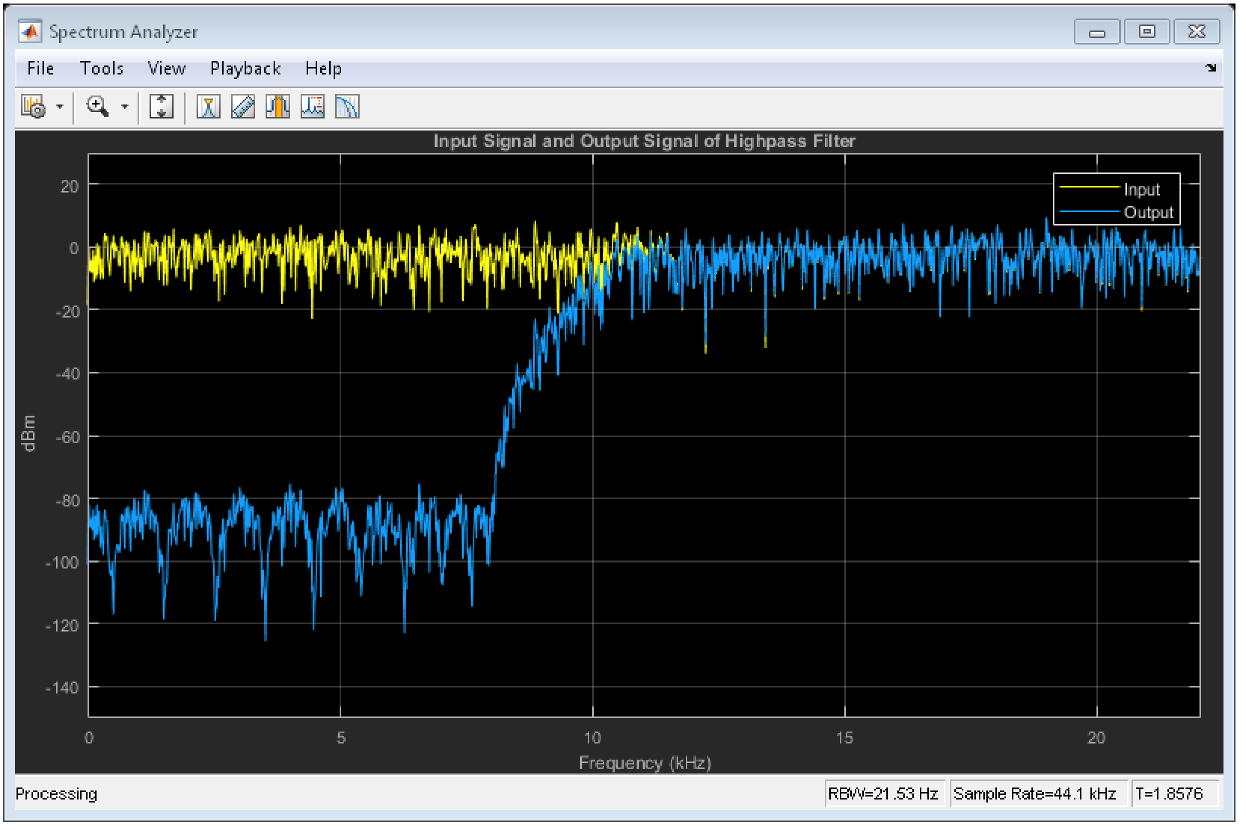
```
HPF = dsp.HighpassFilter;
```

Create a spectrum analyzer object.

```
hSA = dsp.SpectrumAnalyzer('SampleRate',44.1e3,...
    'PlotAsTwoSidedSpectrum',false,'ShowLegend',true,'YLimits',...
           [-150 30],...
           'Title',...
           'Input Signal and Output Signal of Highpass Filter');
hSA.ChannelNames = {'Input','Output'};
```

Implement `step` on `HPF` to filter the white Gaussian noisy input signal. View the input and output signals using the spectrum analyzer.

```
for k = 1:100
    Input = randn(1024,1);

    Output = step(HPF,Input);

    step(hSA,[Input,Output]);
end
```



Introduced in R2015a

dsp.Histogram System object

Package: dsp

Histogram of input or sequence of inputs

Description

The `Histogram` object generates a histogram for an input or a sequence of inputs.

To generate a histogram for an input or a sequence of inputs:

1. Define and set up your histogram object. See "Construction" on page 4-968.
2. Call step to generate the histogram for an input according to the properties of `dsp.Histogram`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`hist = dsp.Histogram` returns a histogram object, `hist`, that computes the frequency distribution of the elements in each input matrix.

`hist = dsp.Histogram('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a histogram object, `hist`, with each specified property set to the specified value.

`hist =`
`dsp.Histogram(`*`MIN`*`,`*`MAX`*`,`*`NUMBINS`*`,'`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a histogram object, `hist`, with the `LowerLimit` property set to *`MIN`*, `UpperLimit` property set to *`MAX`*, `NumBins` property set to *`NUMBINS`* and other specified properties set to the specified values.

Properties

LowerLimit

Lower boundary

Specify the lower boundary of the lowest-valued bin as a real-valued scalar. `NaN` and `Inf` are not valid values for this property. The default is `0`. This property is tunable.

UpperLimit

Upper boundary

Specify the upper boundary of the highest-valued bin as a real-valued scalar. `NaN` and `Inf` are not valid values for this property. The default is `10`. This property is tunable.

NumBins

Number of bins in histogram

Specify the number of bins in the histogram. The default is `11`.

Dimension

Specify how the histogram calculation is performed over the data as one of `| All | Column |`. The default is `Column`.

Normalize

Enable output vector normalization

Specify whether the histogram object normalizes the output vector, v, so that $sum(v) = 1$. When you set this property to `true`, the output vector is normalized. When you set it to `false`, the object supports fixed-point operations and does not use this property for normalization. The default is `false`.

RunningHistogram

Enable calculation over successive `step` method calls

Set this property to `true` to enable running histogram calculations for the input elements over successive calls to the `step` method. Set this property to `false` to compute a histogram for the current input. The default is `false`.

ResetInputPort

Enable resetting in running histogram mode

Set this property to true to enable resetting the running histogram. When you set the property to true, specify a reset input to the step method that resets the running histogram. This property applies when you set the RunningHistogram on page 4-969 property to true. When this property is false, the histogram object does not reset. The default is false.

ResetCondition

Reset condition for running histogram mode

Specify the event that resets the running histogram as one of | Rising edge | Falling edge | Either edge | Non-zero |. This property applies when you set the ResetInputPort property to true. The default is Non-zero

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of | Same as input | Custom |. The default is Same as input.

CustomProductDataType

Custom product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType on page 4-970` property to `Custom`. The default is `numerictype([],32,30)`.

`AccumulatorDataType`

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `|` `Same as product` `|` `Same as input` `|` `Custom` `|`. The default is `Same as input`.

`CustomAccumulatorDataType`

Custom accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType on page 4-971` property to `Custom`. The default is `numerictype([],32,30)`.

Methods

reset	Reset histogram bin values to zero
step	Histogram for input data

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Compute Histogram of a Sequence

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Compute a histogram with four bins, for possible input values 1 through 4.

```
hist = dsp.Histogram(1,4,4);
y = hist([1 2 2 3 3 3 4 4 4 4]');
```

y is equal to [1; 2; 3; 4] - one ones, two twos, etc.

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Histogram` block reference page. The object properties correspond to the block parameters, except:

- The **Reset port** block parameter corresponds to both the `ResetCondition` and the `ResetInputPort` object properties.
- The **Find histogram over** block parameter corresponds to the Dimension on page 4-969 property of the object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This object has no tunable properties for code generation.
- See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

dsp.Minimum | dsp.Maximum | dsp.Mean

Introduced in R2012a

reset

System object: dsp.Histogram
Package: dsp

Reset histogram bin values to zero

Syntax

```
reset(hist)
```

Description

`reset(hist)` sets the `Histogram` object bin values to zero when you set the `RunningHistogram` property to `true`.

step

System object: dsp.Histogram
Package: dsp

Histogram for input data

Syntax

```
Y = step(hist,X)
Y = step(hist,X,R)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(hist,X)` returns a histogram `Y` for the input data `X` . When the `RunningHistogram` property is `true`, `Y` corresponds to the histogram of the input elements over successive calls to the `step` method.

`Y = step(hist,X,R)` resets the histogram state based on the value of `R` and the object's `ResetCondition` property. You can reset the histogram state only when the `RunningHistogram` and the `ResetInputPort` properties are `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.IDCT System object

Package: dsp

Inverse discrete cosine transform (IDCT)

Description

The `IDCT` object computes the inverse discrete cosine transform (IDCT) of an input.

To compute the IDCT of an input:

1. Define and set up your IDCT object. See "Construction" on page 4-975.
2. Call step to compute the IDCT of an input according to the properties of `dsp.IDCT`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`idct = dsp.IDCT` returns a inverse discrete cosine transform (IDCT) object, `idct`. This object computes the IDCT of a real or complex input signal using the `Table lookup` method.

`idct = dsp.IDCT('PropertyName',PropertyValue,...)` returns an inverse discrete cosine transform (IDCT) object, `idct`, with each property set to the specified value.

Properties

`SineComputation`

Method to compute sines and cosines

Specify how the IDCT object computes the trigonometric function values as `Trigonometric function` or `Table lookup`. You must set this property to `Table lookup` for fixed-point inputs. The default is `Table lookup`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling`, `Convergent`, `Floor` , `Nearest`, `Round`, `Simplest`, or `Zero`. This property applies when you set the `SineComputation` on page 4-975 property to `Table lookup`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` or `Saturate`. This property applies when you set the `SineComputation` on page 4-975 property to `Table lookup`.

SineTableDataType

Sine table word-length designation

Specify the sine table fixed-point data type as one of `Same word length as input` or `Custom`. This property applies when you set the `SineComputation` on page 4-975 property to `Table lookup`.

CustomSineTableDataType

Sine table word length

Specify the sine table fixed-point type as a signed, unscaled `numerictype` object. This property applies when you set the `SineComputation` on page 4-975 property to `Table lookup` and the `SineTableDataType` on page 4-976 property to `Custom`. The default is `numerictype(true,16)`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of `Full precision`, `Same as input`, `Custom`. This property applies when you set the `SineComputation` on page 4-975 property to `Table lookup`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a signed, scaled `numerictype` object. This property applies when you set the `SineComputation` on page 4-975 property to `Table lookup` and the `ProductDataType` on page 4-976 property to `Custom`. The default is `numerictype(true,32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Full precision`, `Same as input`, `Same as product`, or `Custom`. This property applies when you set the `SineComputation` on page 4-975 property to `Table lookup`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a signed, scaled `numerictype` object. This property applies when you set the `SineComputation` on page 4-975 property to `Table lookup` and the `AccumulatorDataType` on page 4-977 property to `Custom`. The default is `numerictype(true,32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of `Full precision` , `Same as input`, `Custom`. This property applies when you set the `SineComputation` on page 4-975 property to `Table lookup`. The default is `Full precision`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a signed, scaled `numerictype` object. This property applies when you set the `SineComputation` on page 4-975 property to `Table lookup` and the `OutputDataType` on page 4-977 property to `Custom`. The default is`numerictype(true,16,15)`.

Methods

step	Inverse discrete cosine transform (IDCT) of input

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Analyze the Energy Content in a Sequence

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Use DCT to analyze the energy content in a sequence:

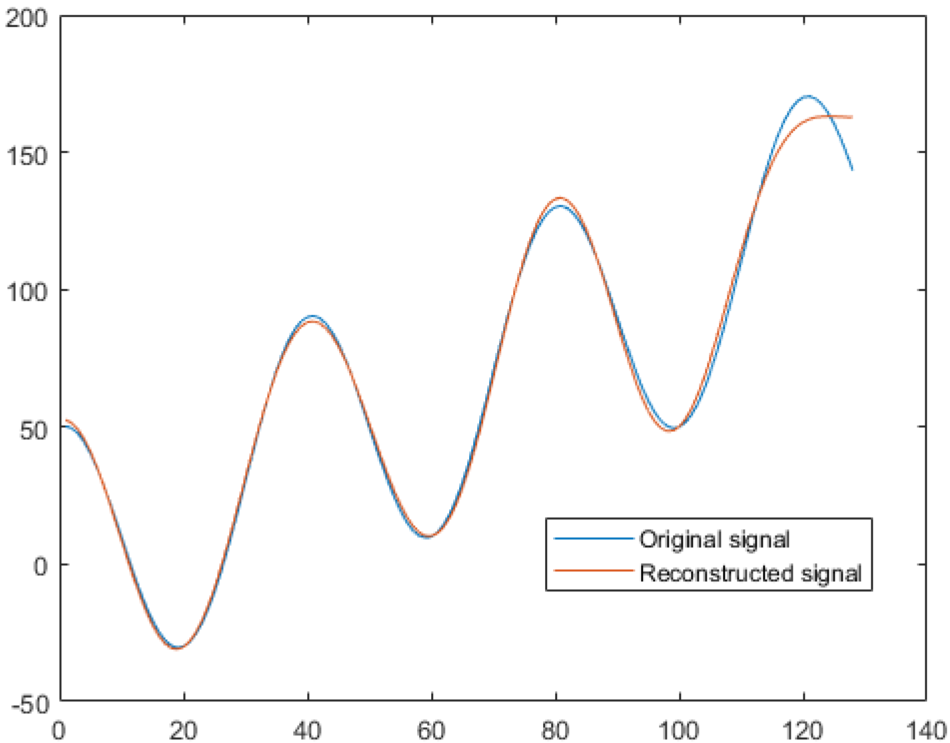
```
x = (1:128).' + 50*cos((1:128).'*2*pi/40);
dct = dsp.DCT;
X = dct(x);
```

Set the DCT coefficients which represent less than 0.1% of the total energy to 0 and reconstruct the sequence using IDCT.

```
[XX, ind] = sort(abs(X),1,'descend');
ii = 1;
while (norm([XX(1:ii);zeros(128-ii,1)]) <= 0.999*norm(XX))
    ii = ii+1;
end
disp(['Number of DCT coefficients that represent 99.9%',...
    'of the total energy in the sequence: ',num2str(ii)]);
XXt = zeros(128,1);
XXt(ind(1:ii)) = X(ind(1:ii));
```

```
idct = dsp.IDCT;
xt = idct(XXt);
plot(1:128,[x xt]);
legend('Original signal','Reconstructed signal',...
    'location','best');
```

```
Number of DCT coefficients that represent 99.9%of the total energy in the sequence: 10
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `IDCT` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.DCT | dsp.IFFT. | dsp.FFT

Introduced in R2012a

step

System object: dsp.IDCT
Package: dsp

Inverse discrete cosine transform (IDCT) of input

Syntax

```
Y = step(idct,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(idct,X)` computes the inverse discrete cosine transform, `Y` , of input `X`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.IFFT System object

Package: dsp

Inverse discrete Fourier transform (IDFFT)

Description

The `IFFT` object computes the inverse discrete Fourier transform (IDFFT) of the input. The object uses one or more of the following fast Fourier transform (FFT) algorithms depending on the complexity of the input and whether the output is in linear or bit-reversed order:

- Double-signal algorithm
- Half-length algorithm
- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm
- An algorithm chosen by FFTW [1], [2]

To compute the IFFT of the input:

1. Define and set up your IFFT object. See "Construction" on page 4-982.
2. Call step to compute the IFFT of the input according to the properties of `dsp.IFFT`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ift = dsp.IFFT` returns an `IFFT` object, `ift`, that computes the IDFT of a column vector or N-D array. For column vectors or N-D arrays, the `IFFT` object computes the IDFT along the first dimension of the array. If the input is a row vector, the `IFFT` object computes a row of single-sample IDFTs and issues a warning.

`ift = dsp.IFFT('PropertyName',PropertyValue, ...)` returns an `IFFT` object, `ift`, with each property set to the specified value.

Properties

`FFTImplementation`

FFT implementation

Specify the implementation used for the FFT as one of `Auto | Radix-2 | FFTW`. When you set this property to `Radix-2`, the FFT length must be a power of two.

`BitReversedInput`

Enable bit-reversed order interpretation of input elements

Set this property to`true` if the order of Fourier transformed input elements to the `IFFT` object are in bit-reversed order. This property applies only when the `FFTLengthSource` on page 4-983 property is `Auto`. The default is `false`, which denotes linear ordering.

`ConjugateSymmetricInput`

Enable conjugate symmetric interpretation of input

Set this property to `true` if the input is conjugate symmetric to yield real-valued outputs. The discrete Fourier transform of a real valued sequence is conjugate symmetric, and setting this property to `true` optimizes the IDFT computation method. Setting this property to `false` for conjugate symmetric inputs may result in complex output values with nonzero imaginary parts. This occurs due to rounding errors. Setting this property to `true` for nonconjugate symmetric inputs results in invalid outputs. This property applies only when the`FFTLengthSource` on page 4-983 property is `Auto`. The default is `false`.

`Normalize`

Enable dividing output by FFT length

Specify whether to divide the IFFT output by the FFT length. The default is `true` and each element of the output is divided by the FFT length.

`FFTLengthSource`

Source of FFT length

Specify how to determine the FFT length as `Auto` or `Property`. When you set this property to `Auto`, the FFT length equals the number of rows of the input signal. This property applies only when both the `BitReversedInput` on page 4-983 and `ConjugateSymmetricInput` on page 4-983 properties are `false`. The default is `Auto`.

FFTLength

FFT length

Specify the FFT length as a numeric scalar. This property applies when you set the `BitReversedInput` on page 4-983 and `ConjugateSymmetricInput` on page 4-983 properties to `false`, and the `FFTLengthSource` on page 4-983 property to `Property`. The default is `64`.

This property must be a power of two when the input is a fixed-point data type, or when you set the `FFTImplementation` on page 4-983 property to `Radix-2`.

When you set the `FFT implementation` property to `Radix-2`, or when you set the `BitReversedOutput` property to `true`, this value must be a power of two.

WrapInput

Boolean value of wrapping or truncating input

Wrap input data when `FFTLength` is shorter than input length. If this property is set to true, modulo-length data wrapping occurs before the FFT operation, given `FFTLength` is shorter than the input length. If this property is set to false, truncation of the input data to the `FFTLength` occurs before the FFT operation. The default is `true`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

SineTableDataType

Sine table word and fraction lengths

Specify the sine table data type as `Same word length as input` or `Custom`. The default is `Same word length as input`.

CustomSineTableDataType

Sine table word and fraction lengths

Specify the sine table fixed-point type as an unscaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SineTableDataType` on page 4-985 property to `Custom`. The default is `numerictype([],16)`.

ProductDataType

Product word and fraction lengths

Specify the product data type as `Full precision`, `Same as input`, or `Custom`. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` on page 4-985 property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator data type as `Full precision`, `Same as input`, `Same as product`, or `Custom`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` on page 4-985 property to `Custom`. The default is `numerictype([],32,30)`.

`OutputDataType`

Output word and fraction lengths

Specify the output data type as `Full precision`, `Same as input`, or `Custom`. The default is `Full precision`.

`CustomOutputDataType`

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` on page 4-986 property to `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Inverse discrete Fourier transform of input

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Inverse Discrete Fourier Transform of a Signal

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

This example analyzes the energy content in a sequence.

```
Fs = 40; L = 128;
t = (0:L-1)'/Fs;
x = sin(2*pi*10*t) + 0.75*cos(2*pi*15*t);
y = x + .5*randn(size(x)); % noisy signal
ft = dsp.FFT;
ift = dsp.IFFT('ConjugateSymmetricInput', true);
X = ft(x);
[XX, ind] = sort(abs(X),1,'descend');
XXn = sqrt(cumsum(XX.^2))/norm(XX);
ii = find(XXn > 0.999, 1);
disp('Number of FFT coefficients that represent 99.9% ')
disp(['of the total energy in the sequence: ', num2str(ii)]);
XXt = zeros(128,1);
XXt(ind(1:ii)) = X(ind(1:ii));
xt = ift(XXt);
```

```
Number of FFT coefficients that represent 99.9%
of the total energy in the sequence: 4
```

Verify that the reconstructed signal matches the original signal.

```
norm(x-xt)
```

```
ans =

   1.2012e-13
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `IFFT` block reference page. The object properties correspond to the block parameters, except: **Output sampling mode** parameter is not supported by `dsp.IFFT`.

References

[1] FFTW (http://www.fftw.org)

[2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT,"*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).
- When the following conditions apply, the executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB:
  - `FFTImplementation` is set to `'FFTW'`.
  - `FFTImplementation` is set to `'Auto'`, `FFTLengthSource` is set to `'Property'`, and `FFTLength` is not a power of two.

  Use the `packNGo` function to package the code generated from this System object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- When the FFT length is a power of two, you can generate standalone C and C++ code from this System object.

See Also

See Also

System Objects
dsp.DCT | dsp.FFT | dsp.IDCT

Introduced in R2012a

step

System object: dsp.IFFT
Package: dsp

Inverse discrete Fourier transform of input

Syntax

```
Y = step(ift,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(ift,X)` computes the inverse discrete Fourier transform (IDFT) , `Y` , of the input `X` along the first dimension of `X`. When the `FFTLengthSource` property is `Auto`, the length of `X` along the first dimension must be a positive integer power of two. When the `FFTLengthSource` property is `'Property'`, the length of `X` along the first dimension can be any positive integer and the `FFTLength` property must be a positive integer power of two.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.IIRFilter System object

Package: dsp

Infinite Impulse Response (IIR) filter

Description

The `IIRFilter` object filters each channel of the input using IIR filter implementations.

To filter each channel of the input:

1. Define and set up your IIR filter. See "Construction" on page 4-990.
2. Call step to filter each channel of the input according to the properties of `dsp.IIRFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`iir = dsp.IIRFilter` returns an IIR filter System object, `iir`, which independently filters each channel of the input over successive calls to the `step` method. This System object uses a specified IIR filter implementation.

`iir = dsp.IIRFilter('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns an IIR filter System object, `iir`, with each property set to the specified value.

Properties

`Structure`

Filter structure

Specify the filter structure.
You can specify the filter structure as one of | `Direct form I` | `Direct form I transposed` | `Direct form II` | `Direct form II transposed`. The default is `Direct form II transposed`. This property is nontunable.

Numerator

Filter numerator coefficients

Specify the numerator coefficients as a real or complex numeric row vector. The default is `[1 1]`. This property is tunable.

Denominator

Filter denominator coefficients

Specify the denominator coefficients as a real or complex numeric row vector. The default value of this property is `[1 0.1]`. This property is tunable.

InitialConditions

Initial conditions for the IIR filter

Specify the initial conditions of the filter states. This property is applicable when the structure is one of `Direct form II` | `Direct form II transposed`. The default value is `0`. The number of states equals the `max(N,M)-1`, where `N` and `M` are the number of poles and zeros respectively.

You can specify the initial conditions as a scalar, vector, or matrix. If you specify a scalar value, this System object initializes all delay elements in the filter to that value. You can also specify a vector whose length equals the number of delay elements in the filter. When you do so, each vector element specifies a unique initial condition for the corresponding delay element. The object applies the same vector of initial conditions to each channel of the input signal.

You can also specify a matrix with the same number of rows as the number of delay elements in the filter and one column for each channel of the input signal. In this case, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel. This property is tunable.

NumeratorInitialConditions

Initial conditions on zeros side

Specify the initial conditions of the filter states on the side of the filter structure with the zeros. This property is applicable when the Structure property is one of `Direct form I | Direct form I transposed`. The default value of this property is `0`. The number of states equals the `max(N,M)-1`, where `N` and `M` are the number of poles and zeros respectively.

You can specify the initial conditions as a scalar, vector, or matrix. When you specify a scalar, the IIR filter initializes all delay elements on the zeros side in the filter to that value. You can specify a vector of length equal to the number of delay elements on the zeros side in the filter. When you do so, each vector element specifies a unique initial condition for the corresponding delay element on the zeros side.

You can also specify a matrix with the same number of rows as the number of delay elements on the zeros side in the filter, and one column for each channel of the input signal. In this case, each element specifies a unique initial condition for the corresponding delay element on the zeros side in the corresponding channel. This property is tunable.

DenominatorInitialConditions

Initial conditions on poles side

Specify the initial conditions of the filter states on the side of the filter structure with the poles. This property is applicable when the Structure property is one of `Direct form I | Direct form I transposed`. The default value of this property is `0`. The number of states equals the `max(N,M)-1`, where `N` and `M` are the number of poles and zeros respectively.

You can specify the initial conditions as a scalar, vector, or matrix. When you specify a scalar, the IIR filter initializes all delay elements on the poles side of the filter to that value. You can specify a vector of length equal to the number of delay elements on the poles side in the filter. When you do so, each vector element specifies a unique initial condition for the corresponding delay element on the poles side.

You can also specify a matrix with the same number of rows as the number of delay elements on the poles side in the filter, and one column for each channel of the input signal. In this case, each element specifies a unique initial condition for the corresponding delay element on the poles side in the corresponding channel. This property is tunable.

NumeratorCoefficientsDataType

Numerator coefficients word- and fraction-length designations

Specify the numerator coefficients fixed-point data type as one of `Same word length as input` | `Custom`. The default is `Same word length as input`. This property is nontunable.

`DenominatorCoefficientsDataType`

Denominator coefficients word- and fraction-length designations

Specify the denominator coefficients fixed-point data type as one of `Same word length as input` | `Custom`. The default is `Same word length as input`. This property is nontunable.

Fixed-Point Properties

`NumeratorProductDataType`

Product word- and fraction-length designations on zeros side

Specify the numerator product fixed-point data type as one of | `Full precision` | `Same as input` | `Custom` |. The default is `Full precision`. This property is nontunable.

`DenominatorProductDataType`

Product word- and fraction-length designations on poles side

Specify the denominator product fixed-point data type as one of | `Full precision` | `Same as input` | `Custom` |. The default is `Full precision`. This property is nontunable.

`NumeratorAccumulatorDataType`

Accumulator word- and fraction-length designations on zeros side

Specify the numerator accumulator fixed-point data type to one of | `Full precision` | `Same as input` | `Same as product` | `Custom` |. The default is `Full precision`. This property is nontunable.

`DenominatorAccumulatorDataType`

Accumulator word- and fraction-length designations on poles side

Specify the denominator accumulator fixed-point data type to one of | `Full precision` | `Same as input` | `Same as product` | `Custom` |. The default is `Full precision`. This property is nontunable.

OutputDataType

Output word- and fraction-length designations

Specify the output fixed-point data type as one of | `Full precision` | `Same as input` | `Custom` |. The default is `Same as input`. This property is nontunable.

StateDataType

State word- and fraction-length designations

Specify the state fixed-point data type as one of | `Same as input` | `Custom` |. This property does not apply to `Direct form I` filter structure. The default is `Same as input`. This property is nontunable.

MultiplicandDataType

Multiplicand word- and fraction-length designations

Specify the multiplicand fixed-point data type as one of | `Same as input` | `Custom` |. This property is applicable only when the `Structure` property is `Direct form I transposed`. The default is `Same as input`. This property is nontunable.

CustomNumeratorCoefficientsDataType

Numerator coefficients word- and fraction-lengths

Specify the numerator coefficients fixed-point type as an autosigned numerictype (Fixed-Point Designer) object. This property is applicable when the `NumeratorCoefficientsDataType` on page 4-992 property is `Custom`. The default value of this property is numerictype ([],16,15). This property is nontunable.

CustomDenominatorCoefficientsDataType

Denominator coefficients word- and fraction-lengths

Specify the denominator coefficients fixed-point type as an autosigned numerictype (Fixed-Point Designer) object. This property is applicable when the

`DenominatorCoefficientsDataType` on page 4-993 property is `Custom`. The default value of this property is numerictype([], 16, 15) . This property is nontunable.

CustomNumeratorProductDataType

Custom product word- and fraction-lengths on zeros side

Specify the numerator product fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `NumeratorProductDataType` on page 4-993 property to `Custom`. The default is `numerictype ([],32,30)`. This property is nontunable.

CustomDenominatorProductDataType

Custom product word- and fraction-lengths on poles side

Specify the denominator product fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `DenominatorProductDataType` on page 4-993 property to `Custom`. The default is `numerictype ([],32,30)`. This property is nontunable.

CustomNumeratorAccumulatorDataType

Custom accumulator word- and fraction-lengths on zeros side

Specify the numerator accumulator fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `NumeratorAccumulatorDataType` on page 4-993 property to `Custom`. The default is `numerictype([],32,30)`. This property is nontunable.

CustomDenominatorAccumulatorDataType

Custom accumulator word- and fraction-lengths on poles side

Specify the denominator accumulator fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `DenominatorAccumulatorDataType` on page 4-993 property to `Custom`. The default is `numerictype([],32,30)`. This property is nontunable.

CustomStateDataType

Custom state word- and fraction-lengths

Specify the state fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `StateDataType` on page 4-994 property to `Custom`. The default is `numerictype([],16,15)`. This property is nontunable.

CustomOutputDataType

Custom output word- and fraction-lengths

Specify the output fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `OutputDataType` on page 4-994 property to `Custom`. The default is `numerictype([],16,15)`. This property is nontunable.

CustomMultiplicandDataType

Custom multiplicand word- and fraction-lengths

Specify the multiplicand fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `MultiplicandDataType` on page 4-994 property to `Custom`. The default is `numerictype([],16,15)`. This property is nontunable.

Methods

freqz	Frequency response
fvtool	Open filter visualization tool
impz	Impulse response
phasez	Unwrapped phase response
reset	Reset internal states of IIR filter
step	Filter input with IIR filter object

More "Analysis Methods for Filter System Objects" on page 3-2.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object

Common to All System Objects	
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Filter a Noisy Signal Using the IIRFilter System object

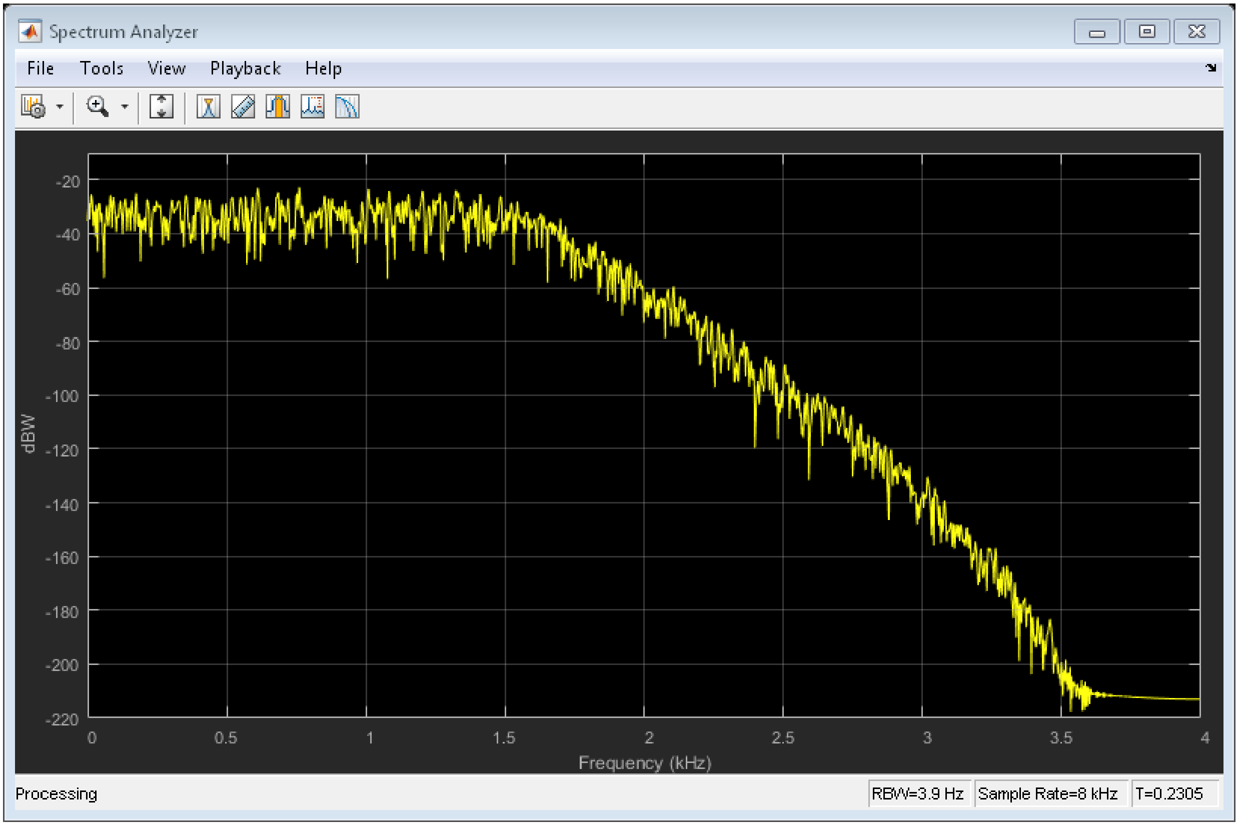
This example shows how to filter a noisy signal with a `dsp.IIRFilter` System object™. View the magnitude response of the IIR filter. Use the `Spectrum Analyzer` to display the power spectrum of the output signal.

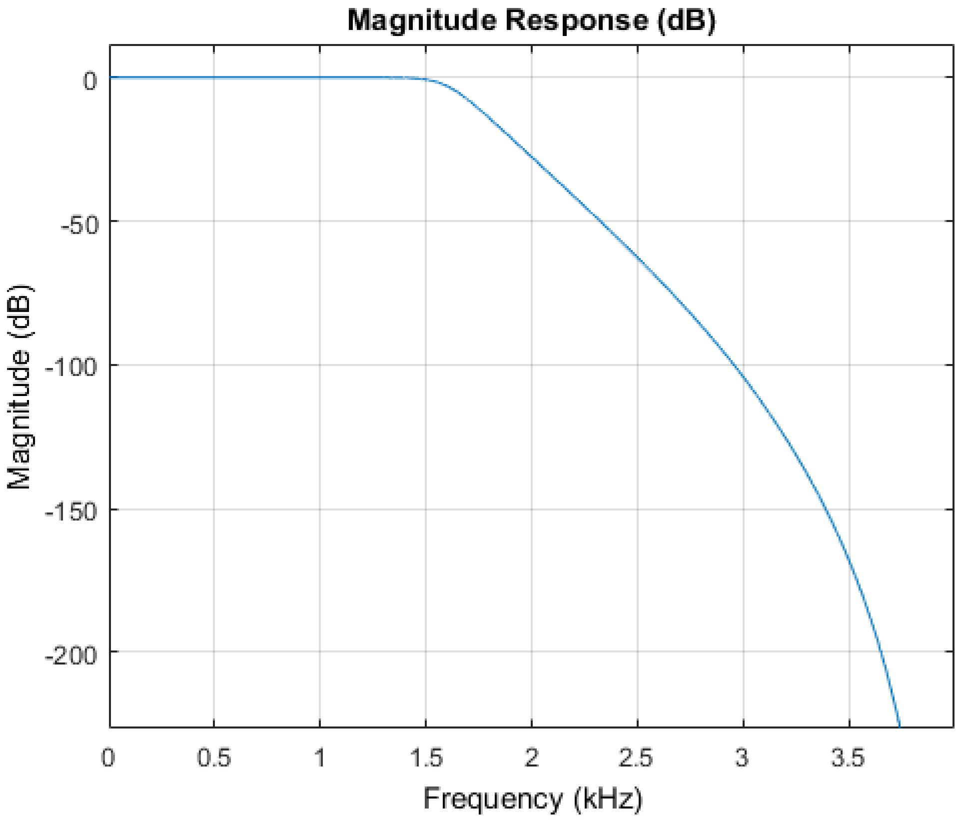
Initialization

```
x = randn(2048,1);
x = x-mean(x);
src = dsp.SignalSource;
src.Signal = x;
sink = dsp.SignalSink;

N = 10;
Fc = 0.4;
[b,a] = butter(N,Fc);
iir = dsp.IIRFilter('Numerator',b,'Denominator',a);

sa = dsp.SpectrumAnalyzer('SampleRate',8e3,...
    'PlotAsTwoSidedSpectrum',false,...
    'OverlapPercent', 80,'PowerUnits','dBW',...
    'YLimits', [-220 -10]);
```

Filter the Signal

```
while ~isDone(src)
  input = src();
  output = iir(input);
  sa(output)
  sink(output);
end

Result = sink.Buffer;
fvtool(iir,'Fs',8000)
```


Spectrum Analyzer
File Tools View Playback Help
-20
-40
-60
-80
-100
-120
-140
-160
-180
-200
-220
dBW
0 0.5 1 1.5 2 2.5 3 3.5 4
Frequency (kHz)
Processing
RBW=3.9 Hz
Sample Rate=8 kHz
T=0.2305



Design an IIR Filter

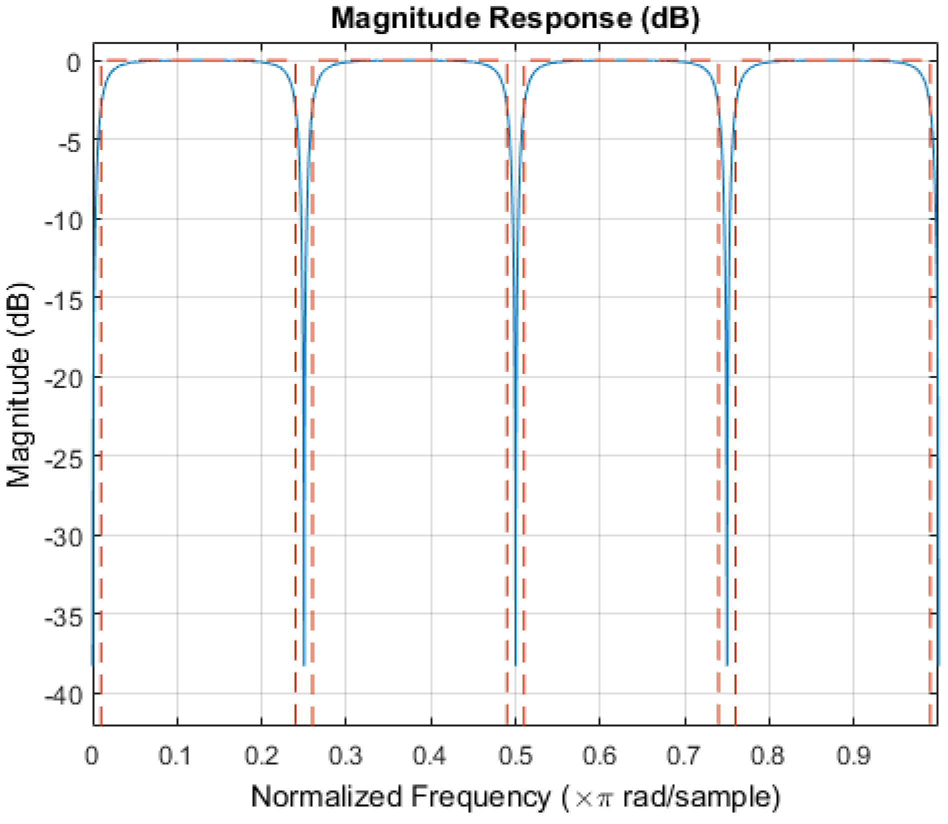
This example shows the two ways of designing an IIR Filter

Design the filter using `fdesign` and `design`.

```
D = fdesign.comb('notch','N,BW',8,0.02);
iir = design(D,'systemobject',true)
fvtool(iir);
```

```
iir =

  dsp.IIRFilter with properties:
```

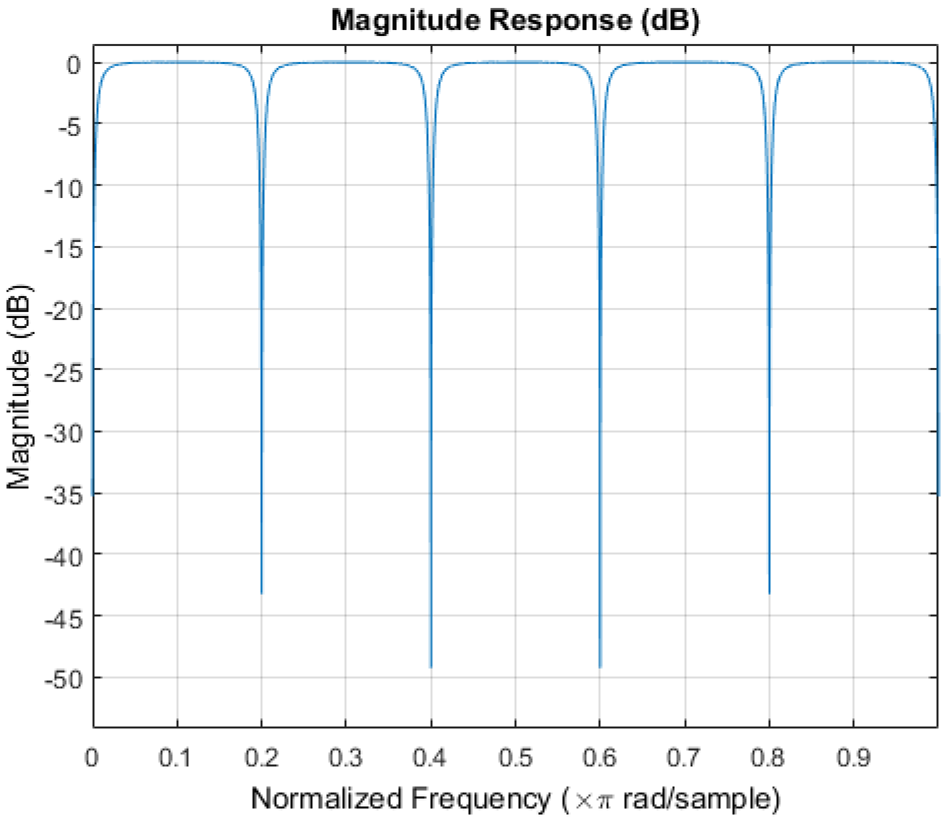
```
            Structure: 'Direct form II'
            Numerator: [0.8878 0 0 0 0 0 0 0 -0.8878]
          Denominator: [1 0 0 0 0 0 0 0 -0.7757]
    InitialConditions: 0

  Use get to show all properties
```



Design the filter using filter coefficients.

```
b = [0.9 zeros(9,1)' -0.9];
a = [1 zeros(9,1)' -0.8];
iir = dsp.IIRFilter('Numerator',b,'Denominator',a)
fvtool(iir);
```

```
iir =

  dsp.IIRFilter with properties:

            Structure: 'Direct form II transposed'
            Numerator: [0.9000 0 0 0 0 0 0 0 0 0 -0.9000]
          Denominator: [1 0 0 0 0 0 0 0 0 0 -0.8000]
    InitialConditions: 0

  Use get to show all properties
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the Discrete Filter block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the `Numerator` and `Denominator` properties are tunable for code generation.
- See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

dsp.FIRFilter | dsp.BiquadFilter | dsp.AllpoleFilter

Introduced in R2012b

freqz

System object: dsp.IIRFilter
Package: dsp

Frequency response

Syntax

```
[h,w] = freqz(iir)
[h,w] = freqz(iir,n)
[h,w] = freqz(iir,Name,Value)
freqz(iir)
```

Description

`[h,w] = freqz(iir)` returns the complex, 8192–element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample.

`[h,w] = freqz(iir,n)` returns the complex, n-element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[h,w] = freqz(iir,Name,Value)` returns the frequency response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`freqz(iir)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the filter System object `iir`.

fvtool

System object: dsp.IIRFilter
Package: dsp

Open filter visualization tool

Syntax

```
fvtool(iir)
fvtool(iir,'Arithmetic',ARITH,...)
```

Description

`fvtool(iir)` performs an analysis and computes the magnitude response of the filter System object `iir`.

`fvtool(iir,'Arithmetic',ARITH,...)` analyzes the filter System object `iir`, based on the arithmetic specified in the ARITH input. ARITH can be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The 'Arithmetic' input is only relevant for the analysis of filter System objects.

impz

System object: dsp.IIRFilter
Package: dsp

Impulse response

Syntax

```
[h,t] = impz(iir)
[h,t] = impz(iir,Name,Value)
impz(iir)
```

Description

`[h,t] = impz(iir)` returns the impulse response `h`, and the corresponding time points `t` at which the impulse response of`iir` is computed.

`[h,t] = impz(iir,Name,Value)` returns the impulse response `h`, and the corresponding time points`t` , with additional options specified by one or more Name, Value pair arguments.

`impz(iir)` uses FVTool to plot the impulse response of the filter System object `iir`.

Note: You can use `impz` for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

phasez

System object: dsp.IIRFilter
Package: dsp

Unwrapped phase response

Syntax

```
[phi,w] = phasez(iir)
[phi,w] = phasez(iir,n)
[phi,w] = phasez(iir,Name,Value)
phasez(iir)
```

Description

`[phi,w] = phasez(iir)` returns the 8192–element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample.

`[phi,w] = phasez(iir,n)` returns the n-element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[phi,w] = phasez(iir,Name,Value)` returns the phase response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`phasez(iir)` uses FVTool to plot the phase response of the filter System object `iir`.

reset

System object: dsp.IIRFilter
Package: dsp

Reset internal states of IIR filter

Syntax

```
reset(iir)
```

Description

`reset(iir)` resets the filter states of the IIR filter object, `iir`, to their initial values of 0. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the IIR filter object to nonzero input data, the states may be nonzero. Invoking the `step` method again without first invoking the`reset` method may produce different outputs for an identical input.

Reset an IIR Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
iir = dsp.IIRFilter;
[b,a] = ellip(20, 0.5, 80, 0.25);
iir.Numerator = b;
iir.Denominator = a;
n = 0:100;
x = cos(0.2*pi*n)+sin(0.8*pi*n);
y = iir(x);
```

Filter states are nonzero. Call the IIR filter without resetting states.

```
y1 = iir(x);
isequal(y,y1)
```

```
ans =

  logical

   0
```

Now reset filter states to 0.

```
reset(iir)
```

Call the IIR filter again.

```
y2 = iir(x);
isequal(y,y2)
```

```
ans =

  logical

   1
```

step

System object: dsp.IIRFilter
Package: dsp

Filter input with IIR filter object

Syntax

```
Y = step(iir,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(iir,X)` filters the real or complex input signal `X` using the IIR filter, `iir`, to produce the output `Y` . When the input data is of a fixed-point type, it must be signed. The IIR filter object operates on each channel of the input signal independently over successive calls to step method.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.IIRHalfbandDecimator System object

Package: dsp

Decimate by factor of two using polyphase IIR

Description

`dsp.IIRHalfbandDecimator` performs efficient polyphase decimation of the input signal by a factor of two. To design the halfband filter, you can specify the object to use an elliptic design or a quasi-linear phase design. The object uses these design methods to compute the filter coefficients. To filter the inputs, the object uses a polyphase structure. The allpass filters in the polyphase structure are in a minimum multiplier form.

Elliptic design introduces nonlinear phase and creates the filter using fewer coefficients than quasi linear design. Quasi-linear phase design overcomes phase nonlinearity at the cost of additional coefficients.

Alternatively, instead of designing the halfband filter using a design method, you can specify the filter coefficients directly. When you choose this option, the allpass filters in the two branches of the polyphase implementation can be in a minimum multiplier form or in a wave digital form.

You can also use the `dsp.IIRHalfbandDecimator` object to implement the analysis portion of a two-band filter bank to filter a signal into lowpass and highpass subbands. This object supports ARM Cortex code generation.

To filter and downsample your data:

1. Define and set up your halfband decimator. See "Construction" on page 4-1012.
2. Call step to filter the input signal according to the properties of `dsp.IIRHalfbandDecimator`. The input signal can be a real- or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The number of rows in the input signal must be a multiple of 2.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`IIRHalfbandDecim = dsp.IIRHalfbandDecimator` returns a halfband decimator, `IIRHalfbandDecim`, with the default settings. Calling `step` with the default property settings filters and downsamples the input data with a halfband frequency of `11025` Hz, a transition width of `4100` Hz, and a stopband attenuation of `80` dB.

`IIRHalfbandDecim = dsp.IIRHalfbandDecimator(Name,Value)` sets each property Name to the specified Value. Unspecified properties have default values.

Properties

`Specification` — Filter design parameters

`'Transition width and stopband attenuation'` (default) | `'Filter order and stopband attenuation'` | `'Filter order and transition width'` | `'Coefficients'`

Filter design parameters, specified as a character vector. When you set `Specification` to one of the filter design options, you can specify the filter design parameters using the corresponding `FilterOrder`, `StopbandAttenuation`, and `TransitionWidth` properties. Also, you can specify the design method using `DesignMethod`. When you set `Specification` to `'Coefficients'`, you can specify the coefficients directly.

`FilterOrder` — Order of the IIR halfband filter

9 (default) | positive scalar integer

Order of the IIR halfband filter, specified as a positive scalar integer. If you set `DesignMethod` to `'Elliptic'`, then `FilterOrder` must be an odd integer greater than one. If you set `DesignMethod` to `'Quasi-linear phase'`, then `FilterOrder` must be a multiple of four. This property applies when you set `Specification` to `'Filter order and stopband attenuation'` or `'Filter order and transition width'`.

`StopbandAttenuation` — Minimum attenuation needed in stopband

80 (default) | positive real scalar

Minimum attenuation needed in the stopband of the IIR halfband filter, specified as a positive real scalar. Units are in dB. This property applies only when you set `Specification` to `'Filter order and stopband attenuation'` or `'Transition width and stopband attenuation'`.

`TransitionWidth` — Transition width

4100 (default) | positive real scalar

Transition width of the IIR halfband filter, specified as a positive real scalar. Units are in Hz. The value of the transition width must be less than half the input sample rate. This property applies only when you set `Specification` to `'Transition width and stopband attenuation'` or `'Filter order and transition width'`.

`DesignMethod` — Design method

`'Elliptic'` (default) | `'Quasi-linear phase'`

Design method for the IIR halfband filter, specified as `'Elliptic'` or `'Quasi-linear phase'`. When the property is set to `'Quasi-linear phase'`, the first branch of the polyphase structure is a pure delay, which results in an approximately linear phase response. This property applies only when you set `Specification` to any accepted value except `'Coefficients'`.

`SampleRate` — Input sample rate

44100 (default) | positive real scalar

Input sample rate, specified as a positive real scalar. Units are in Hz. This property applies only when you set `Specification` to any accepted value except `'Coefficients'`.

`Structure` — Internal allpass filter implementation structure

`'Minimum multiplier'` (default) | `'Wave Digital Filter'`

Internal allpass filter implementation structure, specified as `'Minimum multiplier'` or `'Wave Digital Filter'`. This property applies only when you set `'Specification'` to `'Coefficients'`. Each structure uses a different coefficients set, independently stored in the corresponding object property.

This property is not tunable.

`AllpassCoefficients1` — Allpass polynomial filter coefficients of first branch

`[0.1284563; 0.7906755]` (default) | `[0.1284563 0.1534; 0.7906755 0.6745]`

Allpass polynomial filter coefficients of the first branch, specified as an N-by-1 or N-by-2 matrix. N is the number of first-order or second-order allpass sections. This property applies only when you set `Specification` to `'Coefficients'` and `Structure` to `'Minimum multiplier'`.

This property is tunable.

`AllpassCoefficients2` — Allpass polynomial filter coefficients of second branch

`[0.4295667]` (default) | `[0.7906755 0.1534]`

Allpass polynomial filter coefficients of the second branch, specified as an N-by-1 or N-by-2 matrix. N is the number of first-order or second-order allpass sections. This property applies only when you set `Specification` to `'Coefficients'` and `Structure` to `'Minimum multiplier'`.

This property is tunable.

`WDFCoefficients1` — Allpass filter coefficients of first branch in Wave Digital Filter form

`[0.1284563; 0.7906755]` (default) | `[0.1284563 0.1534; 0.7906755 0.6745]`

Allpass filter coefficients of the first branch in Wave Digital Filter form, specified as an N-by-1 or N-by-2 matrix. N is the number of first-order or second-order allpass sections. This property applies only when you set `Specification` to `'Coefficients'` and `Structure` to `'Wave Digital Filter'`. Each element must have an absolute value less than or equal to 1.

This property is not tunable.

`WDFCoefficients2` — Allpass filter coefficients of second branch in Wave Digital Filter form

`[0.4295667]` (default) | `[0.7906755 0.1534]`

Allpass filter coefficients of the second branch in Wave Digital Filter form, specified as the comma-separated pair consisting of `'WDFCoefficients2'` and a N-by-1 or N-by-2 matrix. N is the number of first-order or second-order allpass sections. This property applies only when you set `'Specification'` to `'Coefficients'` and `'Structure'` to `'Wave Digital Filter'`. Each element must have an absolute value less than or equal to 1.

This property is not tunable.

`HasPureDelayBranch` — Make first branch a pure delay

`false` (default) | `true`

Flag to make the first allpass branch a delay, specified as a logical scalar. This property applies only when you set `Specification` to `'Coefficients'`. When this property is true, the first branch is treated as a pure delay and the properties `AllpassCoefficients1` and `WDFCoefficients1` do not apply.

This property is not tunable.

`Delay` — Length of delay

`1` (default) | finite positive scalar

Length of the first branch delay, specified as a finite positive scalar. This property applies only when you set `Specification` to `'Coefficients'` and `HasPureDelayBranch` to 1. The value of this property specifies the number of samples by which you can delay the input to the first branch.

This property is not tunable.

`HasTrailingFirstOrderSection` — Treat the last section of the second branch as first order

`false` (default) | `true`

Option to treat the last section of the second branch as first order, specified as a logical scalar. This property applies only when you set `Specification` to `'Coefficients'`. When this property is 1 and the coefficients of the second branch are in an N-by-2 matrix, the object ignores the second element of the last row of the matrix. The last section of the second branch then becomes a first-order section. When this property is set to `0`, the last section of the second branch is a second-order section. When the coefficients of the second branch are in an N-by-1 matrix, this property is ignored.

This property is not tunable.

Methods

reset	Reset internal states of IIR halfband decimator
step	Filter input with IIR halfband decimator

For additional methods, see "Analysis Methods for Filter System Objects" on page 3-2.

For a complete list of analysis methods supported for the `dsp.IIRHalfbandDecimator` object, enter `dsp.IIRHalfbandDecimator.helpFilterAnalysis` at the MATLAB command prompt.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Frequency Response of Quasi-Linear Phase IIR Halfband Decimator

Create a minimum-order lowpass IIR halfband decimation filter for data sampled at 44.1 kHz. The filter has a transition width of 4.1 kHz, and a stopband attenuation of 80 dB.

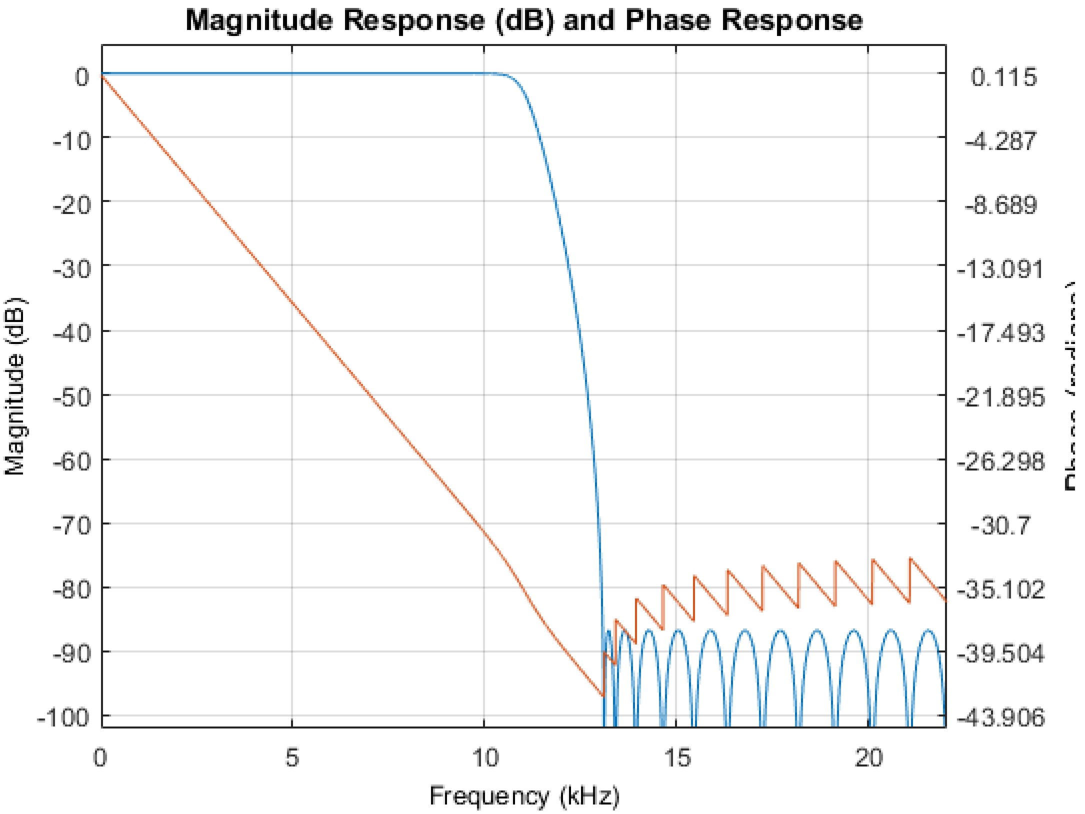
```
IIRHalfbandDecim = dsp.IIRHalfbandDecimator(...
                        'DesignMethod', 'Quasi-linear phase');
```

Obtain the filter coefficients.

```
c = coeffs(IIRHalfbandDecim);
```

Plot the magnitude and phase response.

```
fvtool(IIRHalfbandDecim,'Analysis','freq')
```



Extract Low Frequency Subband from Speech

Use a halfband analysis filter bank and interpolation filter to extract the low frequency subband from a speech signal.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

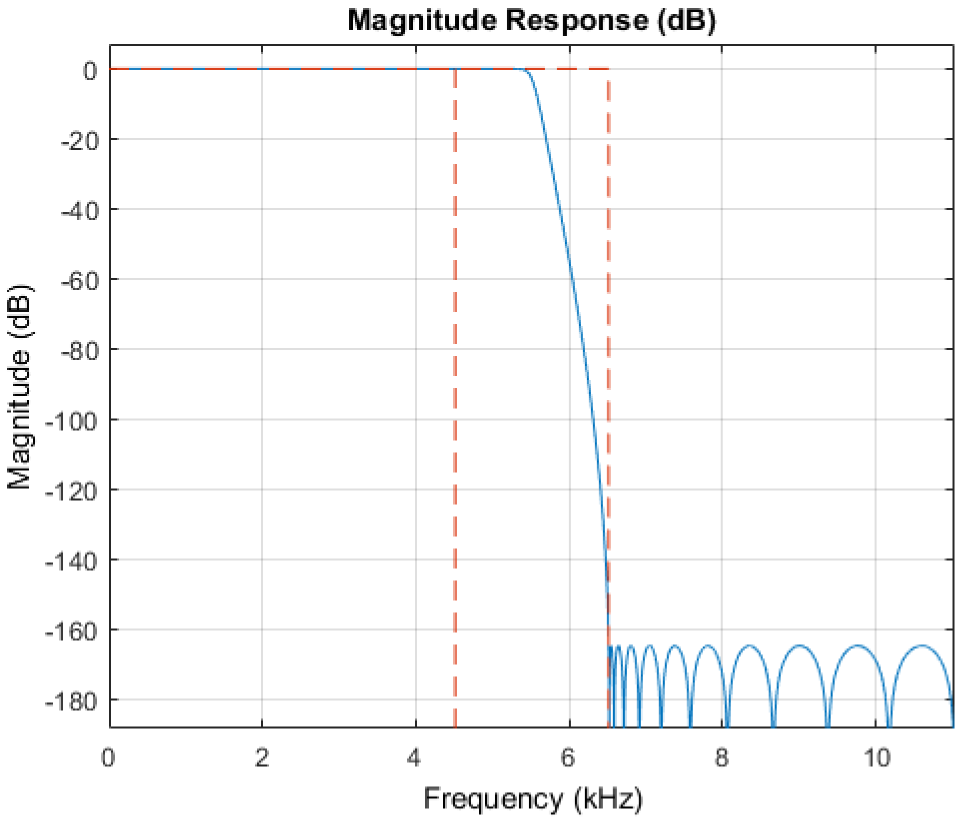
Set up the audio file reader, the analysis filter bank, the audio device writer, and the interpolation filter. The sampling rate of the audio data is 22050 Hz. The halfband filter has an order of 21 and a transition width of 2 kHz.

```
afr = dsp.AudioFileReader('speech_dft.mp3','SamplesPerFrame',1024);
```

```
filterspec = 'Filter order and transition width';
Order = 21;
TW = 2000;

IIRHalfbandDecim = dsp.IIRHalfbandDecimator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',afr.SampleRate);

IIRHalfbandInterp = dsp.IIRHalfbandInterpolator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',afr.SampleRate/2);

ap = audioDeviceWriter('SampleRate',afr.SampleRate);
```

View the magnitude response of the halfband filter.

```
fvtool(IIRHalfbandDecim)
```



Read the speech signal from the audio file in frames of 1024 samples. Filter the speech signal into lowpass and highpass subbands with a halfband frequency of 5512.5 Hz. Reconstruct a lowpass approximation of the speech signal by interpolating the lowpass subband. Play the filtered output.

```
while ~isDone(afr)
  audioframe = afr();
  xlo = IIRHalfbandDecim(audioframe);
  ylow = IIRHalfbandInterp(xlo);
  ap(ylow);
end
```

Wait until the audio file ends, and then close the input file and release the audio output resource.

```
release(afr);
release(ap);
```

Two-Channel Filter Bank

Use a halfband decimator and interpolator to implement a two-channel filter bank. This example uses an audio file input and shows that the power spectrum of the filter bank output does not differ significantly from the input.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

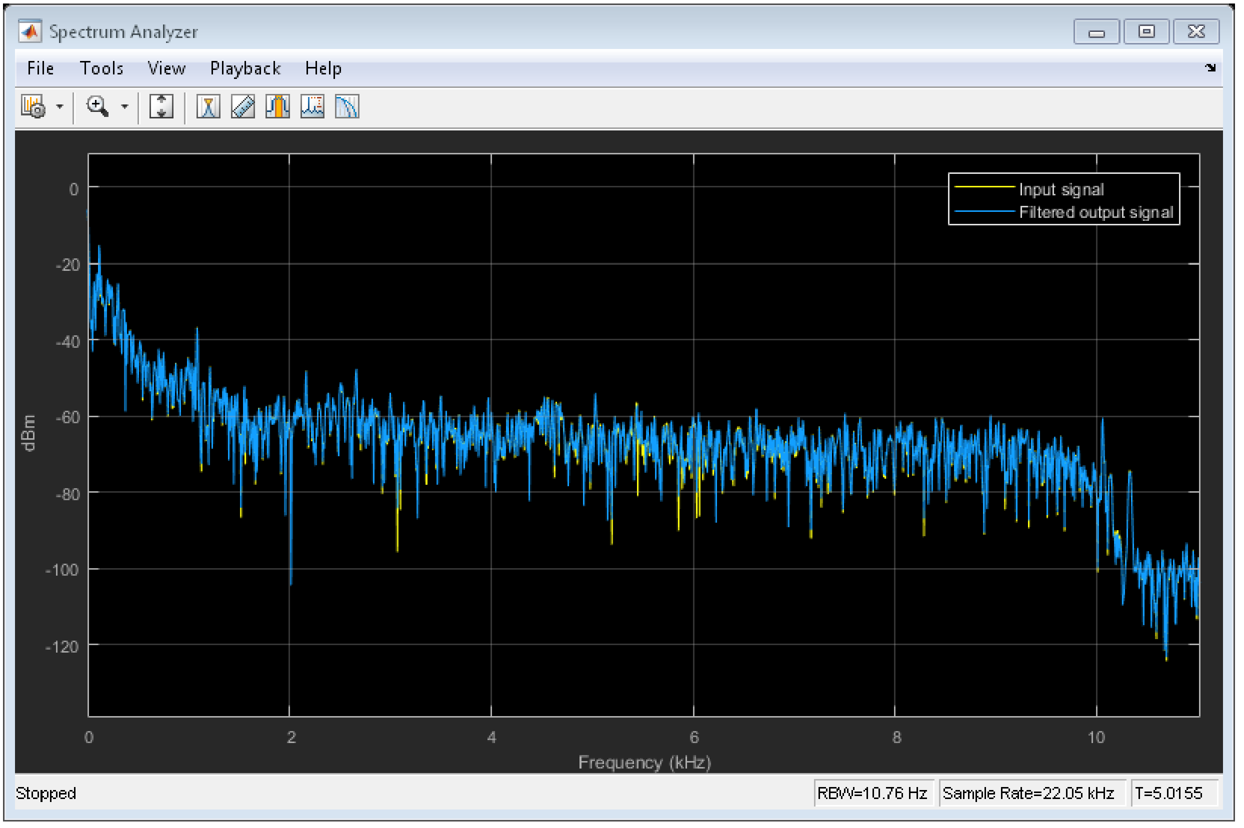
Set up the audio file reader and audio device writer. Construct the IIR halfband decimator and interpolator. Finally, set up the spectrum analyzer to display the power spectra of the filter-bank input and output.

```
AF = dsp.AudioFileReader('speech_dft.mp3','SamplesPerFrame',1024);
AP = audioDeviceWriter('SampleRate',AF.SampleRate);

filterspec = 'Filter order and transition width';
Order = 51;
TW = 2000;

IIRHalfbandDecim = dsp.IIRHalfbandDecimator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',AF.SampleRate);

IIRHalfbandInterp = dsp.IIRHalfbandInterpolator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',AF.SampleRate/2,...
    'FilterBankInputPort',true);

SpecAna = dsp.SpectrumAnalyzer('SampleRate',AF.SampleRate,...
    'PlotAsTwoSidedSpectrum',false,'ReducePlotRate',false,...
    'ShowLegend',true,...
    'ChannelNames',{'Input signal','Filtered output signal'});
```

Read the audio 1024 samples at a time. Filter the input to obtain the lowpass and highpass subband signals decimated by a factor of two. This is the analysis filter bank. Use the halfband interpolator as the synthesis filter bank. Display the running power spectrum of the audio input and the output of the synthesis filter bank. Play the output.

```
while ~isDone(AF)
    audioInput = AF();
```

```
    [xlo,xhigh] = IIRHalfbandDecim(audioInput);
    audioOutput = IIRHalfbandInterp(xlo,xhigh);
    spectrumInput = [audioInput audioOutput];
    SpecAna(spectrumInput);
    AP(audioOutput);
end

release(AF);
release(AP);
release(SpecAna);
```



Algorithms

Polyphase Implementation with Halfband Filters

When you call step to filter your signal, `dsp.IIRHalfbandDecimator` uses an efficient polyphase implementation for halfband filters. You can use the polyphase implementation to move the downsample operation before filtering. This change enables you to filter at the lower sampling rate.

IIR halfband filters are generally modeled using two parallel allpass filter branches.

$$H(z) = 0.5 * [A_1(z^2) + z^{-1} A_2(z^2)]$$

Elliptic Design

The allpass filters for elliptic IIR halfband filter are given as

$$A_1(z) = \prod_{k=1}^{K_1} \frac{a_k^{(1)} + z^{-1}}{1 + a_k^{(1)} z^{-1}}$$

$$A_2(z) = \prod_{k=1}^{K_2} \frac{a_k^{(2)} + z^{-1}}{1 + a_k^{(2)} z^{-1}}$$

Quasi-Linear Phase Design

To achieve a near-linear phase response for IIR halfband filters, make one of the branches a pure delay. In this design, the cost of the filter increases.

The allpass filters for the quasi-linear phase IIR halfband filter are

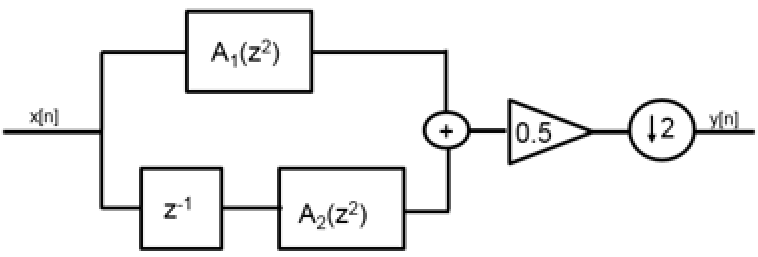
$$A_1(z) = z^{-k}$$

where k is the length of the delay.

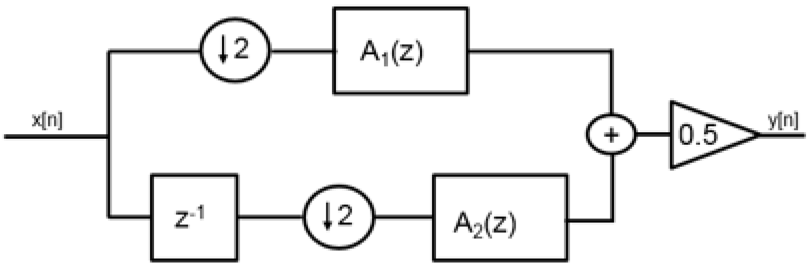
$$A_2(z) = \prod_{K=1}^{K_2^{(1)}} \frac{a_k + z^{-1}}{1 + a_k z^{-1}} \prod_{K=1}^{K_2^{(2)}} \frac{c_k + b_k z^{-1} + z^{-2}}{1 + b_k z^{-1} + c_k z^{-2}}$$

where N is the order of the IIR halfband filter.

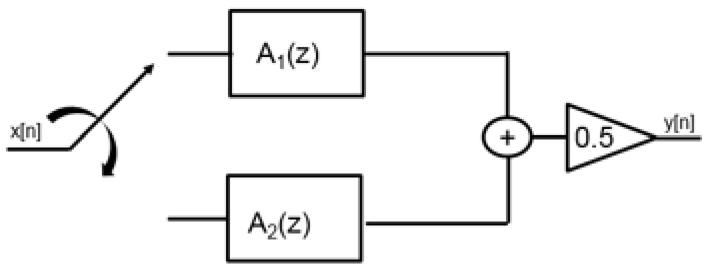
This figure represents filtering and downsampling the input by two.



Using the multirate noble identity for downsampling, you can move the downsampling operation before filtering. This change enables you to filter at the lower rate.



To implement the halfband decimator efficiently, `dsp.IIRHalfbandDecimator` replaces the delay block and downsampling operator with a commutator switch.

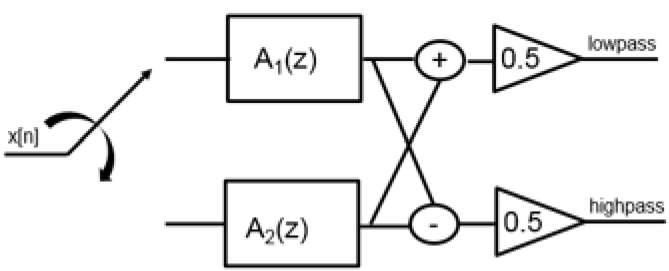


Analysis Filter Bank

The transfer function of the complementary highpass filter branch of the analysis filter bank is given by

$$G(z) = 0.5 * [A_1(z^2) - z^{-1} A_2(z^2)]$$

Graphically, you can represent the analysis filter bank as follows.



`dsp.IIRHalfbandDecimator` generates two power-complementary output signals by adding and subtracting the two polyphase branch outputs respectively.

To summarize, `dsp.IIRHalfbandDecimator`

- Decimates the input prior to filtering.
- Acts as an analysis filter bank.
- Has non-linear phase response and uses few coefficients with elliptic design method.
- Has near-linear phase response at the cost of additional coefficients with quasi-linear phase design method. One of the branches in this design is a pure delay.

References

[1] Lang, M. *Allpass Filter Design and Applications*. IEEE Transactions on Signal Processing. Vol. 46, No. 9, Sept 1998, pp. 2505–2514.

[2] Harris, F.J. *Multirate Signal Processing for Communication Systems*. Prentice Hall. 2004, pp. 208–209.

[3] Regalia, Phillip A., Sanjit K. Mitra, and P. P. Vaidyanathan. "The Digital All-Pass Filter: A Versatile Signal Processing Building Block." *Proceedings of the IEEE*. Vol. 76, Number 1, 1988, pp. 19-37.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

System Objects

dsp.FIRHalfbandDecimator | dsp.FIRHalfbandInterpolator | dsp.IIRHalfbandInterpolator

Blocks

`FIR Halfband Decimator` | `IIR Halfband Decimator`

Introduced in R2015b

reset

System object: dsp.IIRHalfbandDecimator
Package: dsp

Reset internal states of IIR halfband decimator

Syntax

`reset(IIRHalfbandDecim)`

Description

`reset(IIRHalfbandDecim)` resets the filter states of the IIR halfband decimator to their initial values of 0. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the halfband decimator to nonzero input data, the filter might have nonzero states. If you Invoke the `step` method again without first invoking the `reset` method, the object might produce different outputs for an identical input.

Input Arguments

`IIRHalfbandDecim` — IIR halfband decimator

`IIRHalfbandDecimator` System object

IIR halfband decimator, specified as an `IIRHalfbandDecimator` System object.

Examples

Reset an IIR Halfband Decimator

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create an IIR halfband decimator with default properties.

```
IIRHalfbandDecim = dsp.IIRHalfbandDecimator;
```

Create a two-channel random signal. Apply the `step` method twice on the signal.

```
x = randn(10,2);

y1 = IIRHalfbandDecim(x);
y2 = IIRHalfbandDecim(x);

no = all(y2==y1)
```

```
no =

  1×2 logical array

   0   0
```

The output is different because the internal states of `IIRHalfbandDecim` have changed. Use `reset` to reset the IIR halfband decimator and apply `step` again. Verify that the output is unchanged.

```
reset(IIRHalfbandDecim)

y3 = IIRHalfbandDecim,(x);

yes = all(y3==y1)
```

```
y3 =

  dsp.IIRHalfbandDecimator with properties:

          Specification: 'Transition width and stopband attenuation'
        TransitionWidth: 4100
    StopbandAttenuation: 80
           DesignMethod: 'Elliptic'
             SampleRate: 44100


yes =
```

```
1×2 logical array

 0   0
```

step

System object: dsp.IIRHalfbandDecimator
Package: dsp

Filter input with IIR halfband decimator

Syntax

```
Y = step(IIRHalfbanddecim,X)
[Ylow,Yhigh] = step(IIRHalfbanddecim,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(IIRHalfbanddecim,X)` filters the real or complex input signal, `X`, using the IIR halfband filter, `hiirhalfbanddecim`, and downsamples the output by a factor of 2. `Y` is a lowpass filtered and downsampled version of the input `X`. The input `X` must be a column vector or matrix with an even number of rows. If `X` is a matrix, each column is treated as an independent channel. The input `X` supports single and double data types.

`[Ylow,Yhigh] = step(IIRHalfbanddecim,X)` computes the `Ylow` and `Yhigh`, of the analysis filter bank, `IIRHalfbanddecim` for input `X`. A *Ki*-by-*N* input matrix is treated as *N* independent channels. The System object generates two power-complementary output signals by adding and subtracting the two polyphase branch outputs respectively. `Ylow` and `Yhigh` are of the same size (`Ko-by-N`) and data type. `Ko = Ki/2`, where `2` is the decimation factor.

Examples

Filter Input into Lowpass and Highpass Subbands

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create a halfband decimator for data sampled at 44.1 kHz. Use a minimum-order design with a transition width of 2 kHz and a stopband attenuation of 60 dB.

```
IIRHalfbanddecim = dsp.IIRHalfbandDecimator(...
    'Specification','Transition width and stopband attenuation',...
    'TransitionWidth',2000,'StopbandAttenuation',60,'SampleRate',44.1e3);
```

Filter a two-channel input into lowpass and highpass subbands.

```
x = randn(1024,2);
[ylow,yhigh] = IIRHalfbanddecim(x);
```

Introduced in R2015b

dsp.IIRHalfbandInterpolator System object

Package: dsp

Interpolate by a factor of two using polyphase IIR

Description

`dsp.IIRHalfbandInterpolator` performs efficient polyphase interpolation of the input signal by a factor of two. To design the halfband filter, you can specify the object to use an elliptic design or a quasi-linear phase design. The object uses these design methods to compute the filter coefficients. To filter the inputs, the object uses a polyphase structure. The allpass filters in the polyphase structure are in a minimum multiplier form.

Elliptic design introduces nonlinear phase and creates the filter using fewer coefficients than quasi linear design. Quasi-linear phase design overcomes phase nonlinearity at the cost of additional coefficients.

Alternatively, instead of designing the halfband filter using a design method, you can specify the filter coefficients directly. When you choose this option, the allpass filters in the two branches of the polyphase implementation can be in a minimum multiplier form or in a wave digital form.

You can also use `dsp.IIRHalfbandInterpolator` object to implement the synthesis portion of a two-band filter bank to synthesize a signal from lowpass and highpass subbands. This object supports ARM Cortex code generation.

To upsample and interpolate your data:

1. Define and set up your halfband interpolator. See “Construction” on page 4-1033.
2. Call step to filter the input signal according to the properties of `dsp.IIRHalfbandInterpolator`. The input signal can be a real- or complex-valued column vector or matrix. If the input signal is a matrix, each column of the matrix is treated as an independent channel.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`IIRHalfbandInterp = dsp.IIRHalfbandInterpolator` returns an IIR halfband interpolation filter, `IIRHalfbandInterp`, with the default settings. Calling `step` with the default property settings upsamples and interpolates the input data using a halfband frequency of `11025` Hz, a transition width of `4100` Hz, and a stopband attenuation of `80` dB.

`IIRHalfbandInterp = dsp.IIRHalfbandInterpolator(Name,Value)` sets each property Name to the specified Value. Unspecified properties have default values.

Properties

`Specification` — Filter design parameters

`'Transition width and stopband attenuation'` (default) | `'Filter order and stopband attenuation'` | `'Filter order and transition width'` | `'Coefficients'`

Filter design parameters, specified as a character vector. When you set `Specification` to one of the filter design options, you can specify the filter design parameters using the corresponding `FilterOrder`, `StopbandAttenuation`, and `TransitionWidth` properties. Also, you can specify the design method using `DesignMethod`. When you set `Specification` to `'Coefficients'`, you can specify the coefficients directly.

`FilterOrder` — Order of the IIR halfband filter

9 (default) | positive scalar integer

Order of the IIR halfband filter, specified as a positive scalar integer. If you set `DesignMethod` to `'Elliptic'`, then `FilterOrder` must be an odd integer greater than one. If you set `DesignMethod` to `'Quasi-linear phase'`, then `FilterOrder` must be a multiple of four. This property applies when you set `Specification` to `'Filter order and stopband attenuation'` or `'Filter order and transition width'`.

`StopbandAttenuation` — Minimum attenuation needed in stopband

80 (default) | positive real scalar

Minimum attenuation needed in the stopband of the IIR halfband filter, specified as a positive real scalar. Units are in dB. This property applies only when you set `Specification` to `'Filter order and stopband attenuation'` or `'Transition width and stopband attenuation'`.

`TransitionWidth` — Transition width

4100 (default) | positive real scalar

Transition width of the IIR halfband filter, specified as a positive real scalar. Units are in Hz. The value of the transition width must be less than half the input sample rate. This property applies only when you set `Specification` to `'Transition width and stopband attenuation'` or `'Filter order and transition width'`.

`DesignMethod` — Design method

`'Elliptic'` (default) | `'Quasi-linear phase'`

Design method for the IIR halfband filter, specified as `'Elliptic'` or `'Quasi-linear phase'`. When the property is set to `'Quasi-linear phase'`, the first branch of the polyphase structure is a pure delay, which results in an approximately linear phase response. This property applies only when you set `Specification` to any accepted value except `'Coefficients'`.

`SampleRate` — Input sample rate

44100 (default) | positive real scalar

Input sample rate, specified as a positive real scalar. Units are in Hz. This property applies only when you set `Specification` to any accepted value except `'Coefficients'`.

`FilterBankInputPort` — Option to use object as synthesis filter bank

`false` (default) | `true`

Option to use object as synthesis filter bank, specified as a logical value. If this property is `false`, `dsp.IIRHalfbandInterpolator` acts as an interpolator. If this property is `true`, then `dsp.IIRHalfbandInterpolator` acts as a synthesis filter bank and `step` accepts two inputs: the lowpass and highpass subbands. This property applies only when you set `Specification` to any accepted value except `'Coefficients'`.

`Structure` — Filter structure used in coefficient mode

`'Minimum multiplier'` (default) | `'Wave Digital Filter'`

Internal allpass filter implementation structure, specified as 'Minimum multiplier' or 'Wave Digital Filter'. This property applies only when you set 'Specification' to 'Coefficients'. Each structure uses a different coefficients set, independently stored in the corresponding object property.

This property is not tunable.

AllpassCoefficients1 — Allpass polynomial filter coefficients of first branch

[0.1284563; 0.7906755] (default) | [0.1284563 0.1534; 0.7906755 0.6745]

Allpass polynomial filter coefficients of the first branch, specified as an *N*-by-1 or *N*-by-2 matrix. *N* is the number of first-order or second-order allpass sections. This property applies only when you set Specification to 'Coefficients' and Structure to 'Minimum multiplier'.

This property is tunable.

AllpassCoefficients2 — Allpass polynomial filter coefficients of second branch

[0.4295667] (default) | [0.7906755 0.1534]

Allpass polynomial filter coefficients of the second branch, specified as an *N*-by-1 or *N*-by-2 matrix. *N* is the number of first-order or second-order allpass sections. This property applies only when you set Specification to 'Coefficients' and Structure to 'Minimum multiplier'.

This property is tunable.

WDFCoefficients1 — Allpass filter coefficients of first branch in Wave Digital Filter form

[0.1284563; 0.7906755] (default) | [0.1284563 0.1534; 0.7906755 0.6745]

Allpass filter coefficients of the first branch in Wave Digital Filter form, specified as an *N*-by-1 or *N*-by-2 matrix. *N* is the number of first-order or second-order allpass sections. This property applies only when you set Specification to 'Coefficients' and Structure to 'Wave Digital Filter'. All elements must have an absolute value less than or equal to 1.

This property is not tunable.

WDFCoefficients2 — Allpass filter coefficients of second branch in Wave Digital Filter form

[0.4295667] (default) | [0.7906755 0.1534]

Allpass filter coefficients of the second branch in Wave Digital Filter form, specified as an N-by-1 or N-by-2 matrix. N is the number of first-order or second-order allpass sections. This property applies only when you set `Specification` to `'Coefficients'` and `Structure` to `'Wave Digital Filter'`. All elements must have an absolute value less than or equal to `1`.

This property is not tunable.

`HasPureDelayBranch` — Make the first branch a pure delay

`false` (default) | `true`

Flag to make the first allpass branch a delay, specified as a logical scalar. This property applies only when you set `Specification` to `'Coefficients'`. When this property is true, the first branch is treated as a pure delay and the properties `AllpassCoefficients1` and `WDFCoefficients1` do not apply.

This property is not tunable.

`Delay` — Length of the delay

`1` (default) | finite positive scalar

Length of the first branch delay, specified as a finite positive scalar. This property is applicable only when you set `'Specification'` to `'Coefficients'` and `HasPureDelayBranch` to 1. The value of this property specifies the number of samples by which you can delay the input to the first branch.

This property is not tunable.

`HasTrailingFirstOrderSection` — Make the last section of the second branch as first order

`false` (default) | `true`

Option to treat the last section of the second branch as first order, specified as a logical scalar. This property applies only when you set `Specification` to `'Coefficients'`. When this property is 1 and the coefficients of the second branch are in an N-by-2 matrix, the object ignores the second element of the last row of the matrix. The last section of the second branch then becomes a first-order section. When this property is set to `0`, the last section of the second branch is a second-order section. When the coefficients of the second branch are in an N-by-1 matrix, this property is ignored.

This property is not tunable.

Methods

reset	Reset internal states of IIR halfband interpolator
step	Filter input with IIR halfband interpolator

For additional methods, see "Analysis Methods for Filter System Objects" on page 3-2.

For a complete list of analysis methods supported for the `dsp.IIRHalfbandInterpolator` object, enter `dsp.IIRHalfbandInterpolator.helpFilterAnalysis` at the MATLAB command prompt.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Frequency response of Quasi-linear Phase IIR Halfband Interpolator

Create a minimum order lowpass IIR half-band interpolation filter for data sampled at 44.1 kHz. The filter has a transition width of 4.1 kHz, and a stopband attenuation of 80 dB.

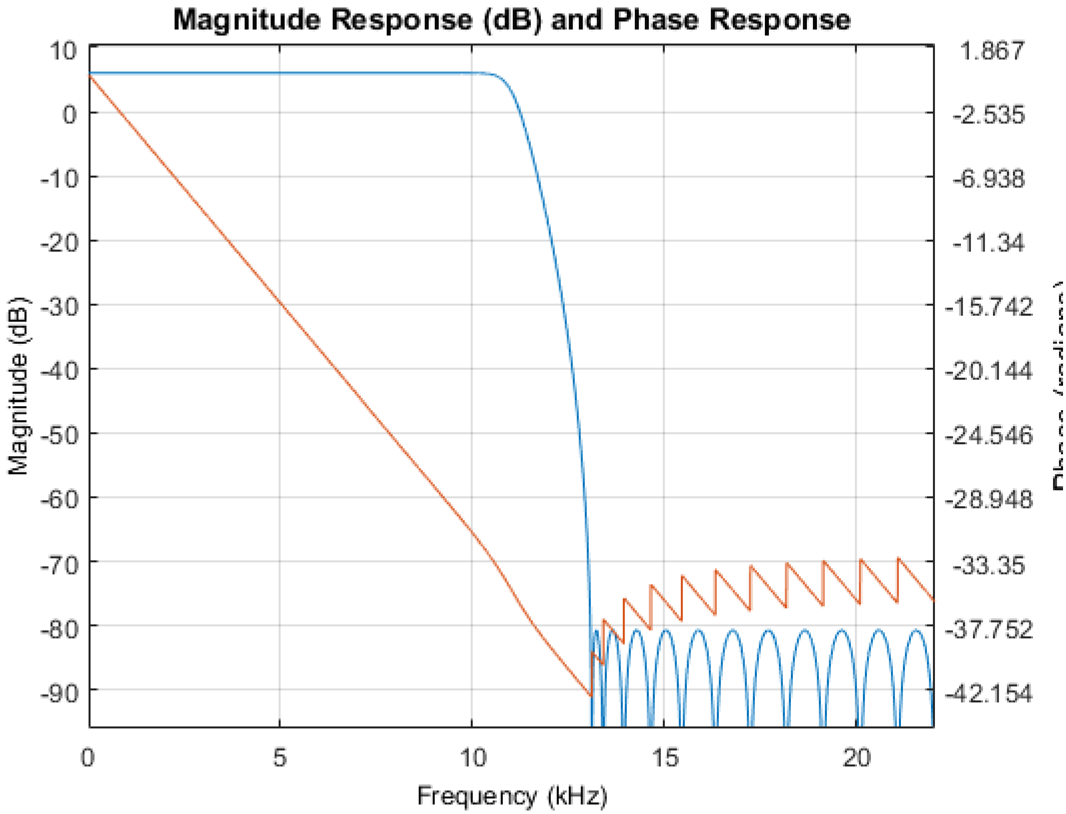
```
IIRHalfbandInterp = dsp.IIRHalfbandInterpolator(...
                        'DesignMethod', 'Quasi-linear phase');
```

Obtain filter coefficients

```
c = coeffs(IIRHalfbandInterp);
```

Plot the Magnitude and Phase response

```
fvtool(IIRHalfbandInterp,'Analysis','freq')
```



Extract Low Frequency Subband from Speech

Use a halfband analysis filter bank and interpolation filter to extract the low frequency subband from a speech signal.

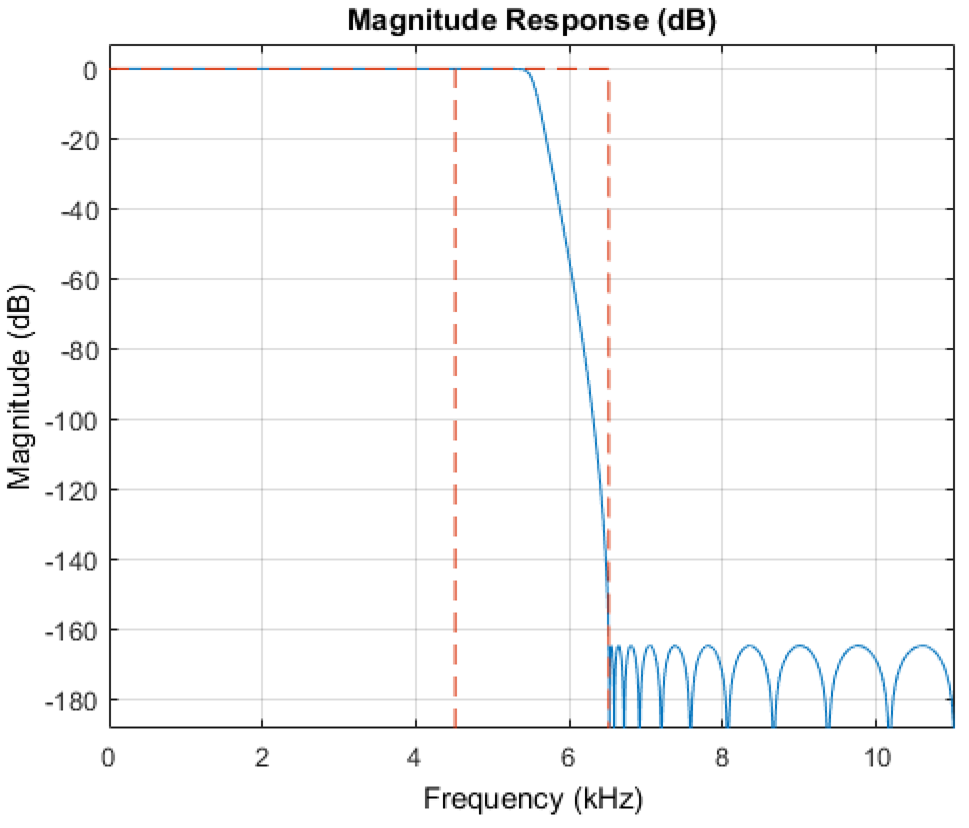
Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Set up the audio file reader, the analysis filter bank, the audio device writer, and the interpolation filter. The sampling rate of the audio data is 22050 Hz. The halfband filter has an order of 21 and a transition width of 2 kHz.

```
afr = dsp.AudioFileReader('speech_dft.mp3','SamplesPerFrame',1024);

filterspec = 'Filter order and transition width';
Order = 21;
TW = 2000;

IIRHalfbandDecim = dsp.IIRHalfbandDecimator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',afr.SampleRate);

IIRHalfbandInterp = dsp.IIRHalfbandInterpolator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',afr.SampleRate/2);

ap = audioDeviceWriter('SampleRate',afr.SampleRate);
```

View the magnitude response of the halfband filter.

```
fvtool(IIRHalfbandDecim)
```



Read the speech signal from the audio file in frames of 1024 samples. Filter the speech signal into lowpass and highpass subbands with a halfband frequency of 5512.5 Hz. Reconstruct a lowpass approximation of the speech signal by interpolating the lowpass subband. Play the filtered output.

```
while ~isDone(afr)
  audioframe = afr();
  xlo = IIRHalfbandDecim(audioframe);
  ylow = IIRHalfbandInterp(xlo);
  ap(ylow);
end
```

Wait until the audio file ends, and then close the input file and release the audio output resource.

```
release(afr);
release(ap);
```

Two-Channel Filter Bank

Use a halfband decimator and interpolator to implement a two-channel filter bank. This example uses an audio file input and shows that the power spectrum of the filter bank output does not differ significantly from the input.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

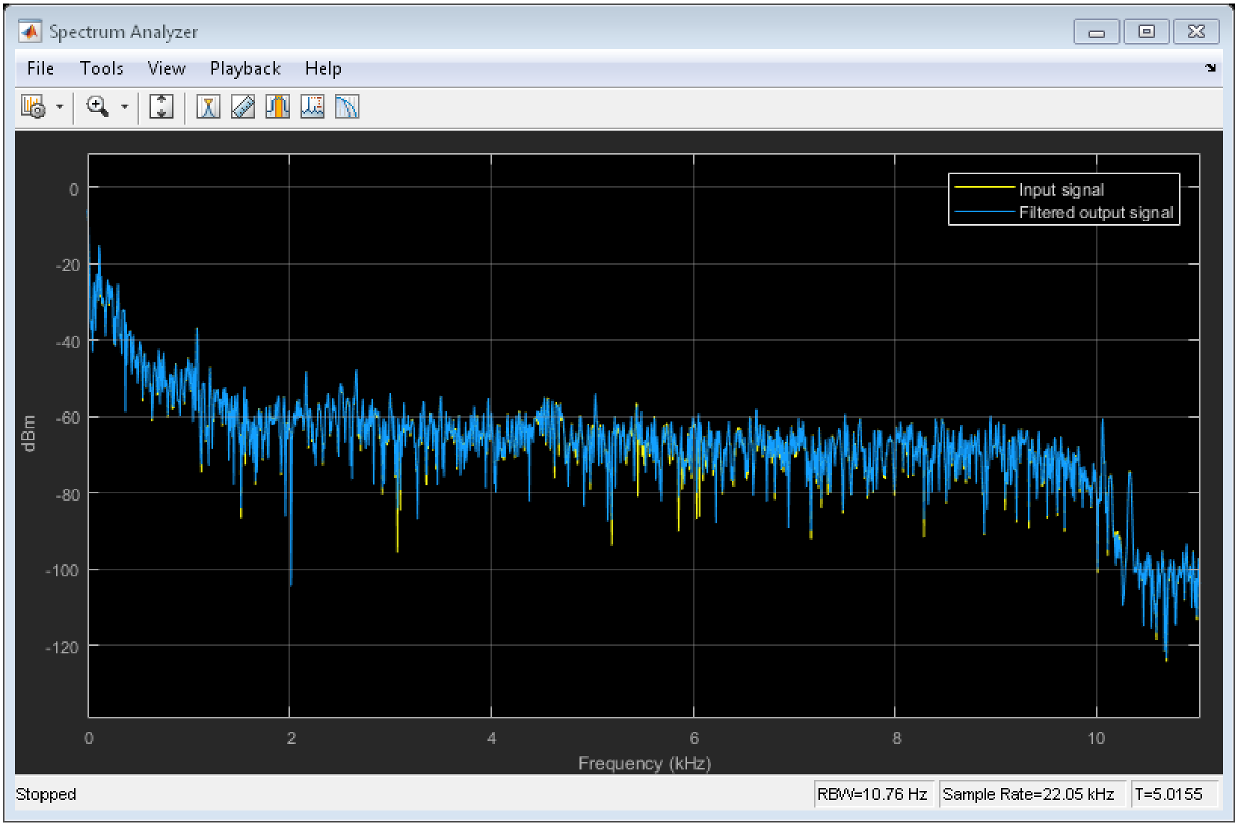
Set up the audio file reader and audio device writer. Construct the IIR halfband decimator and interpolator. Finally, set up the spectrum analyzer to display the power spectra of the filter-bank input and output.

```
AF = dsp.AudioFileReader('speech_dft.mp3','SamplesPerFrame',1024);
AP = audioDeviceWriter('SampleRate',AF.SampleRate);

filterspec = 'Filter order and transition width';
Order = 51;
TW = 2000;

IIRHalfbandDecim = dsp.IIRHalfbandDecimator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',AF.SampleRate);

IIRHalfbandInterp = dsp.IIRHalfbandInterpolator(...
    'Specification',filterspec,'FilterOrder',Order,...
    'TransitionWidth',TW,'SampleRate',AF.SampleRate/2,...
    'FilterBankInputPort',true);

SpecAna = dsp.SpectrumAnalyzer('SampleRate',AF.SampleRate,...
    'PlotAsTwoSidedSpectrum',false,'ReducePlotRate',false,...
    'ShowLegend',true,...
    'ChannelNames',{'Input signal','Filtered output signal'});
```

Read the audio 1024 samples at a time. Filter the input to obtain the lowpass and highpass subband signals decimated by a factor of two. This is the analysis filter bank. Use the halfband interpolator as the synthesis filter bank. Display the running power spectrum of the audio input and the output of the synthesis filter bank. Play the output.

```
while ~isDone(AF)
    audioInput = AF();
```

```
    [xlo,xhigh] = IIRHalfbandDecim(audioInput);
    audioOutput = IIRHalfbandInterp(xlo,xhigh);
    spectrumInput = [audioInput audioOutput];
    SpecAna(spectrumInput);
    AP(audioOutput);
end

release(AF);
release(AP);
release(SpecAna);
```



Algorithms

Polyphase Implementation with Halfband Filters

When you call step to filter your signal, `dsp.IIRHalfbandInterpolator` uses an efficient polyphase implementation for halfband filters. You can use a polyphase implementation to move the upsampling operation after filtering. This change enables you to filter at the lower sampling rate.

IIR halfband filters are generally modeled using two parallel allpass filter branches.

$$H(z) = 0.5 * [A_1(z^2) + z^{-1} A_2(z^2)]$$

Elliptic Design

The allpass filters for elliptic IIR halfband filter are given as

$$A_1(z) = \prod_{k=1}^{K_1} \frac{a_k^{(1)} + z^{-1}}{1 + a_k^{(1)} z^{-1}}$$

$$A_2(z) = \prod_{k=1}^{K_2} \frac{a_k^{(2)} + z^{-1}}{1 + a_k^{(2)} z^{-1}}$$

Quasi-Linear Phase Design

A near-linear phase response for IIR halfband filters is achieved by making one of the branches a pure delay. In this design, the cost of the filter increases.

The allpass filters for quasi-linear phase IIR halfband filter are

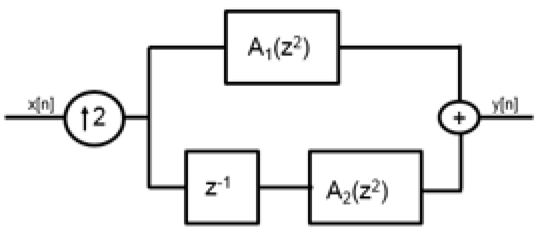
$$A_1(z) = z^{-k}$$

where, k is the length of the delay.

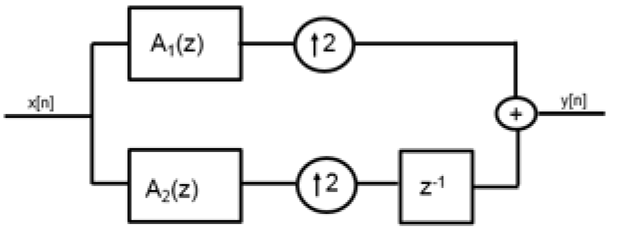
$$A_2(z) = \prod_{K=1}^{K_2^{(1)}} \frac{a_k + z^{-1}}{1 + a_k z^{-1}} \prod_{K=1}^{K_2^{(2)}} \frac{c_k + b_k z^{-1} + z^{-2}}{1 + b_k z^{-1} + c_k z^{-2}}$$

where N is the order of the IIR halfband filter.

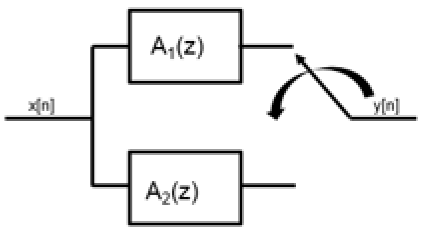
Graphically, you can represent upsampling by two followed by filtering with the following figure



Using the multirate noble identity for upsampling, you can move the upsampling operation after filtering. This enables you to filter at the lower rate.



To efficiently implement the halfband interpolator, `dsp.IIRHalfbandInterpolator` replaces the upsampling operator, delay block, and adder with a commutator switch. The commutator switch operates at twice the input sample rate. This is shown in the following figure:



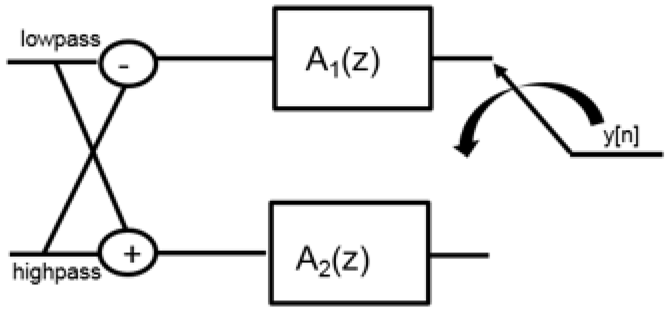
The commutator switch takes input samples from the two branches alternately, one sample at a time. This doubles the sampling rate of the input signal.

Synthesis Filter Bank

Transfer function of the complementary high-pass filter branch of the synthesis filter bank is given by

$$G(z) = 0.5 * [A_1(z^2) - z^{-1} A_2(z^2)]$$

Graphically, you can represent the synthesis filter bank as



`dsp.IIRHalfbandInterpolator` to implement the synthesis portion of a two-band filter bank to synthesize a signal from lowpass and highpass subbands.

To summarize, `dsp.IIRHalfbandInterpolator`

- Filters the input before upsampling
- acts as a synthesis filter bank
- has non-linear phase response and uses few coefficients with elliptic design method
- has near-linear phase response at the cost of additional coefficients with quasi-linear phase design method. One of the branches in this design is a pure delay.

References

[1] Lang, M. *Allpass Filter Design and Applications.* IEEE Transactions on Signal Processing. Vol. 46, No. 9, Sept 1998, pp. 2505–2514.

[2] Harris, F.J. *Multirate Signal Processing for Communication Systems*. Prentice Hall. 2004, pp. 208–209.

[3] Regalia, Phillip A., Sanjit K. Mitra, and P. P. Vaidyanathan. "The Digital All-Pass Filter: A Versatile Signal Processing Building Block." *Proceedings of the IEEE.* Vol. 76, Number 1, 1988, pp. 19-37.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

dsp.FIRHalfbandDecimator | dsp.FIRHalfbandInterpolator | dsp.IIRHalfbandDecimator | `IIR Halfband Interpolator`

Introduced in R2015b

reset

System object: dsp.IIRHalfbandInterpolator
Package: dsp

Reset internal states of IIR halfband interpolator

Syntax

`reset(IIRHalfbandInterp)`

Description

`reset(IIRHalfbandInterp)` resets the filter states of the IIR halfband interpolator,`IIRHalfbandInterp`, to their initial values of 0. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the halfband interpolator to nonzero input data, the states may be nonzero. If you Invoke the `step` method again without first invoking the `reset` method, the object might produce different outputs for an identical input.

Input Arguments

`IIRHalfbandInterp` — IIR halfband interpolator
`dsp.IIRHalfbandInterpolator` System object

IIR halfband interpolator, specified as a `dsp.IIRHalfbandInterpolator` System object.

Examples

Reset an IIR Halfband Interpolator

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create an IIR halfband interpolator with default properties.

```
IIRHalfbandInterp = dsp.IIRHalfbandInterpolator;
```

Create a two-channel random signal. Apply the `step` method twice on the signal.

```
x = randn(10,2);

y1 = IIRHalfbandInterp(x);
y2 = IIRHalfbandInterp(x);

no = all(y2==y1)
```

```
no =

  1×2 logical array

   0   0
```

The output is different because the internal states of `IIRHalfbandInterp` have changed. Use `reset` to reset the IIR halfband interpolator and apply `step` again. Verify that the output is unchanged.

```
reset(IIRHalfbandInterp)

y3 = IIRHalfbandInterp(x);

yes = all(y3==y1)
```

```
yes =

  1×2 logical array

   1   1
```

step

System object: dsp.IIRHalfbandInterpolator
Package: dsp

Filter input with IIR halfband interpolator

Syntax

```
Y = step(IIRHalfbandInterp,X)
Y = step(hiirhalfbandinterp,X1,X2)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(IIRHalfbandInterp,X)` upsamples by two and interpolates the real or complex input signal `X` using the IIR halfband interpolator, `hfirhalfbandinterp`. The input `X` must be a column vector or matrix. If `X` is a matrix, each column is treated as an independent channel. The input `X` supports single, and double data types.

`Y = step(hiirhalfbandinterp,X1,X2)` implements a halfband synthesis filter bank for the inputs `X1` and `X2`. `X1` is the lowpass output of a halfband analysis filter bank and `X2` is the highpass output of a halfband analysis filter bank. `dsp.IIRHalfbandInterpolator` implements a synthesis filter bank only when the `FilterBankInputPort` property is `true`.

Examples

Upsample and Interpolate Multichannel Input with IIR Halfband Interpolator

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create a half-band interpolation filter for data sampled at 44.1 kHz. The filter order is 51 with a transition width of 4.1 kHz. Use the filter to upsample and interpolate a multichannel input.

```
Fs = 44.1e3;
filterspec = 'Filter order and transition width';
Order = 51;
TW = 4.1e3;
iirhalfbandinterp = dsp.IIRHalfbandInterpolator(...
                                               'Specification',filterspec,...
                                               'FilterOrder',Order,...
                                               'TransitionWidth',TW,...
                                                'SampleRate',Fs);

x = randn(1024,4);
y = iirhalfbandinterp(x);
```

dsp.Interpolator System object

Package: dsp

Linear or FIR interpolation

Description

The `Interpolator` object interpolates real-valued inputs using linear or polyphase FIR interpolation.

To interpolate real-valued inputs:

1. Define and set up your interpolation object. See "Construction" on page 4-1052.
2. Call step to interpolate the input according to the properties of `dsp.Interpolator`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`interp = dsp.Interpolator` returns a handle to an interpolation object, `interp`. The interpolation method defaults to `Linear` and the interpolation points default to a 5×1 vector.

`interp = dsp.Interpolator('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an interpolation object, `interp`, with the specified property name and value pairs.

Properties

`InterpolationPointsSource`

Source of interpolation points

You can select a value of `Property` or `Input port`. If you set this property to `Input port`, you must provide the interpolation points as an input to the `step` method when you interpolate the input. The default is `Property`.

InterpolationPoints

Interpolation points

This property only applies when you set the `InterpolationPointsSource` on page 4-1052 property to `Property`. Specify the interpolation points as a column vector or a matrix with the same number of channels as the input.

The valid range of the values in the interpolation vector is from `1` to the number of samples in each channel of the input. If you specify interpolation points outside the valid range, the object *clips* the point to the nearest point in the valid range. For example, if the input is `[2 3.1 -2.1]`, the valid range of interpolation points is from `1` to `3`. If you specify the following vector of interpolation points `[-1 1.5 2 2.5 3 3.5]`, the interpolator object clips `-1` to `1` and `3.5` to `3`. This clipping results in the interpolation points `[1 1.5 2 2.5 3 3]`. The default is `[1.1;4.8;2.67;1.6;3.2]`. This property is tunable.

Method

Interpolation method

Specify the interpolation method as `Linear` or `FIR`. When this property is `Linear`, the interpolator object interpolates data values by assuming that the data varies linearly between adjacent samples. When this property is `FIR`, the interpolator object uses polyphase interpolation to replace filtering (convolution) at the upsampled rate with a series of convolutions at the lower rate. The interpolator object always performs linear interpolation if there are insufficient low-rate samples to perform FIR interpolation as explained in the description of the `FilterHalfLength` on page 4-1053 property. The default is `Linear`.

FilterHalfLength

Interpolation filter half length

This property applies only when you set the `Method` on page 4-1053 property to `FIR`. For a filter half length of P, the polyphase FIR subfilters have length $2P$. FIR interpolation always requires $2P$ low-rate samples for every interpolation point.

- If the interpolation point does not correspond to a low-rate sample, FIR interpolation requires P low-rate samples below and P low-rate samples above the interpolation point.
- If the interpolation point corresponds to a low-rate sample, the $2P$-sample requirement includes the low-rate sample.
- If there are less than $2P$ neighboring low-rate samples, the interpolator object uses linear interpolation.

For example, for an input `[1 4 1 4 1 4 1 4]`, upsampling by a factor of four results in the equally spaced interpolation points `InterP = 1:0.25:8;`. The points `InterP(9:12)` are `[3.0 3.25 3.5 3.75]`. Assuming a `FilterHalfLength` property equal to 2, interpolating at these points uses the four low-rate samples from the input with indices `(2,3,4,5)`. If you set the `FilterHalfLength` property to 4 in this case, the interpolator object uses linear interpolation because there are not enough low-rate samples to perform FIR interpolation.

The longer the `FilterHalfLength` property, the better the quality of the interpolation. However, the costs are increased computation and requiring more low-rate samples below and above the interpolation point. In general, setting the `FilterHalfLength` property between 4 and 6 provides reasonably accurate interpolation.

InterpolationPointsPerSample

Interpolation points per input sample

This property only applies when you set the `Method` on page 4-1053 property to `FIR` and indicates the upsampling factor, L. An upsampling factor of L inserts L–1 zeros between low-rate samples. Interpolation results from filtering the upsampled sequence with a lowpass anti-imaging filter. The interpolator object uses a polyphase FIR implementation with `InterpolationPointsPerSample` subfilters of length $2P$, where P is the `FilterHalfLength` on page 4-1053 property. Denoting the low-rate samples in the upsampled input by nL, $n=1,2,...$, the interpolator object uses exactly one of the `InterpolationPointsPerSample` subfilters, or filter arms, to interpolate at the points $nL+i/L$, where $i=0,1,2,...,L-1$.

If you specify interpolation points which do not correspond to a polyphase subfilter, the object rounds the point down to the nearest interpolation point associated with a polyphase subfilter. For example, you set the `InterpolationPointsPerSample` property to `4` and interpolate at the points `[3 3.2 3.4 3.6 3.8]`. The interpolator object uses the first polyphase subfilter for the points `[3.0 3.2]`, the second subfilter for the point `3.4`, the third subfilter for the point `3.6`, and the fourth subfilter for the point `3.8`.

Bandwidth

Normalized input bandwidth

This property applies only when you set the `Method` on page 4-1053 property to `FIR`. The normalized bandwidth is a real-valued scalar between 0 and 1, where 1 equals the Nyquist frequency, or 1/2 the sampling frequency (*Fs*). You may know that your input does not have frequency content above some cutoff frequency less than the Nyquist frequency. You can use this information to improve the FIR interpolation filters by relaxing the stopband requirements in frequency regions where the signal has no energy. For example, if the input signal does not have frequency content above *Fs/8*, you can specify a value of `0.25` for the `Bandwidth` property. The default value is `0.5`.

Methods

step	Linear or FIR interpolation

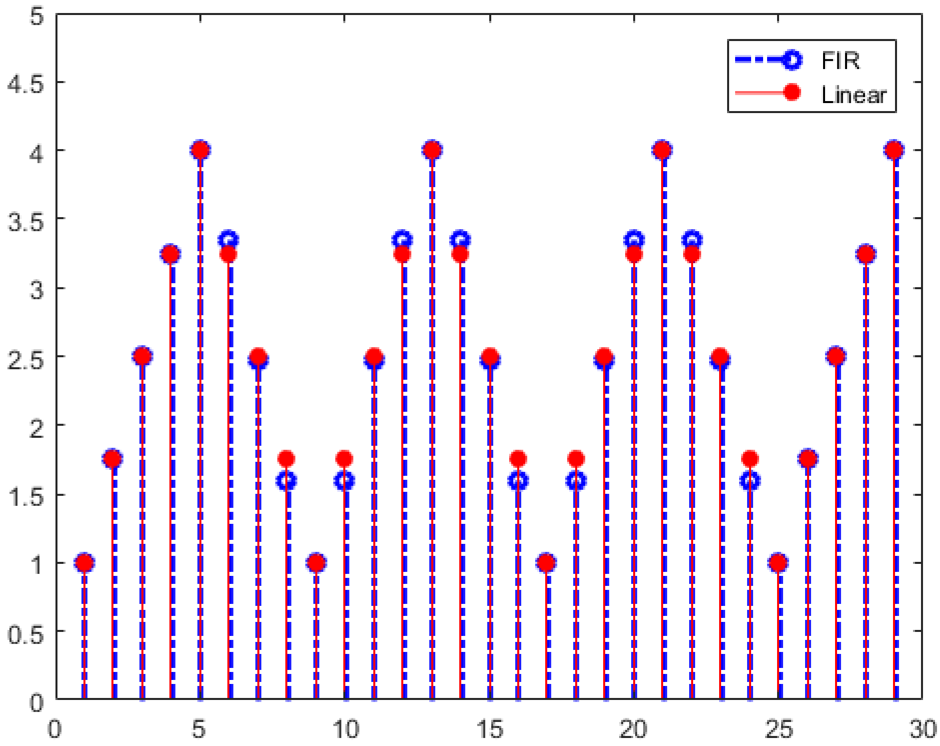
Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Compare Linear Interpolation With FIR Interpolation

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
x =[1 4];
x = repmat(x,1,4);
x1 = 1:0.25:8;
firInterp =dsp.Interpolator('Method','FIR','FilterHalfLength',2,...
'InterpolationPoints',x1','InterpolationPointsPerSample',4);
linInterp = dsp.Interpolator('InterpolationPoints',x1');
OutFIR = firInterp(x');
OutLin = linInterp(x');
stem(OutFIR,'b-.','linewidth',2); hold on;
stem(OutLin,'r','markerfacecolor',[1 0 0]);
axis([0 30 0 5]); legend('FIR','Linear','Location','Northeast');
```

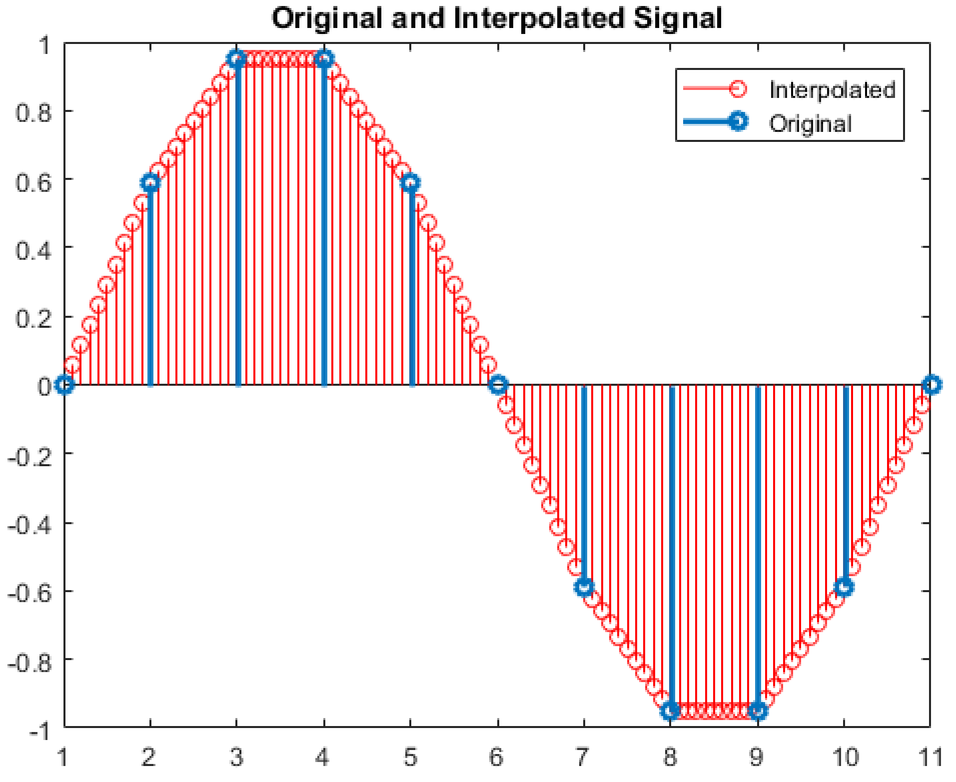


For the indices 1 to 5 and 25 to 29, the interpolator object uses linear interpolation in both cases. The reason for this is that there are not enough low-rate samples surrounding the interpolation points at those indices to use FIR interpolation with the specified filter length.

Interpolate a Sinusoid With Linear Interpolation

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
t = 0:.0001:.0511;
x = sin(2*pi*20*t);
x1 = x(1:50:end);
I = 1:0.1:length(x1);
interp = dsp.Interpolator('InterpolationPointsSource',...
    'Input port');
y = interp(x1',I');
stem(I',y, 'r');
title('Original and Interpolated Signal');
hold on; stem(x1, 'Linewidth', 2);
legend('Interpolated','Original');
```



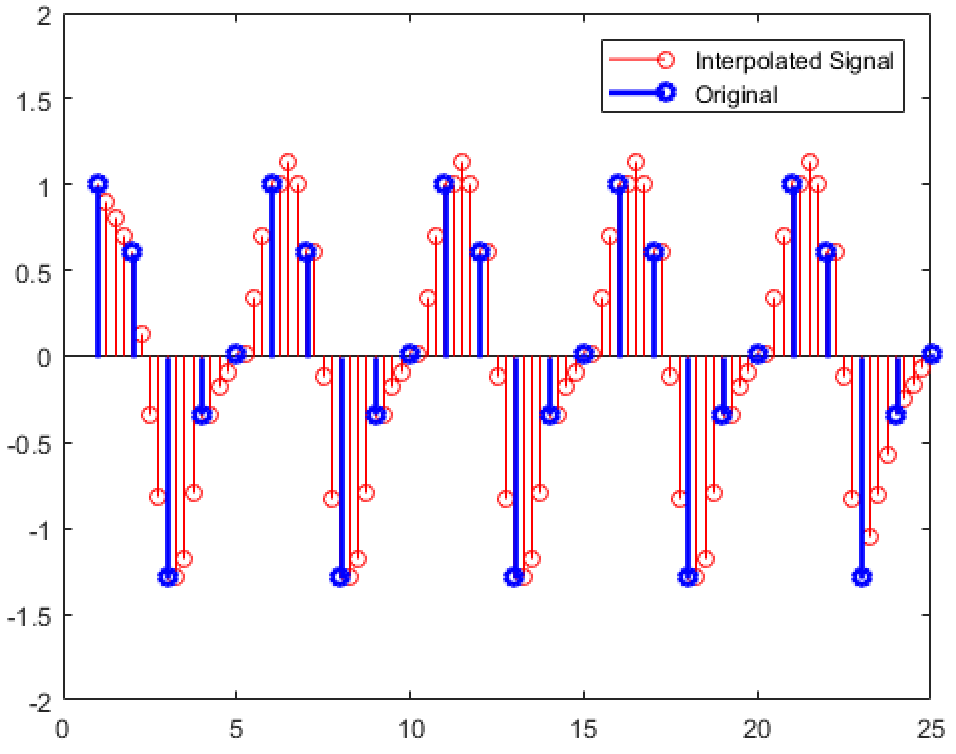
Interpolate a Sum of Sinusoids

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Interpolate a sum of sinusoids with FIR interpolation and `Input Port` as the source of interpolation points.

```
Fs = 1000;
t = 0:(1/Fs):0.1-(1/Fs);
x = cos(2*pi*50*t)+0.5*sin(2*pi*100*t);
x1 = x(1:4:end);
I = 1:(1/4):length(x1);
```

```
interp = dsp.Interpolator('Method','FIR',...
'FilterHalfLength',3,'InterpolationPointsSource','Input Port');
y = interp(x1',I');
stem(I,y,'r'); hold on;
axis([0 25 -2 2]);
stem(x1,'b','linewidth',2);
legend('Interpolated Signal','Original',...
'Location','Northeast');
```



Definitions

Polyphase Implementation

A polyphase implementation of an FIR interpolation filter *splits* the lowpass FIR filter impulse response into a number of different subfilters.

Let L represent the number of interpolation points per sample, or the upsampling factor, and P the half length of the polyphase subfilters. Indexing from zero, if *h(n)* is the impulse response of the FIR filter, the k-th subfilter is:

$$h_k(n) = h(k + nL) \quad k = 0,1,\ldots,L-1 \quad n = 0,1,\ldots,2P-1$$

The following table describes the decomposition of an 18-coefficient FIR filter into 3 polyphase subfilters of length 6, the defaults for the FIR interpolator object:

Coefficients	Polyphase Subfilter
h(0),h(3),h(6), ..., h(15)	$h_0(n)$
h(1),h(4),h(7), ..., h(16)	$h_1(n)$
h(2),h(5),h(8), ..., h(17)	$h_2(n)$

The following code shows how to find the polyphase subfilters for the default FIR interpolator object:

```
interp = dsp.Interpolator('Method','FIR');
L = interp.InterpolationPointsPerSample;
P = interp.FilterHalfLength;
FiltCoeffs = intfilt(L,P,interp.Bandwidth);
% Returns filter of length 2*P*L-1
FiltLen=length(FiltCoeffs);
FiltCols = ceil(FiltLen/2/L);

% We need 2*P*L coefficients
% Prepending a zero does not affect the filter magnitude
FiltCoeffs = [zeros(FiltCols*2*L-FiltLen,1); FiltCoeffs(:)];

% Each column of PolyPhaseCoeffs is a polyphase subfilter
PolyPhaseCoeffs  = reshape(FiltCoeffs,FiltCols,2*L)';
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Interpolation` block reference page. The object properties correspond to the Simulink block parameters, except:

Out of range interpolation points — The interpolator object only has the `Clip` option. The Simulink block has the additional `Clip and warn` and `Error` options.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

`intfilt` | dsp.FIRInterpolator | dsp.VariableFractionalDelay

Introduced in R2012a

step

System object: dsp.Interpolator
Package: dsp

Linear or FIR interpolation

Syntax

```
Y = step(interp,X)
Y = step(interp,X,IPTS)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(interp,X)` outputs the interpolated sequence, `Y`, of the input vector or matrix `X` as specified in the `InterpolationPoints` property.

`Y = step(interp,X,IPTS)` outputs the interpolated sequence as specified by the input argument `IPTS` when the `InterpolationPointsSource` property is set to 'Input port'. `IPTS` a column vector or a matrix of interpolation points. If `IPTS` is a matrix, it must have the same number of channels as the input.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.KalmanFilter System object

Package: dsp

Estimate system measurements and states using Kalman filter

Description

The `KalmanFilter` is an estimator used to recursively obtain a solution for linear optimal filtering. This estimation is made without precise knowledge of the underlying dynamic system. The Kalman filter implements the following linear discrete-time process with state, x, at the k^{th} time-step: $x(k) = Ax(k-1) + Bu(k-1) + w(k-1)$ (state equation). This measurement, z, is given as: $z(k) = Hx(k) + v(k)$ (measurement equation).

The Kalman filter algorithm computes the following two steps recursively:

- Prediction: Process parameters x (state) and P (state error covariance) are estimated using the previous state,
- Correction: The state and error covariance are corrected using the current measurement,

To filter each channel of the input:

1. Define and set up your Kalman filter. See “Construction” on page 4-1064.
2. Call step to filter each channel of the input according to the properties of `dsp.KalmanFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`kalman = dsp.KalmanFilter` returns the Kalman filter System object, `kalman`, with default values for the parameters.

`kalman = dsp.KalmanFilter('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns an Kalman filter System object, `kalman`, with each property set to the specified value.

`kalman = dsp.KalmanFilter(STMatrix, MMatrix, PNCovariance, MNCovariance, CIMatrix, 'PropertyName', PropertyValue, ...)` returns a Kalman filter System object, `kalman`. The `StateTransitionMatrix` property is set to `STMatrix`, and `MeasurementMatrix` property is set to `MMatrix`. In addition, the `ProcessNoiseCovariance` property set to `PNCovariance`, `MeasurementNoiseCovariance` property set to `MNCovariance`, `ControlInputMatrix` property set to `CIMatrix`. Other specified properties are set to the specified values.

Properties

`StateTransitionMatrix`

Model of state transition

Specify `A` in the state equation that relates the state at the previous time step to the state at current time step. `A` is a square matrix with each dimension equal to the number of states. The default value is `1`.

`ControlInputMatrix`

Model of relation between control input and states

Specify `B` in the state equation that relates the control input to the state. `B` is a matrix with a number of rows equal to the number of states. This property is activated only when the `ControlInputPort` property value is `true`. The default value is `1`.

`MeasurementMatrix`

Model of relation between states and measurement output

Specify `H` in the measurement equation that relates the states to the measurements. `H` is a matrix with the number of columns equal to the number of measurements. The default value is `1`.

`ProcessNoiseCovariance`

Covariance of process noise

Specify `Q` as a square matrix with each dimension equal to the number of states. This matrix, `Q`, is the covariance of the white Gaussian process noise, `w`, in the state equation. The default value is `0.1`.

`MeasurementNoiseCovariance`

Covariance of measurement noise

Specify `R` as a square matrix with each dimension equal to the number of states. This matrix, `R`, is the covariance of the white Gaussian process noise, `v`, in the measurement equation. The default value is `0.1`.

`InitialStateEstimate`

Initial value for states

Specify an initial estimate of the states of the model as a column vector with length equal to the number of states. The default value is `0`.

`InitialErrorCovarianceEstimate`

Initial value for state error covariance

Specify an initial estimate for covariance of the state error, as a square matrix with each dimension equal to the number of states. The default value is `0.1`.

`DisableCorrection`

Disable port for filters

Specify as a scalar logical value, disabling System object filters from performing the correction step after the prediction step in the Kalman filter algorithm. The default value is `false`.

`ControlInputPort`

Presence of a control input

Specify if the control input is present, using a scalar logical value. The default value is `true`.

Methods

reset	Reset internal states of Kalman filter

step	Filter input with Kalman filter object

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Estimate a Changing Scalar

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

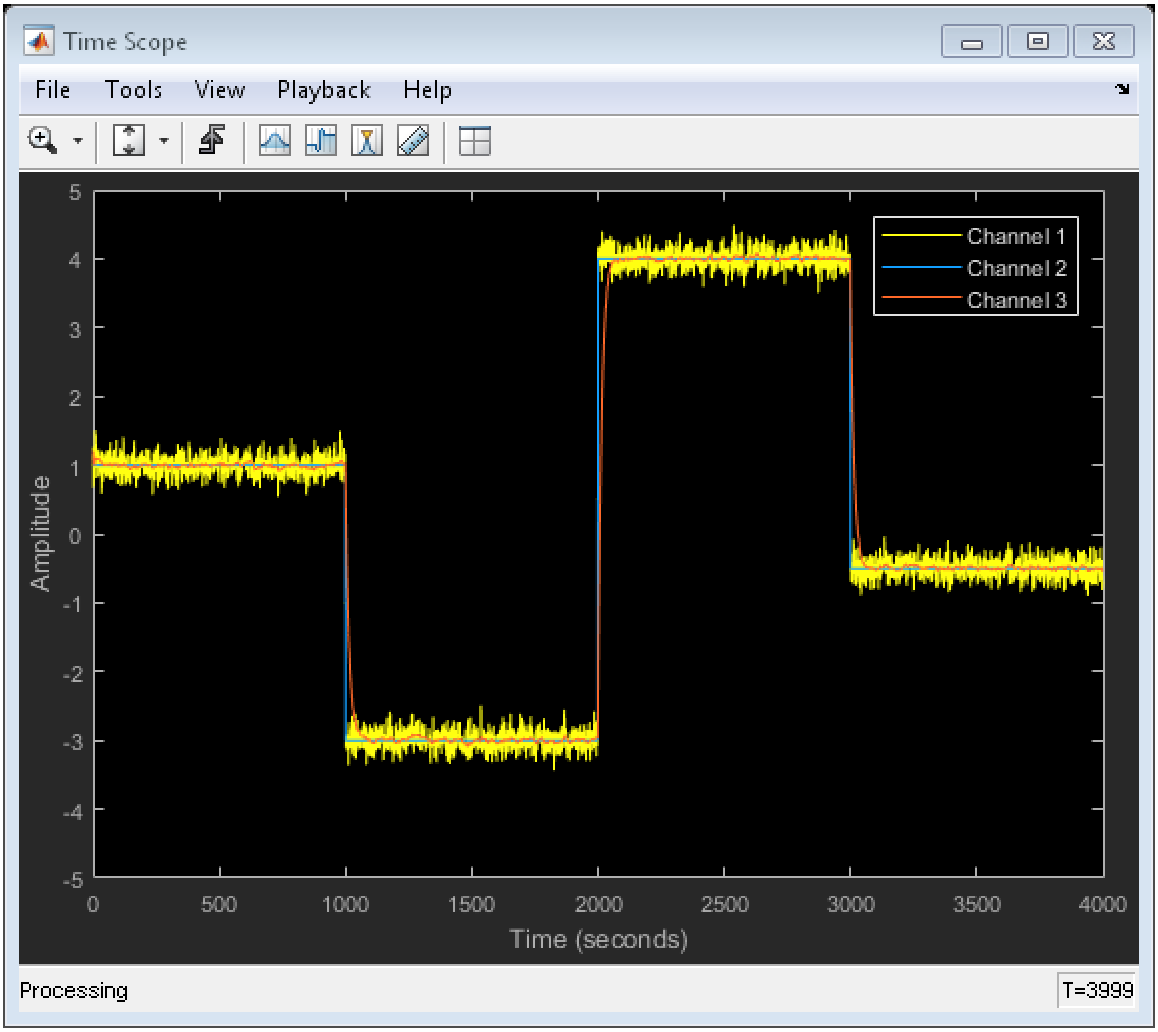
Create the System objects for the changing scalar input, the Kalman filter, and the scope (for plotting).

```
numSamples = 4000;
R = 0.02;
src = dsp.SignalSource;
src.Signal = [ones(numSamples/4,1);   -3*ones(numSamples/4,1);...
    4*ones(numSamples/4,1); -0.5*ones(numSamples/4,1)];
tScope = dsp.TimeScope('NumInputPorts', 3, 'TimeSpan', numSamples, ...
    'TimeUnits', 'Seconds', 'YLimits',[-5 5], ...
    'ShowLegend', true); % Create the Time Scope
kalman = dsp.KalmanFilter('ProcessNoiseCovariance', 0.0001,...
    'MeasurementNoiseCovariance', R,...
    'InitialStateEstimate', 5,...
    'InitialErrorCovarianceEstimate', 1,...
    'ControlInputPort',false); %Create Kalman filter
```

Add noise to the scalar, and pass the result to the Kalman filter. Stream the data, and plot the filtered signal.

```
while(~isDone(src))
   trueVal = src();
   noisyVal = trueVal + sqrt(R)*randn;
```

```
   estVal = kalman(noisyVal);
   tScope(noisyVal,trueVal,estVal);
end
```



Disable the Correction Step in the Kalman Filter Algorithm

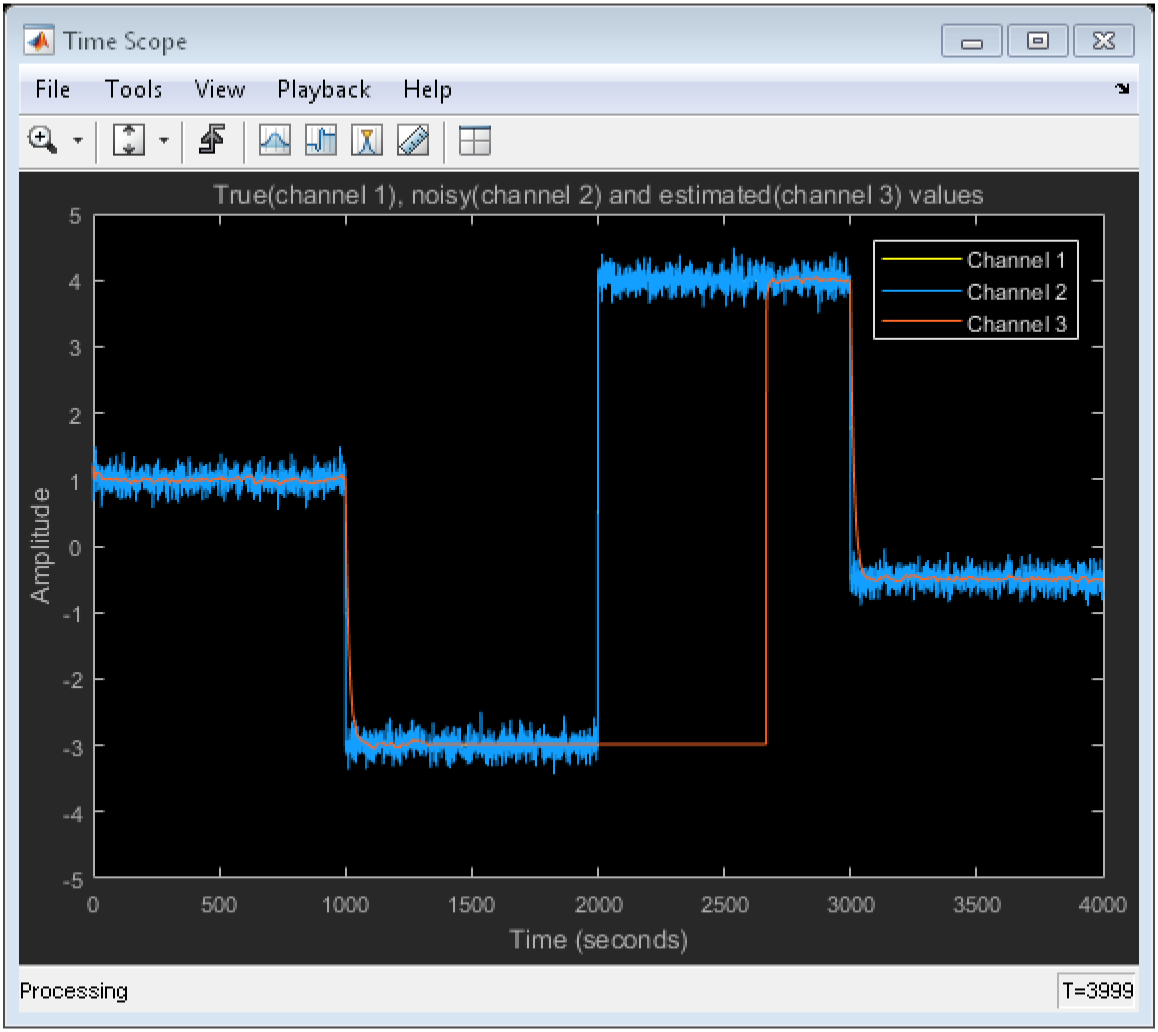
Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create the signal, Kalman Filter, and Time Scope System objects.

```
numSamples = 4000;
R = 0.02;
src = dsp.SignalSource;
src.Signal = [  ones(numSamples/4,1);   -3*ones(numSamples/4,1);...
              4*ones(numSamples/4,1); -0.5*ones(numSamples/4,1)];
tScope = dsp.TimeScope('NumInputPorts', 3, 'TimeSpan', numSamples, ...
          'TimeUnits', 'Seconds', 'YLimits',[-5 5], ...
          'Title', ['True(channel 1), noisy(channel 2) and ',...
          'estimated(channel 3) values'], ...
          'ShowLegend', true);
kalman = dsp.KalmanFilter('ProcessNoiseCovariance', 0.0001,...
          'MeasurementNoiseCovariance', R,...
          'InitialStateEstimate', 5,...
          'InitialErrorCovarianceEstimate', 1,...
          'ControlInputPort',false);
ctr = 0;
```

Add noise to the signal. Stream the data, and plot the filtered signal.

```
while(~isDone(src))
    trueVal = src();
    noisyVal = trueVal + sqrt(R)*randn;
    estVal = kalman(noisyVal);
    tScope(trueVal,noisyVal,estVal);

    % Disabling the correction step of second filter for the middle
    % one-third of the simulation
    if ctr == floor(numSamples/3)
        kalman.DisableCorrection = true;
    end
    if ctr == floor(2*numSamples/3)
        kalman.DisableCorrection = false;
    end
    ctr = ctr + 1;
end
```



Use a Single System object to Track Multiple Scalar Values

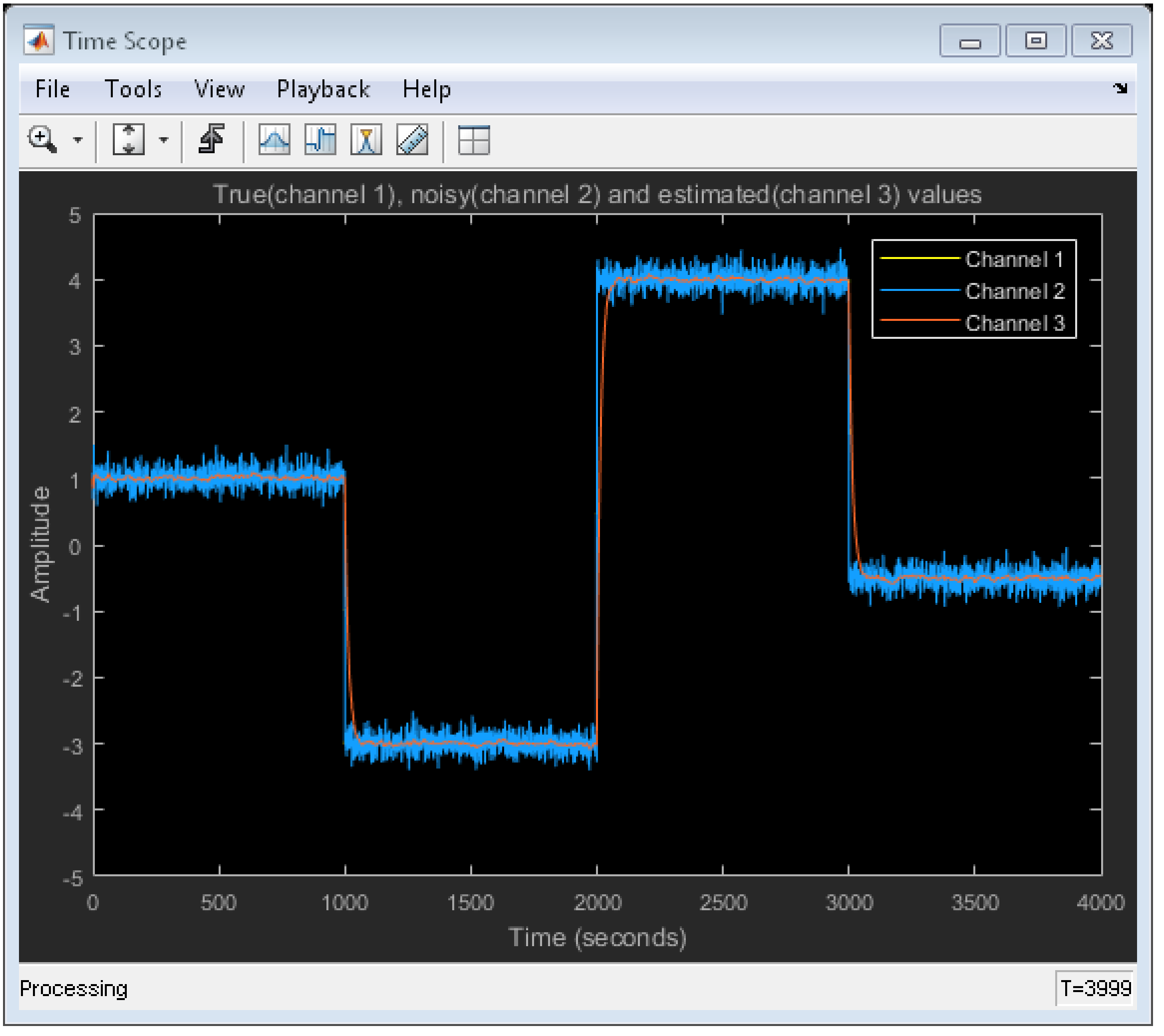
Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

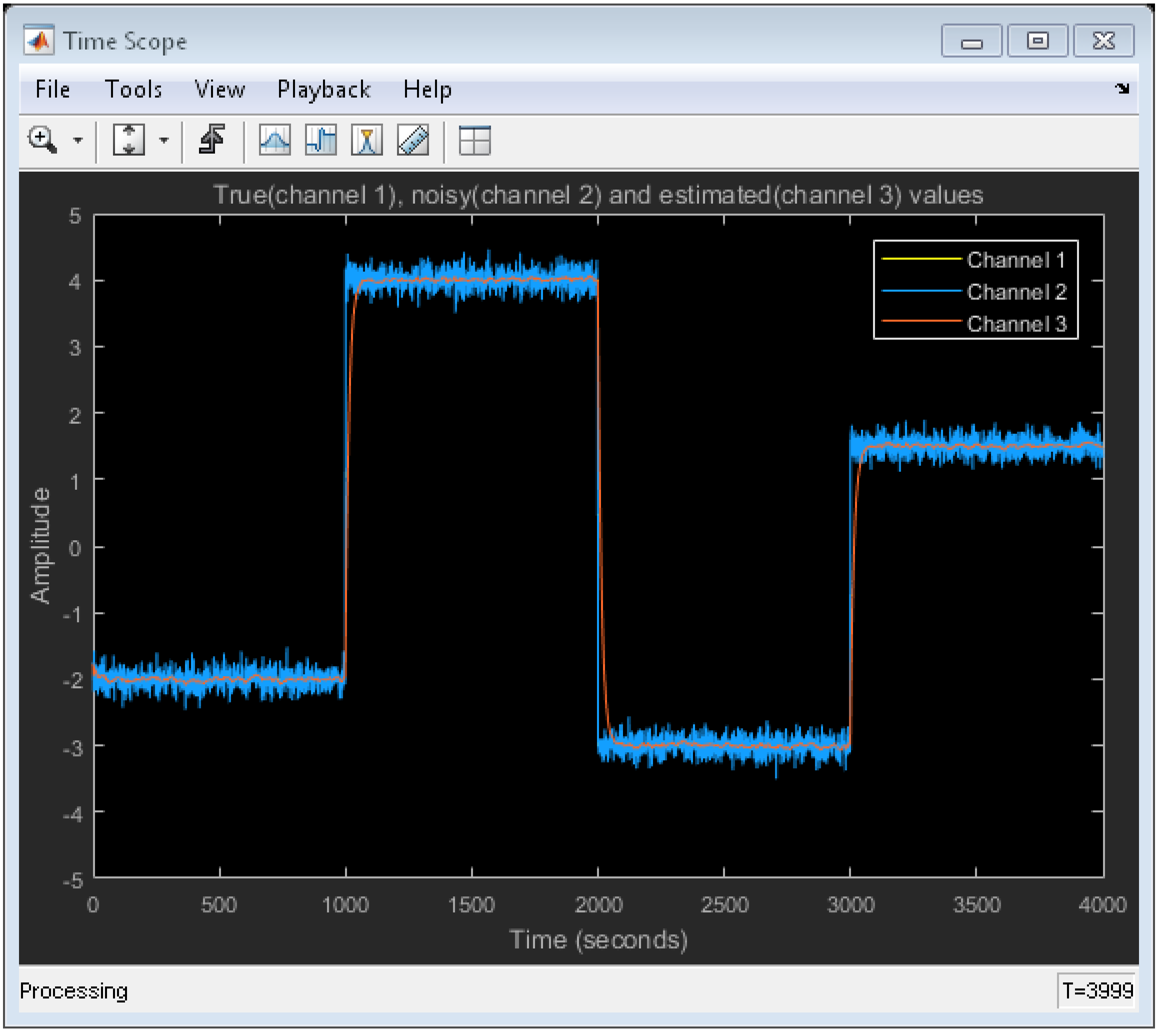
Create the signal where the columns are the two scalar values to be tracked. Also create the Kalman Filter, and the Time Scopes.

```
numSamples = 4000;
R = 0.02;
src = dsp.SignalSource;
sig1 = [  ones(numSamples/4,1);   -3*ones(numSamples/4,1);...
         4*ones(numSamples/4,1); -0.5*ones(numSamples/4,1)];
sig2 = [-2*ones(numSamples/4,1);   4*ones(numSamples/4,1);...
        -3*ones(numSamples/4,1); 1.5*ones(numSamples/4,1)];

src.Signal = [sig1, sig2];

tScope1 = dsp.TimeScope('NumInputPorts', 3, 'TimeSpan', numSamples, ...
           'TimeUnits', 'Seconds', 'YLimits',[-5 5], ...
           'Title', ['True(channel 1), noisy(channel 2) and ',...
           'estimated(channel 3) values'], ...
           'ShowLegend', true);
tScope2 = clone(tScope1);
kalman = dsp.KalmanFilter('ProcessNoiseCovariance', 0.0001,...
    'MeasurementNoiseCovariance', R,...
    'InitialStateEstimate', -3,...
    'InitialErrorCovarianceEstimate', 1,...
    'ControlInputPort',false);
```

Add noise to the signal. Stream the data, and plot the filtered signal.

```
while(~isDone(src))
    trueVal = src();
    noisyVal = trueVal + sqrt(R)*randn(1,2);
    estVal = kalman(noisyVal);
           % Plot results of first channel on Time Scope
    tScope1(trueVal(:,1),noisyVal(:,1),estVal(:,1));
           % Plot results of second channel on Time Scope
    tScope2(trueVal(:,2),noisyVal(:,2),estVal(:,2));
end
```


Time Scope
File Tools View Playback Help
True(channel 1), noisy(channel 2) and estimated(channel 3) values
Channel 1
Channel 2
Channel 3
Amplitude
Time (seconds)
5
4
3
2
1
0
-1
-2
-3
-4
-5
0 500 1000 1500 2000 2500 3000 3500 4000
Processing
T=3999



Use a Unit Step To Track a Ramp Signal

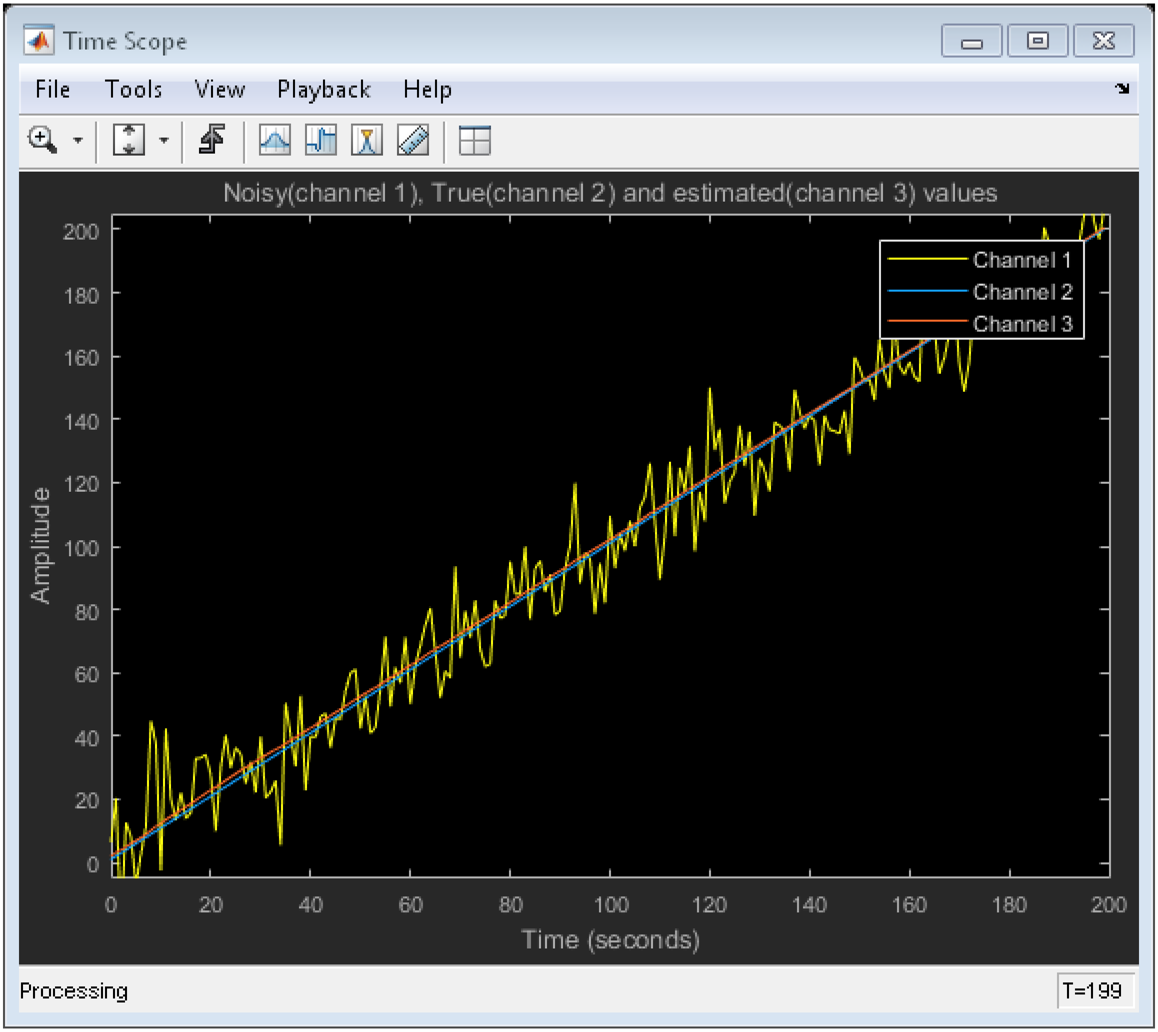
Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Use a unit step as the control input to track a ramp signal. Create the ramp signal to be tracked, the control input, the Time Scope, and the Kalman Filter.

```
numSamples = 200;
R = 100;
src = dsp.SignalSource;
src.Signal = (1:numSamples)';
control = dsp.SignalSource;
control.Signal = ones(numSamples,1);

tScope = dsp.TimeScope('NumInputPorts', 3, 'TimeSpan', numSamples, ...
    'TimeUnits', 'Seconds', 'YLimits',[-5 205], ...
    'Title', ['Noisy(channel 1), True(channel 2) and ',...
    'estimated(channel 3) values'], ...
    'ShowLegend', true);
kalman = dsp.KalmanFilter('ProcessNoiseCovariance', 0.0001,...
    'MeasurementNoiseCovariance', R,...
    'InitialStateEstimate', 1,...
    'InitialErrorCovarianceEstimate', 1);
```

Add noise to the signal. Filter the signal using kalman filter. View the output using time scope.

```
while(~isDone(src))
    trueVal = src();
    ctrl = control();
    noisyVal = trueVal + sqrt(R)*randn;
    estVal = kalman(noisyVal,ctrl);
    tScope(noisyVal,trueVal,estVal);
end
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Kalman Filter` block reference page. The object properties correspond to the block parameters.

References

[1] Greg Welch and Gary Bishop, *An Introduction to the Kalman Filter*, Technical Report TR95 041. University of North Carolina at Chapel Hill: Chapel Hill, NC., 1995.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`Kalman Filter`

Introduced in R2013b

reset

System object: dsp.KalmanFilter
Package: dsp

Reset internal states of Kalman filter

Syntax

`reset(kalman)`

Description

`reset(kalman)` resets the internal states of the Kalman filter object, `kalman`, to their initial values. The reset method is always a no-op for unlocked System objects, as the states may not be allocated when the object is not locked.

step

System object: dsp.KalmanFilter
Package: dsp

Filter input with Kalman filter object

Syntax

```
Y = step(kalman,X)
[Y1,...,YN] = step(kalman,X)
[zEst, xEst, MSE_Est, zPred, xPred, MSE_Pred] = step(kalman, z, u)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(kalman,X)` processes the input data, `X` to produce the output, `Y`, for System object, `kalman`.

`[Y1,...,YN] = step(kalman,X)` produces `N` outputs.

`[zEst, xEst, MSE_Est, zPred, xPred, MSE_Pred] = step(kalman, z, u)` Carries out the iterative Kalman filter algorithm over measurements `z` and control inputs `u`. The columns in `z` and `u` are treated as inputs to separate parallel filters, whose correction (or update) step can be disabled by the `DisableCorrection` property. The values returned are estimated measurements `z_est`, estimated states `x_est`, MSE of estimated states `MSE_Est`, predicted measurements `zPred`, predicted states `xPred` and MSE of predicted states `MSE_Pred`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.LDLFactor System object

Package: dsp

Factor square Hermitian positive definite matrices into components

Description

The `LDLFactor` object factors square Hermitian positive definite matrices into lower, upper, and diagonal components. The object uses only the lower triangle of S.

To factor these matrices into lower, upper, and diagonal components:

1. Define and set up your LDL factor object. See "Construction" on page 4-1081.
2. Call step to factor the matrices according to the properties of `dsp.LDLFactor`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ldl = dsp.LDLFactor` returns an LDL factor System object, `ldl`, that computes unit lower triangular `L` and diagonal `D` such that `S` = LDL for square, symmetric/Hermitian, positive definite input matrix `S`.

`ldl = dsp.LDLFactor('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an `LDL` factor System object, `ldl`, with each specified property set to the specified value.

Properties

Fixed-Point Properties

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Full precision`, `Same as input`, `Same as product` or `Custom`. The default is `Full precision`

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType on page 4-1082` property to `Custom`. The default is `numerictype([],32,30)`.

CustomIntermediateProductDataType

Intermediate product word and fraction lengths

Specify the intermediate product fixed-point type as a signed, scaled `numerictype` object. This property applies when you set the `IntermediateProductDataType on page 4-1083` property to `Custom`. The default is `numerictype(true,16,15)`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType on page 4-1083` property to `Custom`. The default is `numerictype([],16,15)`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType on page 4-1083` property to `Custom`. The default is `numerictype([],32,30)`.

IntermediateProductDataType

Intermediate product word and fraction lengths

Specify the intermediate product fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` or `Saturate`. The default is `Wrap`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as `Full precision`, `Same as input` or `Custom`. The default is `Full precision`.

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as: `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest` or `Zero`. The default is `Floor`.

Methods

step	Decompose matrix into components

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object

Common to All System Objects	
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Decompose a Matrix

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Decompose a square Hermitian positive definite matrix using LDL factor.

```
A = gallery('randcorr',5);
ldl = dsp.LDLFactor;
y = ldl(A);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `LDL Factorization` block reference page. The object properties correspond to the block parameters, except:
No object property that corresponds to the **Non-positive definite input** block parameter. The object does not issue any alerts for nonpositive definite inputs. The output is not a valid factorization. A partial factorization is in the upper left corner of the output.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.LUFactor

Introduced in R2012a

step

System object: dsp.LDLFactor
Package: dsp

Decompose matrix into components

Syntax

`Y = step(ldl,S)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(ldl,S)` decomposes the matrix `S` into lower, upper, and diagonal components. The output `Y` is a composite matrix with the `L` as its lower triangular part and `D` as the diagonal and `L'` as its upper triangular part. If `S` is not positive definite the output `Y` is not a valid factorization.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.LevinsonSolver System object

Package: dsp

Solve linear system of equations using Levinson-Durbin recursion

Description

The `LevinsonSolver` object solves linear systems of equations using Levinson-Durbin recursion.

To solve linear systems of equations using Levinson-Durbin recursion:

1. Define and set up your System object. See "Construction" on page 4-1087.
2. Call step to solve the system of equations according to the properties of `dsp.LevinsonSolver`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`levinson = dsp.LevinsonSolver` returns a System object, `levinson`, that solves a Hermitian Toeplitz system of equations using the Levinson-Durbin recursion.

`levinson = dsp.LevinsonSolver('PropertyName',PropertyValue,...)` returns a Levinson-Durbin object, `levinson`, with each specified property set to the specified value.

Properties

`AOutputPort`

Enable polynomial coefficients output

Set this property to `true` to output the polynomial coefficients *A*. Both `AOutputPort` and `KOutputPort on page 4-1088` properties cannot be `false` at the same time. For scalar inputs, set the `AOutputPort` property to `true`. The default is `false`.

KOutputPort

Enable reflection coefficients output

Set this property to `true` to output the reflection coefficients *K*. You cannot set both the `AOutputPort on page 4-1087` and `KOutputPort` properties to `false` at the same time. For scalar inputs, you must set the `KOutputPort` property to `false`. The default is `true`.

PredictionErrorOutputPort

Enable prediction error output

Set this property to `true` to output the prediction error. The default is `false`.

ZerothLagZeroAction

Action when value of lag zero is zero

Specify the output for an input with the first coefficient as zero. Select `Ignore` or `Use zeros`. The default is `Use zeros`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap`, `Saturate`. The default is `Wrap`.

ACoefficientDataType

A coefficient word and fraction lengths

This constant property has a value of `Custom`.

CustomACoefficientDataType

A coefficient word and fraction lengths

Specify the *A* coefficient fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. The default is `numerictype([],16,15)`.

KCoefficientDataType

K coefficient word and fraction lengths

This constant property has a value of `Custom`.

CustomKCoefficientDataType

K coefficient word and fraction lengths

Specify the *K* coefficient fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. The default is `numerictype([],16,15)`.

PredictionErrorDataType

Prediction error power word and fraction lengths

Specify the prediction error power fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

CustomPredictionErrorDataType

Prediction error power word and fraction lengths

Specify the prediction error power fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `PredictionErrorDataType on page 4-1089` property is `Custom`. The default is `numerictype([],16,15)`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as `Same as input` or `Custom`. The default is `Custom`

`CustomProductDataType`

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `ProductDataType` on page 4-1089 property is `Custom`. The default is `numerictype([],32,30)`.

`AccumulatorDataType`

Accumulator word and fraction lengths

Specify the Accumulator fixed-point data type as `Same as input`, `Same as product`, or `Custom`. The default is `Custom`.

`CustomAccumulatorDataType`

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `AccumulatorDataType` on page 4-1090 property is `Custom`. The default is `numerictype([],32,30)`.

Methods

step	Reflection coefficients corresponding to columns of input

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Compute Polynomial Coefficients

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Use the Levinson solver to compute polynomial coefficients from autocorrelation coefficients.

```
levinson = dsp.LevinsonSolver;
levinson.AOutputPort = true;
levinson.KOutputPort = false;
x = (1:100)';
ac = dsp.Autocorrelator(...
'MaximumLagSource', 'Property', ...
'MaximumLag', 10);
a = ac(x);
c = levinson(a); % Compute polynomial coefficients
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Levinson-Durbin` block reference page. The object properties correspond to the block parameters, except:
Output(s) block parameter corresponds to the `AOutputPort` and the `KOutputPort` object properties.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

dsp.Autocorrelator

Introduced in R2012a

step

System object: dsp.LevinsonSolver
Package: dsp

Reflection coefficients corresponding to columns of input

Syntax

```
K = step(levinson,X)
A = step(levinson,X)
[A, K] = step(levinson,X)
[..., P] = step(levinson,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`K = step(levinson,X)` returns reflection coefficients `K` corresponding to the columns of input `X`. `X` is typically a column or matrix of autocorrelation coefficients with lag `0` as the first element.

`A = step(levinson,X)` returns polynomial coefficients `A` when the `AOutputPort` property is `true` and the `KOutputPort` property is `false`.

`[A, K] = step(levinson,X)` returns polynomial coefficients `A` and reflection coefficients `K` when both the `AOutputPort` and `KOutputPort` properties are `true`.

`[..., P] = step(levinson,X)` also returns the error power `P` when the `PredictionErrorOutputPort` property is `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.LMSFilter System object

Package: dsp

LMS adaptive filter

Description

The `LMSFilter` implements an adaptive FIR filter object that returns the filtered output, the error vector, and filter weights. The LMS filter uses one of five different LMS algorithms.

To implement the adaptive FIR filter object:

1. Define and set up your adaptive FIR filter object. See "Construction" on page 4-1095.
2. Call step to implement the filter according to the properties of `dsp.LMSFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`lms = dsp.LMSFilter` returns an adaptive FIR filter object, `lms`, that computes the filtered output, filter error and the filter weights for a given input and desired signal using the Least Mean Squares (LMS) algorithm.

`lms = dsp.LMSFilter('`*`PropertyName`*`', `*`PropertyValue`*`,...)` returns an LMS filter object, `lms`, with each property set to the specified value.

`lms = dsp.LMSFilter(LEN,'`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an LMS filter object, `lms`, with the `Length` property set to `LEN`, and other specified properties set to the specified values.

Properties

Method

Method to calculate filter weights

Specify the method used to calculate filter weights as LMS, Normalized LMS, Sign-Error LMS, Sign-Data LMS, or Sign-Sign LMS. The default is LMS.

Length

Length of FIR filter weights vector

Specify the length of the FIR filter weights vector as a positive integer. The default is 32.

StepSizeSource

How to specify adaptation step size

Choose how to specify the adaptation step size factor as Property or Input port. The default is Property.

StepSize

Adaptation step size

Specify the adaptation step size factor as a nonnegative real number. For convergence of the normalized LMS method, set the step size greater than 0 and less than 2. This property only applies when the StepSizeSource on page 4-1096 property is Property. The default is 0.1. This property is tunable.

LeakageFactor

Leakage factor used in LMS filter

Specify the leakage factor as a real number between 0 and 1 inclusive. A leakage factor of 1 corresponds to no leakage in the adapting method. The default is 1. This property is tunable.

InitialConditions

Initial conditions of filter weights

Specify the initial values of the FIR filter weights as a scalar or vector of length equal to the `Length` property value. The default is `0`.

AdaptInputPort

Enable weight adaptation

Specify when the LMS filter should adapt the filter weights. By default, the value of this property is `false`, and the object continuously updates the filter weights. When this property is set to `true`, an adaptation control input is provided to the `step` method. If the value of this input is nonzero, the object continuously updates the filter weights. If the input is zero, the filter weights remain at their current value.

WeightsResetInputPort

Enable weight reset

Specify when the LMS filter should reset the filter weights. By default, the value of this property is `false`, and the object does not reset the weights. When this property is set to `true`, a reset control input is provided to the `step` method, and the `WeightsResetCondition` on page 4-1097 property applies. The object resets the filter weights based on the values of the `WeightsResetCondition` property and the `reset` input to the `step` method.

WeightsResetCondition

Reset trigger setting for filter weights

Specify the event to reset the filter weights as `Rising edge`, `Falling edge`, `Either edge`, or `Non-zero`. The LMS filter resets the filter weights based on the values of this property and the `reset` input to the `step` method. This property only applies when the `WeightsResetInputPort` on page 4-1097 property is `true`. The default is `Non-zero`.

WeightsOutput

Enable returning filter weights

Specify how to output the adapted filter weights as one of the following:

- `'Last'` (default) — The object returns a column vector of weights corresponding to the last sample of the data frame. The length of the weights vector is the value given by the `Length` property.

- 'All' — The object returns a *FrameLength*-by-*Length* matrix of weights. The matrix corresponds to the full sample-by-sample history of weights values for all *FrameLength* samples of the input values. Each row in the matrix corresponds to a set of LMS filter weights calculated for the corresponding input sample.
- 'None' — This setting disables the weights output.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as Ceiling, Convergent, Floor, Nearest, Round, Simplest, or Zero. The default is Floor.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as Wrap or Saturate. The default is Wrap.

StepSizeDataType

Step size word and fraction lengths

Specify the step size fixed-point data type as Same word length as first input or Custom. Setting this property also sets the LeakageFactorDataType on page 4-1098 property to the same value. This property only applies when the StepSizeSource on page 4-1096 property is Property. The default is Same word length as first input.

CustomStepSizeDataType

Step size word and fraction lengths

Specify the step size fixed-point type as a numerictype object with a Signedness of Auto. This property only applies when the StepSizeSource on page 4-1096 property is Property and the StepSizeDataType on page 4-1098 property is Custom. The default is numerictype([],16,15).

LeakageFactorDataType

Leakage factor word and fraction lengths

Specify the leakage factor fixed-point data type as `Same word length as first input` or `Custom`. Setting this property also sets the `StepSizeDataType` on page 4-1098 property to the same value. This property only applies when the `StepSizeSource` on page 4-1096 property is `Property`. The default is `Same word length as first input`.

CustomLeakageFactorDataType

Leakage factor word and fraction lengths

Specify the leakage factor fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property only applies when the `StepSizeSource` on page 4-1096 property is `Property` and the `LeakageFactorDataType` on page 4-1098 property is `Custom`. The default is `numerictype([],16,15)`.

WeightsDataType

Weights word and fraction lengths

Specify the filter weights fixed-point data type as `Same as first input` or `Custom`. The default is `Same as first input`.

CustomWeightsDataType

Weights word and fraction lengths

Specify the filter weights fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property only applies when the `WeightsDataType` on page 4-1099 property is `Custom`. The default is `numerictype([],16,15)`.

EnergyProductDataType

Energy product word and fraction lengths

Specify the energy product fixed-point data type as `Same as first input` or `Custom`. This property only applies when the `Method` property is `Normalized LMS`. Setting this property also sets the `ConvolutionProductDataType` on page 4-1100, `StepSizeErrorProductDataType` on page 4-1101, `WeightsUpdateProductDataType` on page 4-1101, and `QuotientDataType` on page 4-1102 properties to the same value. The default is `Same as first input`.

CustomEnergyProductDataType

Energy product word and fraction lengths

Specify the energy product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property only applies when the `Method` on page 4-1096 property is `Normalized LMS` and the `EnergyProductDataType` on page 4-1099 property is `Custom`. The default is `numerictype([],32,20)`.

EnergyAccumulatorDataType

Energy accumulator word and fraction lengths

Specify the energy accumulator fixed-point data type as `Same as first input` or `Custom`. This property only applies when the `Method` on page 4-1096 property is `Normalized LMS`. Setting this property also sets the `ConvolutionAccumulatorDataType` on page 4-1101 property to the same value. The default is `Same as first input`.

CustomEnergyAccumulatorDataType

Energy accumulator word and fraction lengths

Specify the energy accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property only applies when the `Method` on page 4-1096 property is `Normalized LMS` and the `EnergyAccumulatorDataType` on page 4-1100 property is `Custom`. The default is `numerictype([],32,20)`.

ConvolutionProductDataType

Convolution product word and fraction lengths

Specify the convolution product fixed-point data type as `Same as first input` or `Custom`. Setting this property also sets the `EnergyProductDataType` on page 4-1099, `StepSizeErrorProductDataType` on page 4-1101, `WeightsUpdateProductDataType` on page 4-1101 and `QuotientDataType` on page 4-1102 properties to the same value. The default is `Same as first input`.

CustomConvolutionProductDataType

Convolution product word and fraction lengths

Specify the convolution product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property only applies when the `ConvolutionProductDataType` on page 4-1100 property is `Custom`. The default is `numerictype([],32,20)`.

ConvolutionAccumulatorDataType

Convolution accumulator word and fraction lengths

Specify the convolution accumulator fixed-point data type as `Same as first input` or `Custom`. Setting this property also sets the `EnergyAccumulatorDataType` on page 4-1100 property to the same value. The default is `Same as first input`.

CustomConvolutionAccumulatorDataType

Convolution accumulator word and fraction lengths

Specify the convolution accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property only applies when the `ConvolutionAccumulatorDataType` on page 4-1101 property is `Custom`. The default is`numerictype([],32,20)`.

StepSizeErrorProductDataType

Step size error product word and fraction lengths

Specify the step size error product fixed-point data type as `Same as first input` or `Custom`. Setting this property also sets the `ConvolutionProductDataType` on page 4-1100, `EnergyProductDataType` on page 4-1099, `WeightsUpdateProductDataType` on page 4-1101, and `QuotientDataType` on page 4-1102 properties to the same value. The default is `Same as first input`.

CustomStepSizeErrorProductDataType

Step size error product word and fraction lengths

Specify the step size error product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property only applies when the `StepSizeErrorProductDataType` on page 4-1101 property is `Custom`. The default is `numerictype([],32,20)`.

WeightsUpdateProductDataType

Weight update product word and fraction lengths

Specify the weight update product fixed-point data type as `Same as first input` or`Custom`. Setting this property also sets the `ConvolutionProductDataType` on page 4-1100, EnergyProductDataType on page 4-1099, StepSizeErrorProductDataType on page 4-1101, and QuotientDataType on page 4-1102 properties to the same value. The default is Same as first input.

CustomWeightsUpdateProductDataType

Weight update product word and fraction lengths

Specify the weight update product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property only applies when the WeightsUpdateProductDataType on page 4-1101 property is Custom. The default is numerictype([],32,20).

QuotientDataType

Quotient word and fraction lengths

Specify the quotient fixed-point data type as one of Same as first input or Custom. This property only applies when the Method on page 4-1096 property is Normalized LMS. Setting this property also sets the ConvolutionProductDataType on page 4-1100, EnergyProductDataType on page 4-1099, StepSizeErrorProductDataType on page 4-1101, and WeightsUpdateProductDataType on page 4-1101 properties to the same value. The default is Same as first input.

CustomQuotientDataType

Quotient word and fraction lengths

Specify the quotient fixed-point type as a scaled numerictype object with a Signedness of Auto. This property only applies when the Method on page 4-1096 property is Normalized LMS and the QuotientDataType on page 4-1102 property is Custom. The default is numerictype([],32,20).

Methods

maxstep	Maximum step size for LMS adaptive filter convergence

msepred	Predicted mean-square error for LMS filter
msesim	Mean-squared error for LMS filter
reset	Reset filter states for LMS filter
step	Apply LMS adaptive filter to input

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

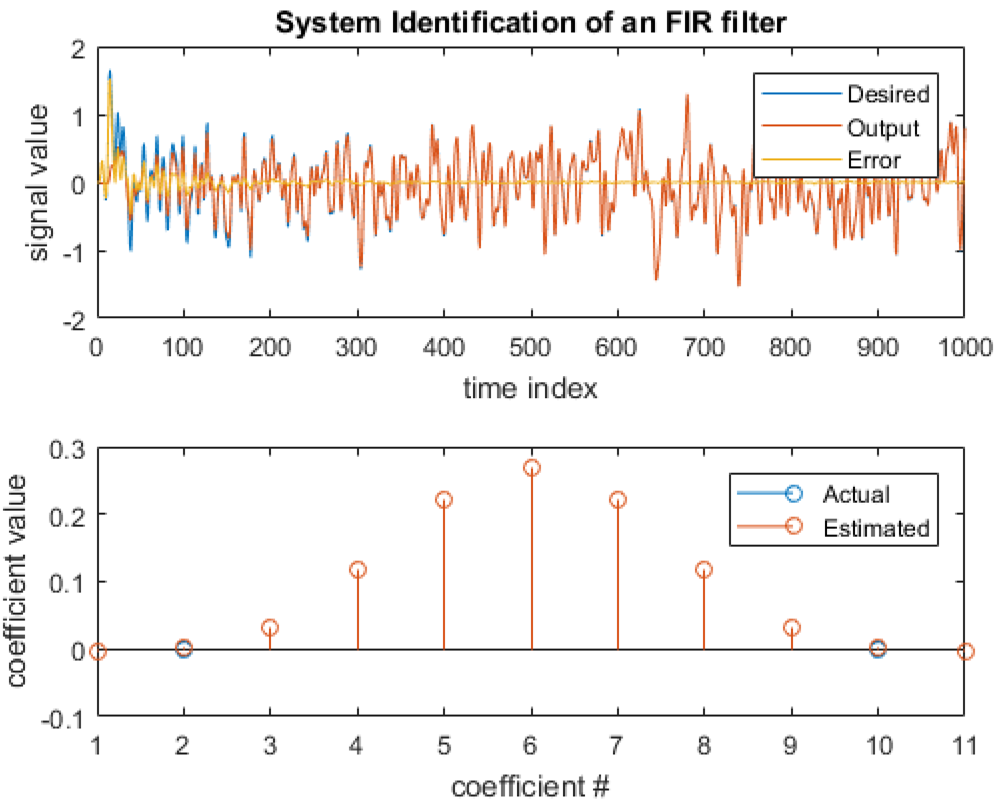
Examples

System Identification Using LMS Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
lms1 = dsp.LMSFilter(11,'StepSize',0.01);
filt = dsp.FIRFilter; % System to be identified
filt.Numerator = fir1(10,.25);
x = randn(1000,1); % input signal

d = filt(x) + 0.01*randn(1000,1); % desired signal
[y,e,w] = lms1(x,d);

subplot(2,1,1);
plot(1:1000, [d,y,e]);
title('System Identification of an FIR filter');
legend('Desired', 'Output', 'Error');
xlabel('time index');
ylabel('signal value');
subplot(2,1,2);
stem([filt.Numerator.',w]);
```

```
legend('Actual','Estimated');
xlabel('coefficient #');
ylabel('coefficient value');
```



Cancel Noise Using LMS Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

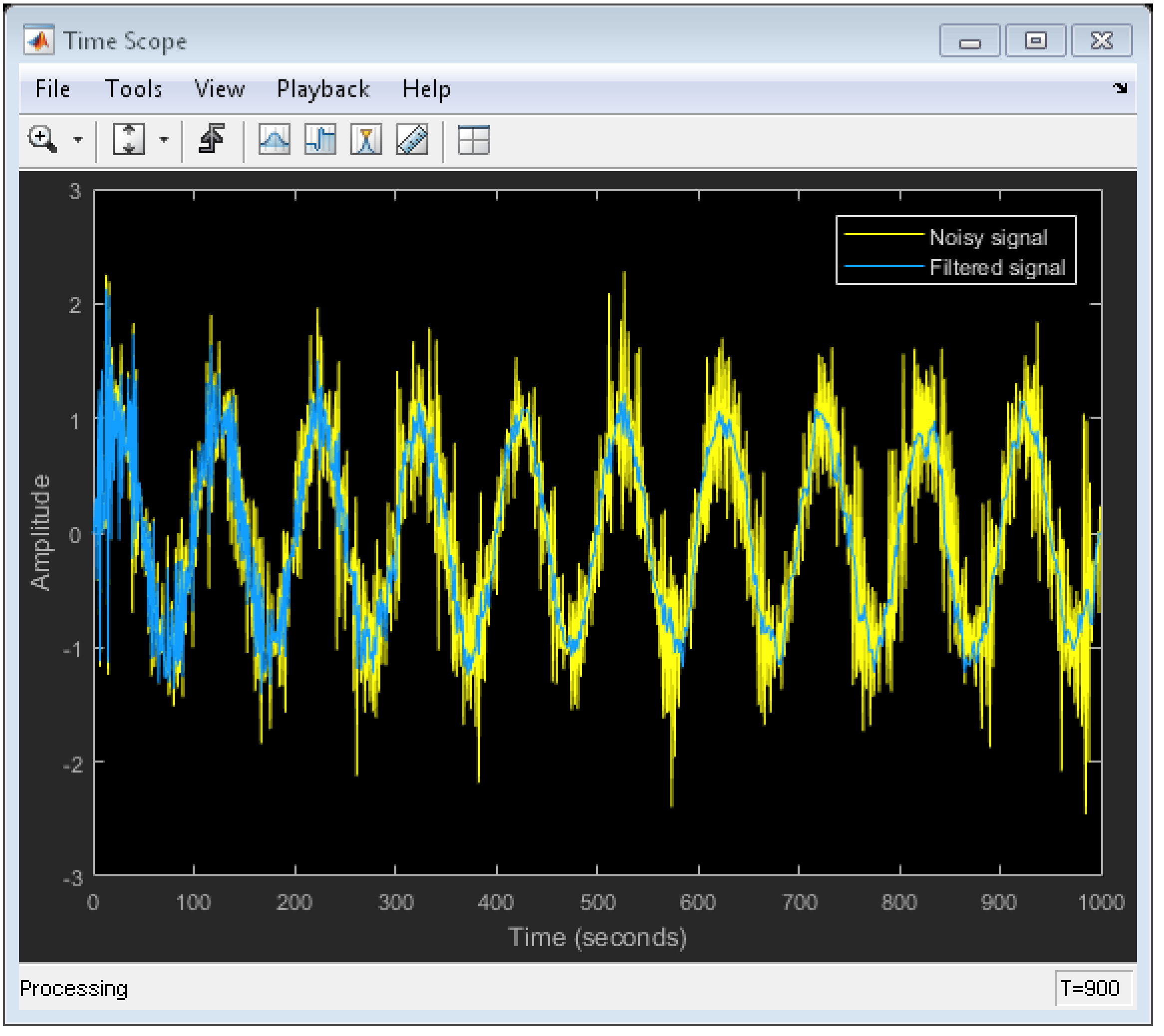
Initialize the LMS filter with a length of 11 and step size of 0.05.

```
FrameSize = 100; NIter = 10;
```

```
lmsfilt2 = dsp.LMSFilter('Length',11,'Method','Normalized LMS', ...
    'StepSize',0.05);
firfilt2 = dsp.FIRFilter('Numerator', fir1(10,[.5, .75]));
sinewave = dsp.SineWave('Frequency',0.01, ...
    'SampleRate',1,'SamplesPerFrame',FrameSize);
TS = dsp.TimeScope('TimeSpan',FrameSize*NIter,'TimeUnits','Seconds',...
    'YLimits',[-3 3],'BufferLength',2*FrameSize*NIter, ...
    'ShowLegend',true,'ChannelNames', ...
    {'Noisy signal', 'Filtered signal'});
```

Pass the noisy input signal into the LMS filter and view the filtered output in the time scope.

```
for k = 1:NIter
    x = randn(FrameSize,1);       % Input signal
    d = firfilt2(x) + sinewave(); % Noise + Signal
    [y,e,w] = lmsfilt2(x,d);
    TS([d,e]);          % Noisy = channel 1; Filtered = channel 2
end
```



Access the Full History of LMS Filter Weights

Note: This example runs only in R2017a or later. If you are using a release earlier than R2017a, the object does not output a full sample-by-sample history of filter weights. If you are using a release earlier than R2016b, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Initialize the `dsp.LMSFilter` System object™ and set the `WeightsOutput` property to `'All'`. This setting enables the LMS filter to output a matrix of weights with dimensions `[FrameLength Length]`, corresponding to the full sample-by-sample history of weights for all `FrameLength` samples of input values.

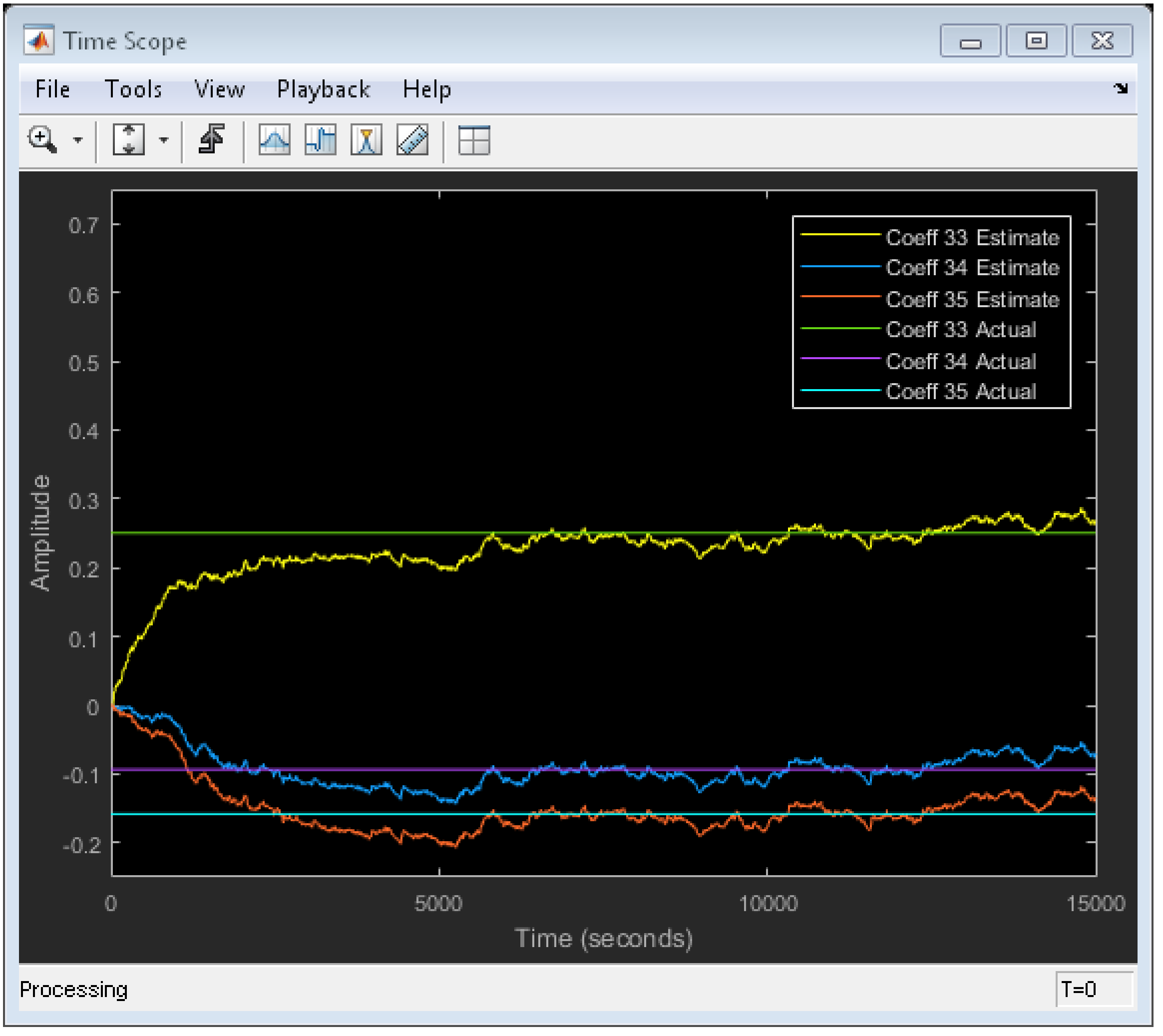
```
FrameSize = 15000;
lmsfilt3 = dsp.LMSFilter('Length',63,'Method','LMS', ...
    'StepSize',0.001,'LeakageFactor',0.99999, ...
    'WeightsOutput','All'); % full Weights history

w_actual = fir1(64,[0.5 0.75]);
firfilt3 = dsp.FIRFilter('Numerator',w_actual);
sinewave = dsp.SineWave('Frequency',0.01, ...
    'SampleRate',1,'SamplesPerFrame',FrameSize);

TS = dsp.TimeScope('TimeSpan',FrameSize,'TimeUnits','Seconds', ...
    'YLimits',[-0.25 0.75],'BufferLength',2*FrameSize, ...
    'ShowLegend',true,'ChannelNames', ...
    {'Coeff 33 Estimate','Coeff 34 Estimate','Coeff 35 Estimate', ...
    'Coeff 33 Actual','Coeff 34 Actual','Coeff 35 Actual'});
```

Run one frame and output the full adaptive weights history, `w`.

```
x = randn(FrameSize,1);       % Input signal
d = firfilt3(x) + sinewave(); % Noise + Signal
[~,~,w] = lmsfilt3(x,d);
```

Each row in `w` is a set of weights estimated for the respective input sample. Each column in `w` gives the complete history of a specific weight. Plot the actual weight and the entire history of the 33rd, 34th, and the 35th weight. In the plot, you can see that the estimated weight output eventually converges with the actual weight as the adaptive filter receives input samples and continues to adapt.

```
idxBeg = 33;
idxEnd = 35;
TS([w(:,idxBeg:idxEnd), repmat(w_actual(idxBeg:idxEnd),FrameSize,1)]);
```



Algorithms

This filter's algorithm is defined by the following equations.

$$y(n) = \mathbf{w}^T(n-1)\mathbf{u}(n)$$
$$e(n) = d(n) - y(n)$$
$$\mathbf{w}(n) = \alpha\mathbf{w}(n-1) + f(\mathbf{u}(n), e(n), \mu)$$

The various LMS adaptive filter algorithms available in this System object are defined as:

- LMS:

  $$f(\mathbf{u}(n), e(n), \mu) = \mu e(n)\mathbf{u}^*(n)$$

- Normalized LMS:

  $$f(\mathbf{u}(n), e(n), \mu) = \mu e(n)\frac{\mathbf{u}^*(n)}{\varepsilon + \mathbf{u}^H(n)\mathbf{u}(n)}$$

- Sign-Error LMS:

  $$f(\mathbf{u}(n), e(n), \mu) = \mu\, sign(e(n))\mathbf{u}^*(n)$$

- Sign-Data LMS:

  $$f(\mathbf{u}(n), e(n), \mu) = \mu e(n)\, sign(\mathbf{u}(n))$$

  where **u**(n) is real.
- Sign-Sign LMS:

  $$f(\mathbf{u}(n), e(n), \mu) = \mu\, sign(e(n))\, sign(\mathbf{u}(n))$$

  where **u**(n) is real.

The variables are as follows:

Variable	Description
n	The current time index
$\mathbf{u}(n)$	The vector of buffered input samples at step n

Variable	Description
u*(n)	The complex conjugate of the vector of buffered input samples at step n
w(n)	The vector of filter weight estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
μ	The adaptation step size
α	The leakage factor ($0 < \alpha \leq 1$)

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

System Objects
dsp.BlockLMSFilter | dsp.FIRFilter

Blocks
`LMS Filter`

Introduced in R2012a

maxstep

System object: dsp.LMSFilter
Package: dsp

Maximum step size for LMS adaptive filter convergence

Syntax

```
MUMAX = maxstep(lms,X)
[MUMAX,MUMAXMSE] = maxstep(lms,X)
```

Description

`MUMAX = maxstep(lms,X)` predicts a bound on the step size to provide convergence of the mean values of the adaptive filter coefficients. The columns of the matrix X contain individual input signal sequences. The signal set is assumed to have zero mean or nearly so.

`[MUMAX,MUMAXMSE] = maxstep(lms,X)` predicts a bound on the adaptive filter step size to provide convergence of the adaptive filter coefficients in mean square.

Examples

Compute the Maximum Step of the LMS Adaptive Filter

Analyze and simulate a 32-coefficient (31st-order) LMS adaptive filter object. To demonstrate the adaptation process, run 2000 iterations and 50 trials.

Specify [numiterations,numexamples] = size(x);

```
x = zeros(2000,50);
d = x;
lp = fdesign.lowpass('n,fc',31,0.5);
FIRFilter = design(lp,'window','SystemObject',true);
coef = FIRFilter.Numerator;
```

Create input and desired response signal matrices.

```
for k=1:size(x,2)
% Set the kth input to the filter.
  x(:,k) = filter(sqrt(0.75),[1 -0.5],sign(randn(size(x,1),1)));
  n = 0.1*randn(size(x,1),1);  % (k)th observation noise signal.
  d(:,k) = filter(coef,1,x(:,k))+n; % (k)th desired signal end.
end
mu = 0.1;      % LMS step size.
LMSFilter = dsp.LMSFilter('Length',32,'StepSize',mu);
```

Compute the maximum step of the filter.

```
[mumax,mumaxmse] = maxstep(LMSFilter,x)
```

```
mumax =

    0.0628


mumaxmse =

    0.0536
```

msepred

System object: dsp.LMSFilter
Package: dsp

Predicted mean-square error for LMS filter

Syntax

```
[MMSE,EMSE] = msepred(sysObj,X,D)
[MMSE,EMSE,MEANW,MSE,TRACEK] = msepred(sysObj,X,D)
[MMSE,EMSE,MEANW,MSE,TRACEK] = msepred(sysObj,X,D,M)
```

Description

`[MMSE,EMSE] = msepred(sysObj,X,D)` predicts the steady-state values at convergence of the minimum mean-squared error (`MMSE`) and the excess mean-squared error (`EMSE`) given the input and desired response signal sequences in `X` and `D` and the quantities in the adaptive filter `sysObj`.

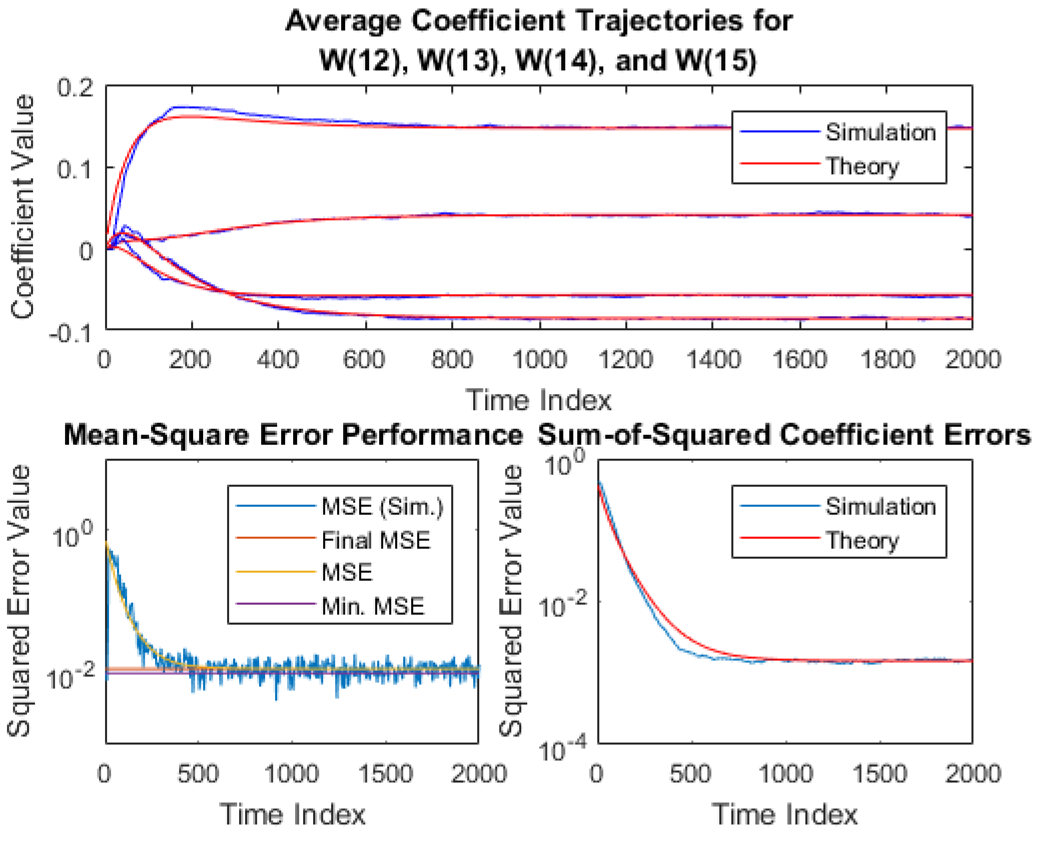
`[MMSE,EMSE,MEANW,MSE,TRACEK] = msepred(sysObj,X,D)` calculates three sequences corresponding to the analytical behavior of the adaptive filter defined by `sysObj`. `MEANW` is the sequence of coefficient vector means. The columns of this matrix contain predictions of the mean values of the adaptive filter coefficients at each time instant. The dimensions of `MEANW` are `(SIZE(X,1))` by `(H.length)`. `MSE` is the sequence of mean-square errors. This column vector contains predictions of the mean-square error of the adaptive filter at each time instant. The length of `MSE` is equal to `SIZE(X,1)`. `TRACEK` is a sequence of total coefficient error powers. This column vector contains predictions of the total coefficient error power of the adaptive filter at each time instant. The length of `TRACEK` is equal to `SIZE(X,1)`.

`[MMSE,EMSE,MEANW,MSE,TRACEK] = msepred(sysObj,X,D,M)` specifies an optional decimation factor for computing `MEANW`, `MSE`, and `TRACEK`. If `M > 1`, every M^{th} predicted value of each of these sequences is saved. If omitted, the value of `M` is the default, which is one.

Examples

Predict Mean Square Error for LMS Filter

```
x = zeros(2000,25); d = x;
firCoeff = fir1(31,0.5);
x = filter(sqrt(0.75),[1 -0.5],sign(randn(size(x))));
n = 0.1*randn(size(x));
d = filter(firCoeff,1,x)+n;
l = 32;
mu = 0.008;
m  = 5;

lms = dsp.LMSFilter('Length',l,'StepSize',mu);
[mmse,emse,meanW,mse,traceK] = msepred(lms,x,d,m);
[simmse,meanWsim,Wsim,traceKsim] = msesim(lms,x,d,m);
nn = m:m:size(x,1);
subplot(2,1,1);
plot(nn,meanWsim(:,12),'b',nn,meanW(:,12),'r',nn,...
meanWsim(:,13:15),'b',nn,meanW(:,13:15),'r');
PlotTitle ={'Average Coefficient Trajectories for';...
            'W(12), W(13), W(14), and W(15)'};
title(PlotTitle);
legend('Simulation','Theory');
xlabel('Time Index'); ylabel('Coefficient Value');
subplot(2,2,3);
semilogy(nn,simmse,[0 size(x,1)],[(emse+mmse)...
(emse+mmse)],nn,mse,[0 size(x,1)],[mmse mmse]);
title('Mean-Square Error Performance');
axis([0 size(x,1) 0.001 10]);
legend('MSE (Sim.)','Final MSE','MSE','Min. MSE');
xlabel('Time Index'); ylabel('Squared Error Value');
subplot(2,2,4);
semilogy(nn,traceKsim,nn,traceK,'r');
title('Sum-of-Squared Coefficient Errors'); axis([0 size(x,1)...
0.0001 1]);
legend('Simulation','Theory');
xlabel('Time Index'); ylabel('Squared Error Value');
```


Average Coefficient Trajectories for
W(12), W(13), W(14), and W(15)
Coefficient Value
Time Index
Simulation
Theory
Mean-Square Error Performance
Squared Error Value
Time Index
MSE (Sim.)
Final MSE
MSE
Min. MSE
Sum-of-Squared Coefficient Errors
Squared Error Value
Time Index
Simulation
Theory

msesim

System object: dsp.LMSFilter
Package: dsp

Mean-squared error for LMS filter

Syntax

```
MSE = msesim(sysObj,X,D)
[MSE,MEANW,W,TRACEK] = msesim(sysObj,X,D)
[...] = msesim(sysObj,X,D,M)
```

Description

`MSE = msesim(sysObj,X,D)` returns a sequence of mean-square errors. This column vector contains estimates of the mean-square error of the adaptive filter at each time instant. The length of `MSE` is equal to `SIZE(X,1)`. The columns of the matrix `X` contain individual input signal sequences, and the columns of the matrix `D` contain corresponding desired response signal sequences.

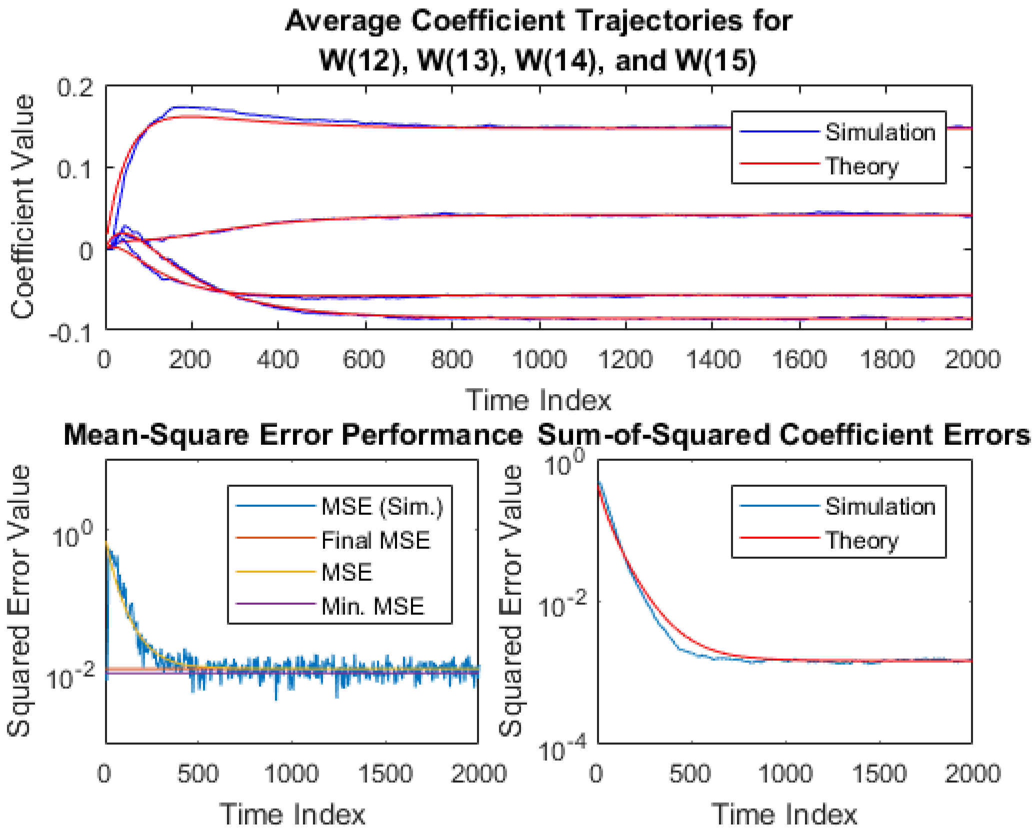
`[MSE,MEANW,W,TRACEK] = msesim(sysObj,X,D)` calculates three parameters corresponding to the simulated behavior of the adaptive filter defined by `sysObj`. `MEANW` is a sequence of coefficient vector means. The columns of this matrix contain estimates of the mean values of the adaptive filter coefficients at each time instant. The dimensions of `MEANW` are `(SIZE(X,1))` by `(sysObj.length)`. `W` is an estimate of the final values of the adaptive filter coefficients for the algorithm corresponding to `sysObj`. `TRACEK` is a sequence of total coefficient error powers. This column vector contains estimates of the total coefficient error power of the adaptive filter at each time instant. The length of `TRACEK` is equal to `SIZE(X,1)`.

`[...] = msesim(sysObj,X,D,M)` specifies an optional decimation factor for computing `MSE`, `MEANW`, and `TRACEK`. If $M > 1$, every M^{th} predicted value of each of these sequences is saved. If omitted, the value of `M` is the default, which is 1.

Examples

Predict Mean Square Error for LMS Filter

```
x = zeros(2000,25); d = x;
firCoeff = fir1(31,0.5);
x = filter(sqrt(0.75),[1 -0.5],sign(randn(size(x))));
n = 0.1*randn(size(x));
d = filter(firCoeff,1,x)+n;
l = 32;
mu = 0.008;
m  = 5;

lms = dsp.LMSFilter('Length',l,'StepSize',mu);
[mmse,emse,meanW,mse,traceK] = msepred(lms,x,d,m);
[simmse,meanWsim,Wsim,traceKsim] = msesim(lms,x,d,m);
nn = m:m:size(x,1);
subplot(2,1,1);
plot(nn,meanWsim(:,12),'b',nn,meanW(:,12),'r',nn,...
meanWsim(:,13:15),'b',nn,meanW(:,13:15),'r');
PlotTitle ={'Average Coefficient Trajectories for';...
            'W(12), W(13), W(14), and W(15)'};
title(PlotTitle);
legend('Simulation','Theory');
xlabel('Time Index'); ylabel('Coefficient Value');
subplot(2,2,3);
semilogy(nn,simmse,[0 size(x,1)],[(emse+mmse)...
(emse+mmse)],nn,mse,[0 size(x,1)],[mmse mmse]);
title('Mean-Square Error Performance');
axis([0 size(x,1) 0.001 10]);
legend('MSE (Sim.)','Final MSE','MSE','Min. MSE');
xlabel('Time Index'); ylabel('Squared Error Value');
subplot(2,2,4);
semilogy(nn,traceKsim,nn,traceK,'r');
title('Sum-of-Squared Coefficient Errors'); axis([0 size(x,1)...
0.0001 1]);
legend('Simulation','Theory');
xlabel('Time Index'); ylabel('Squared Error Value');
```


Average Coefficient Trajectories for
W(12), W(13), W(14), and W(15)
Coefficient Value
Simulation
Theory
0.2
0.1
0
-0.1
0
200
400
600
800
1000
1200
1400
1600
1800
2000
Time Index
Mean-Square Error Performance
Squared Error Value
MSE (Sim.)
Final MSE
MSE
Min. MSE
10^0
10^{-2}
0
500
1000
1500
2000
Time Index
Sum-of-Squared Coefficient Errors
Squared Error Value
Simulation
Theory
10^0
10^{-2}
10^{-4}
0
500
1000
1500
2000
Time Index

reset

System object: dsp.LMSFilter
Package: dsp

Reset filter states for LMS filter

Syntax

```
reset(lms)
```

Description

`reset(lms)` resets the filter states of the LMS FIR filter, `lms`, to their initial values specified in the `InitialConditions` property. The initial filter state values correspond to the initial conditions for difference equation defining the adaptive filter. After the `step` method applies the LMS filter to nonzero input data, the states may be different. Invoking the `step` method again without first invoking the`reset` method may produce different outputs for an identical input.

Reset an LMS Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
lms1 = dsp.LMSFilter(11, 'StepSize', 0.01);
filt = dsp.FIRFilter; % System to be identified
filt.Numerator = fir1(10, .25);
x = randn(1000,1); % input signal
d = filt(x) + 0.01*randn(1000,1); % desired signal
[y,e,w] = lms1(x, d);
```

Call LMS algorithm again without resetting filter states.

```
[y1,e1,w1] = lms1(x,d);
isequal(y,y1)
```

```
ans =

  logical

   0
```

Now reset the filter states to zero.

```
reset(lms1)
```

Call the LMS algorithm.

```
[y2,e2,w2] = lms1(x,d);
isequal(y,y2)
```

```
ans =

  logical

   1
```

step

System object: dsp.LMSFilter
Package: dsp

Apply LMS adaptive filter to input

Syntax

```
[Y,ERR,WTS] = step(lms,X,D)
[Y, ERR] = step(lms,X,D)
[...] = step(lms,X,D,MU)
[...] = step(lms,X,D,A)
[...] = step(lms,X,D,R)
[Y,ERR,WTS] = step(lms,X,D,MU,A,R)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[Y,ERR,WTS] = step(lms,X,D)` applies the LMS filter object, `lms` to the input `X` , using `D` as the desired signal. This approach returns the filtered output in `Y` , the filter error in `ERR` , and the estimated filter weights in `WTS` . The LMS filter estimates the filter weights needed to minimize the mean square error between the output and the desired signal.

`[Y, ERR] = step(lms,X,D)` filters the input `X` , using `D` as the desired signal. This approach returns the filtered output in `Y` and the filter error in `ERR` when the `WeightsOutputPort` property is `false`.

`[...] = step(lms,X,D,MU)` uses `MU` as the step size, when the `StepSizeSource` property is `'Input port'`.

[...] = step(lms,X,D,A) uses A as the adaptation control, when the AdaptInputPort property is true. When A is nonzero, the LMS filter continuously updates the filter weights. When A is zero, the filter weights remain constant.

[...] = step(lms,X,D,R) uses R as a reset signal when the WeightsResetInputPort property is true. The WeightsResetCondition property can be used to set the reset trigger condition. If a reset event occurs, the LMS filter resets the filter weights to their initial values.

[Y,ERR,WTS] = step(lms,X,D,MU,A,R) filters the input X using D as the desired signal, MU as the step size, A as the adaptation control, and R as the reset signal. This approach returns the filtered output in Y , the filter error in ERR , and the adapted filter weights in WTS.

Note: obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

dsp.LogicAnalyzer System object

Package: dsp

Visualize, measure, and analyze transitions and states over time

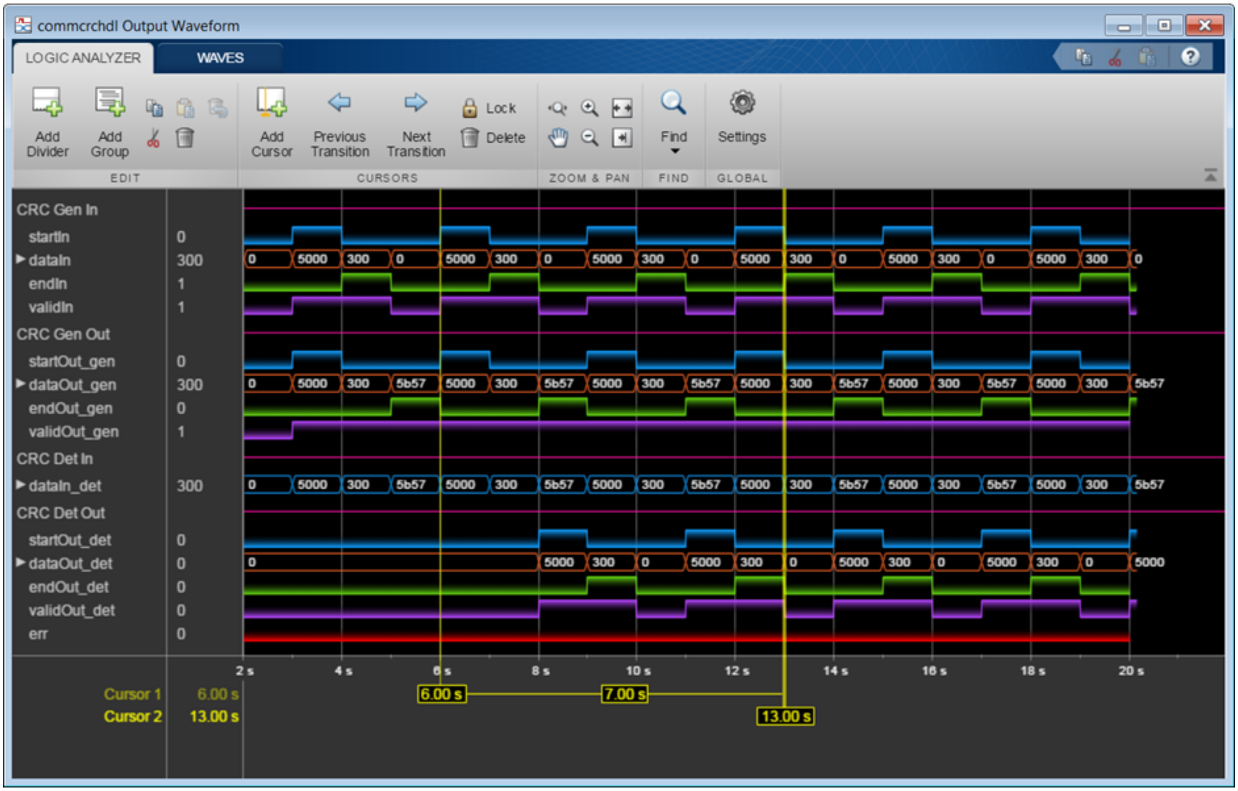
Description

The `LogicAnalyzer` System object displays the transitions in time-domain signals.

To display the transitions of signals in the Logic Analyzer:

1. Define and set up your Logic Analyzer. See "Construction" on page 4-1124.
2. Call step to display the transitions of the signals in the Logic Analyzer figure.

Use the MATLAB `clear` function to close the Logic Analyzer figure window and clear its associated data. Use the hide method to hide the Logic Analyzer window and the show method to make it visible.



Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`scope = dsp.LogicAnalyzer` creates a Logic Analyzer System object, `scope`.

`scope = dsp.LogicAnalyzer(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Properties

If a property is listed as tunable, then you can change its value even when the object is locked.

`BackgroundColor` — Background color for display
'Black' (default) | 'White'

Background color of the display, specified as 'Black' or 'White'.

Tunable: Yes

Data Types: char

`DisplayChannelColor` — Color for channels in the display
[0.0588 1 1] (default) | RGB triplet

Color for channels in the display, specified as an RGB triplet.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7].

Tunable: Yes

Data Types: double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

`DisplayChannelFontSize` — Font size for channels in the display (points)
10 (default) | nonnegative scalar integer

Font size for channels in the display, in points, specified as a nonnegative scalar integer.

Tunable: Yes

Data Types: double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

`DisplayChannelFormat` — Format for channels in the display
'Automatic' (default) | 'Analog' | 'Digital'

Format for channels in the display, specified as one of the following:

- 'Automatic' — Displays floating-point signals in Analog format and integer and fixed-point signals in Digital format. Boolean signals are displayed as zero or one.

- `'Analog'` — Shows values as an analog plot.
- `'Digital'` — Shows values as digital transitions.

Tunable: Yes

Data Types: `char`

`DisplayChannelHeight` — Channel height in the display (pixels)

`12` (default)

Channel height in the display, in pixels, specified as a positive real scalar in the range 8 to 200.

Tunable: Yes

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`DisplayChannelRadix` — Base of the enumeration used to display values

`'Hexadecimal'` (default) | `'Binary'` | `'Octal'` | `'Signed decimal'` | `'Unsigned decimal'`

Base of the enumeration used to display values, specified as one of the following:

- `'Hexadecimal'` — Displays values as symbols from zero to nine and A to F
- `'Binary'` — Displays values as zeros and ones
- `'Octal'` — Displays values as numbers from zero to seven
- `'Signed decimal'` — Displays the signed, stored integer value
- `'Unsigned decimal'` — Displays the stored integer value

This property applies only to fixed-point (`fi`) values.

Tunable: Yes

Data Types: `char`

`DisplayChannelSpacing` — Spacing between channels in display (pixels)

4 (default) | positive scalar integer

Spacing between channels in the display, in pixels, specified as a positive scalar integer.

Tunable: Yes

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`Name` — Caption to display on scope window

`'Logic Analyzer'` (default) | character vector

Caption to display on the scope window, specified as a character vector.

Tunable: Yes

Data Types: `char`

`NumInputPorts` — Number of input signals

`1` | positive scalar integer

Number of input signals, specified as a positive scalar integer. You must call the `step` method with the same number of inputs as the value of this property.

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`Position` — Scope window position (pixels)

`[left bottom width height]` vector

Position of the scope window on your screen, in pixels, specified as a `[left bottom width height]` vector. The default position depends on your screen resolution. By default, the scope window appears in the center of your screen, with a width of `800` pixels and height of `600` pixels.

Tunable: Yes

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`SampleTime` — Sample time of inputs (seconds)

`1` (default) | finite numeric scalar

Sample time of inputs in seconds, specified as a finite numeric scalar. The same sample time is used for all inputs.

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`TimeDisplayOffset` — Time display offset (seconds)

`0` (default) | nonnegative scalar

Time display offset in seconds, specified as a nonnegative scalar.

Tunable: Yes

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`TimeSpan` — Time span (seconds)
`10` | positive scalar

Time span in seconds, specified as a positive scalar. The *x*-axis limits are calculated as follows:

- Minimum *x*-axis limit = min(`TimeDisplayOffset`)
- Maximum *x*-axis limit = max(`TimeDisplayOffset`) + `TimeSpan`

`TimeDisplayOffset` and `TimeSpan` are the values of their respective properties.

Tunable: Yes

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

Methods

addCursor	Add cursor to display
addDivider	Add divider to display
addWave	Add wave corresponding to specified input
deleteCursor	Delete specified cursor
deleteDisplayChannel	Delete specified display channel
getCursorInfo	Return settings for specified cursor
getCursorTags	Return all cursor tags
getDisplayChannelInfo	Return settings for specified display channel
getDisplayChannelTags	Return all display channel tags
hide	Hide scope window
modifyCursor	Modify properties of specified cursor

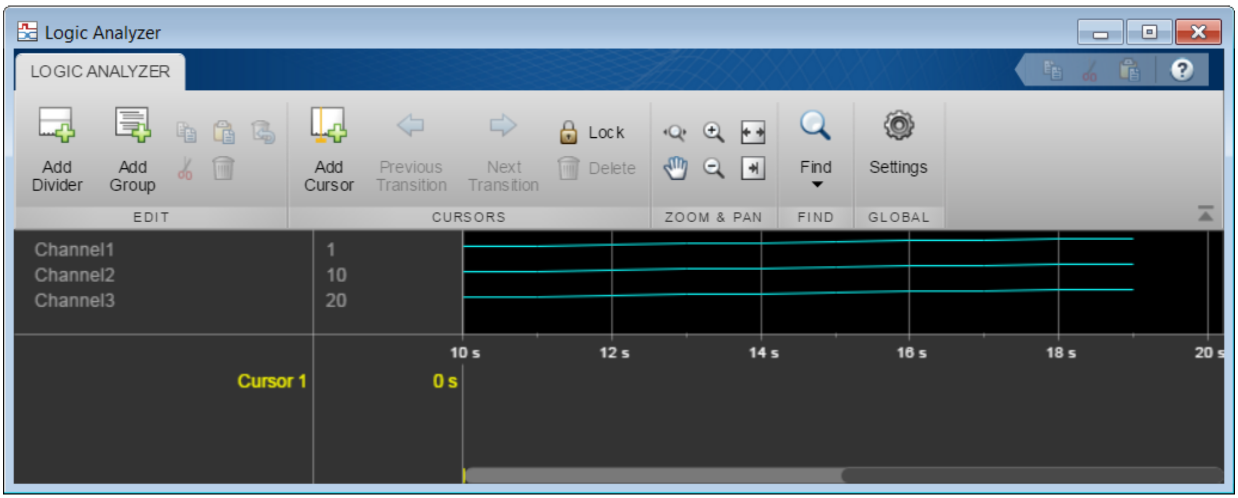
modifyDisplayChannel	Modify properties of specified display channel
moveDisplayChannel	Move position of specified display channel
reset	Reset internal states of scope object
show	Make scope window visible
step	Display signal in scope figure

Examples

Display Simple Ramp Signals

Create a `dsp.LogicAnalyzer` object. Call the scope in a loop to display the signals.

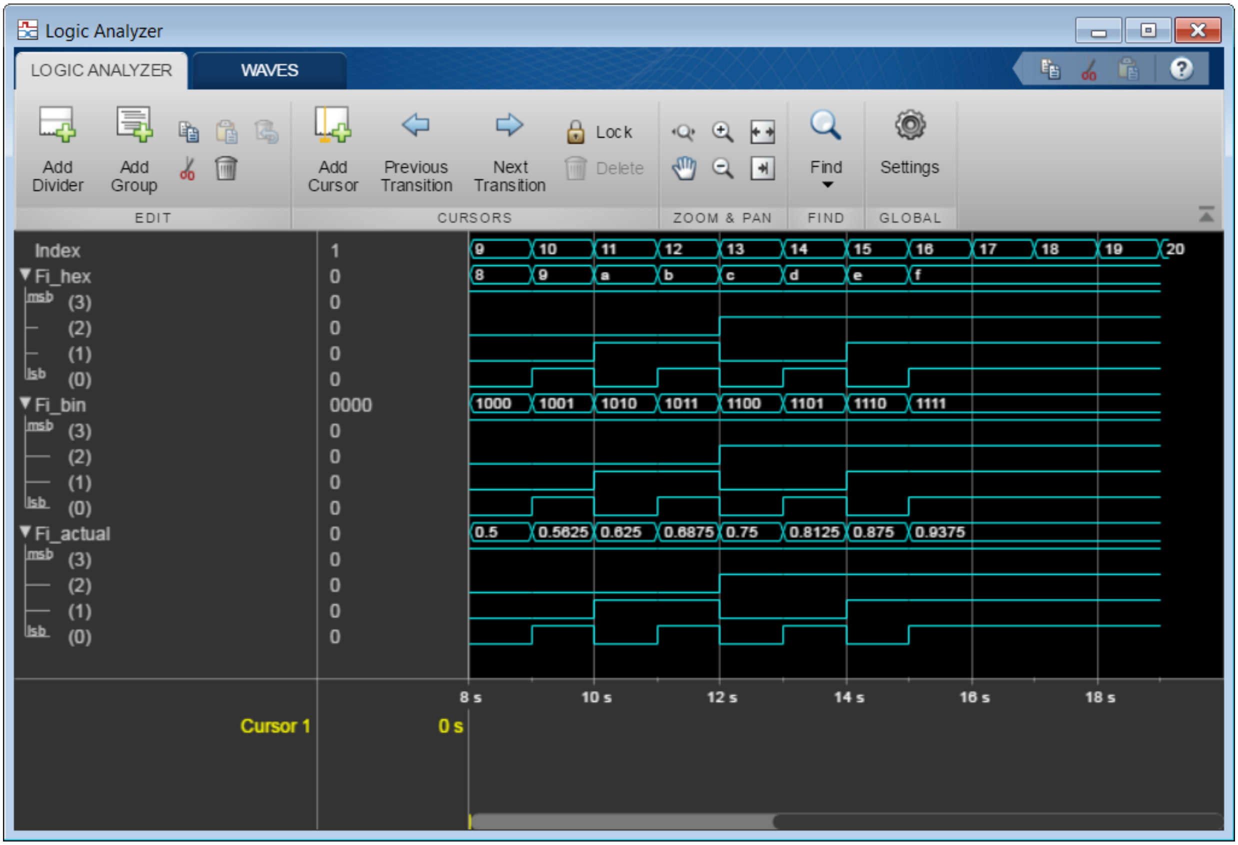
```
scope = dsp.LogicAnalyzer('NumInputPorts',3);
for ii = 1:20
    scope(ii,10*ii,20*ii);
end
```



Display Fixed-Point Signals

Create a `dsp.LogicAnalyzer` object with four channels. Call `modifyDisplayChannel` to set the radix of each of the channels. Run the scope in a loop to display the waves.

```
scope = dsp.LogicAnalyzer('NumInputPorts',4,'DisplayChannelFormat','Digital');
scope.TimeSpan = 12;

modifyDisplayChannel(scope,1,'Name','Index','Radix','Unsigned decimal');
modifyDisplayChannel(scope,2,'Name','Fi_hex','Radix','Hexadecimal');
modifyDisplayChannel(scope,3,'Name','Fi_bin','Radix','Binary');
modifyDisplayChannel(scope,4,'Name','Fi_actual','Radix','Signed decimal');

for ii = 1:20
    fival = fi((ii-1)/16,0,4,4);
    scope(ii,fival,fival,fival);
end
```



Display Vector, Complex, and Enumerated Signals

Define a `WeekDaysInt` class to hold an enumerated list of weekday values. Create and save the following class definition file.

```
classdef WeekDaysInt < int32
  enumeration
    Monday(1), Tuesday(2), Wednesday(3), Thursday(4), Friday(5)
  end
end
```

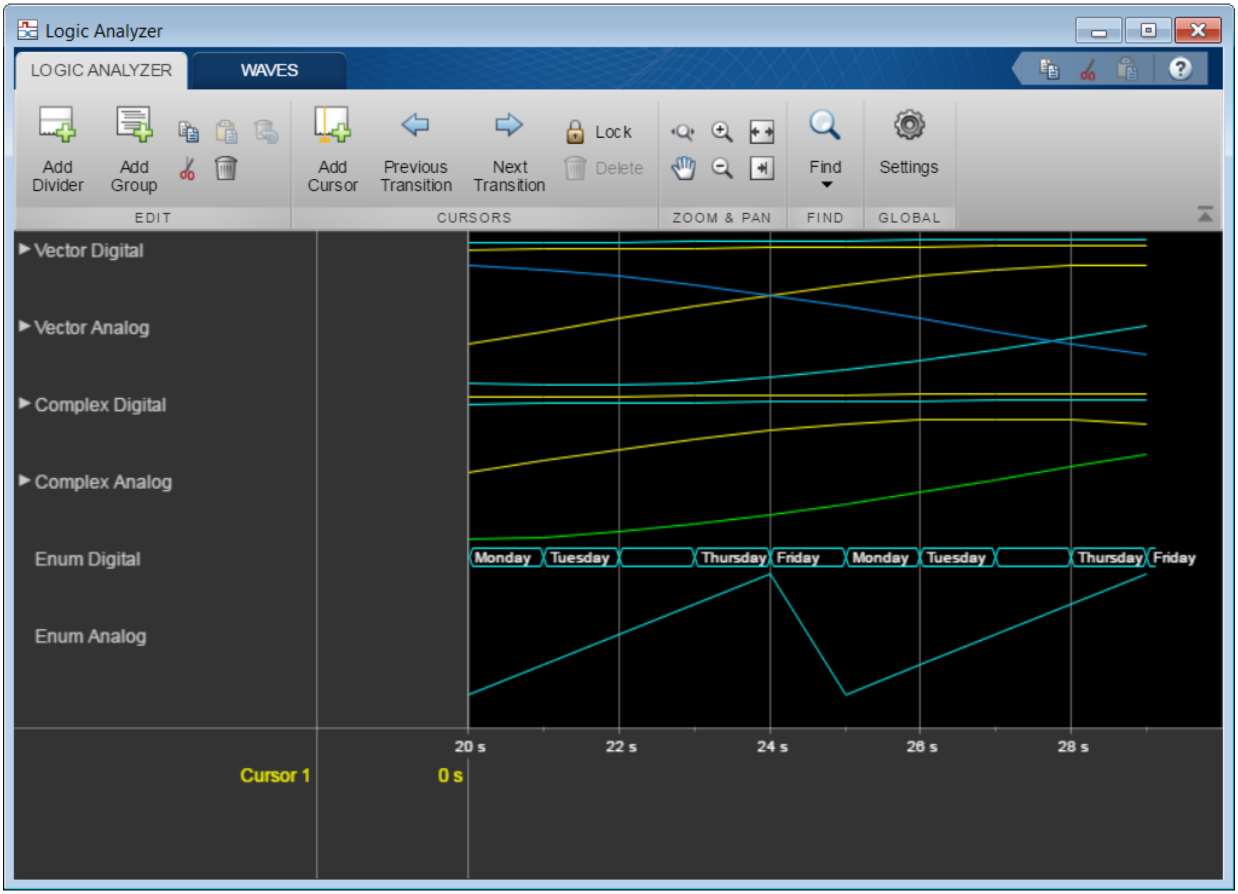
Create and configure the vector, complex, and enumerated data signals.

```
s{1} = dsp.LogicAnalyzerWave('InputChannel',1,...
```

```
     'Name','Vector Digital');
s{2} = dsp.LogicAnalyzerWave('InputChannel',2,...
     'Name','Vector Analog', 'Format','Analog',...
     'Height',80);
s{3} = dsp.LogicAnalyzerWave('InputChannel',3,...
     'Name','Complex Digital');
s{4} = dsp.LogicAnalyzerWave('InputChannel',4,...
     'Name','Complex Analog', 'Format','Analog',...
     'Height',80,'Color','Green');
s{5} = dsp.LogicAnalyzerWave('InputChannel',5,...
     'Name','Enum Digital');
s{6} = dsp.LogicAnalyzerWave('InputChannel',6,...
     'Name','Enum Analog', 'Format','Analog',...
     'Height',80);
```

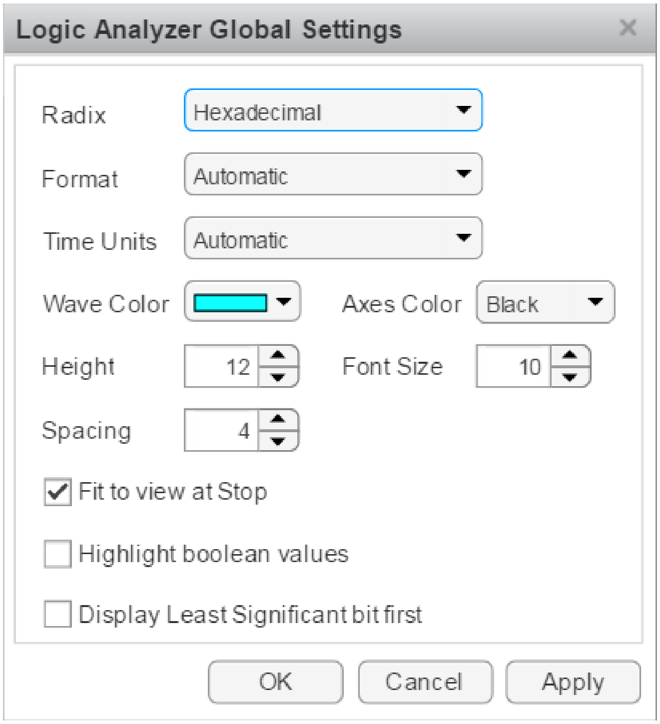
Create a `dsp.LogicAnalyzer` object. Call the object in a loop to display the signals.

```
scope = dsp.LogicAnalyzer('DisplayChannels',s,...
     'NumInputPorts',numel(s));

t = 30;
for c = 1:t
    sinValVec       = sin(c/t*2*pi);
    cosValVec       = cos(c/t*2*pi);
    cosValVecOffset = cos((c+10)/t*2*pi);
    sinValReal      = sin((c+2)/t*2*pi);
    cosValImag      = cos((c+2)/t*2*pi);

    % Create a weekday enumerated value by wrapping the index
    day = WeekDaysInt(1+mod(c-1,5));

    scope(...
          [c (c-(t/2))],...                          % digital vector
          [sinValVec cosValVec cosValVecOffset],... % analog vector
          complex((c-(t/2)),c),...                  % digital complex
          complex(sinValReal, cosValImag),...       % analog complex
          day,...                                    % digital enum
          day...                                     % analog enum
          )
end
```



Configure Visualization of Logic Analyzer

The Logic Analyzer Global Settings dialog box controls the default visual configuration of the Logic Analyzer display. From the Logic Analyzer toolstrip, select **Settings** to open this dialog box. Any settings made on individual signals supersede the global setting.



Set the display **Radix** of your signals as one of the following:

- `Hexadecimal` — Displays values as symbols from zero to nine and A to F
- `Octal` — Displays values as numbers from zero to seven
- `Binary` — Displays values as zeros and ones
- `Signed decimal` — Displays the signed, stored integer value
- `Unsigned decimal` — Displays the stored integer value

Set the display **Format** as one of the following:

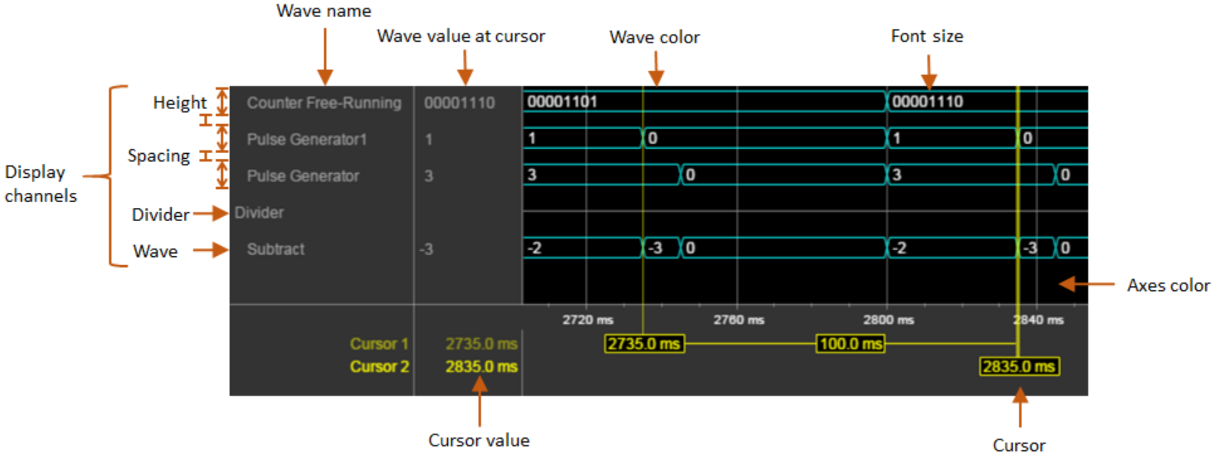
- `Automatic` — Displays floating point signals in `Analog` format and integer and fixed-point signals in `Digital` format. Boolean signals are displayed as zero or one.
- `Analog` — Displays values as an analog plot
- `Digital` — Displays values as digital transitions

Set the display **Time Units** to one of the following:

- `Automatic` — Uses a time scale appropriate to the time range shown in the current plot
- `seconds`
- `milliseconds`
- `microseconds`
- `nanoseconds`
- `picoseconds`
- `femtoseconds`

You can expand fixed-point and integer signals and view individual bits. The **Display Least Significant bit first** option enables you to reverse the order of the displayed bits.

Inspect the graphic for an explanation of the global settings: **Wave Color**, **Axes Color**, **Height**, **Font Size**, and **Spacing**. **Font Size** applies only to the text within the axes.



See Also

See Also

dsp.TimeScope | dsp.SpectrumAnalyzer | dsp.ArrayPlot | Logic Analyzer

Introduced in R2013a

addCursor

System object: dsp.LogicAnalyzer
Package: dsp

Add cursor to display

Syntax

```
cursorTag = addCursor(scope)
cursorTag = addCursor(scope,'Name',Value)
```

Description

`cursorTag = addCursor(scope)` adds in a cursor to the display. A tag value is returned, which can be used to modify and delete the cursor.

`cursorTag = addCursor(scope,'Name',Value)` adds in a cursor to the display with each specified property *`Name`* set to the specified value. You can specify *`Name`*–*`Value`* arguments in any order.

Input Arguments

`scope` — The Logic Analyzer object to which you want to add a cursor
`dsp.LogicAnalyzer` object

Example: `'addCursor(scope)'` adds a cursor with the default characteristics.

Example: `'scope.addCursor()'` adds a cursor with the default characteristics.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Location',2,'Color','Blue'` specifies that a cursor should be moved to the 2-second mark and colored blue.

`'Color'` — Color of the cursor

`'Yellow'` (default) | character vector | three element vector

Color of the cursor, specified as an `[R G B]` value or one of the following:

- `'Black'`
- `'Blue'`
- `'Cyan'`
- `'Green'`
- `'Magenta'`
- `'White'`
- `'Yellow'`

Example: `'Color','Blue'`

Example: `'Color',[0,0,1]`

Data Types: `char` | `double`

`'Location'` — Location of the cursor

`0` (default) | numeric scalar

Specify as a numeric scalar value, in seconds, the cursor location.

Example: `'Location',1`

Data Types: `double`

`'Locked'` — Locked status of the cursor

`false` (default) | `true`

Locked status of the cursor, specified as `false` or `true`.

- `true` — the cursor's location cannot be changed. Logic Analyzer denotes this by assigning a default color of red.
- `false` — the cursor's location can be changed. Logic Analyzer denotes this by assigning a default color of yellow.

Example: `'Locked',true`

addDivider

System object: dsp.LogicAnalyzer
Package: dsp

Add divider to display

Syntax

```
dividerTag = addDivider(scope)
dividerTag = addDivider(scope,'Name',Value)
```

Description

`dividerTag = addDivider(scope)` adds a divider to the display. A tag value is returned, which can be used to modify and delete the divider.

`dividerTag = addDivider(scope,'Name',Value)` adds in a divider to the display with each specified property *`Name`* set to the specified value. You can specify *`Name`*–*`Value`* arguments in any order.

Input Arguments

`scope` — The Logic Analyzer object to which you want to add a divider

dsp.LogicAnalyzer object

Example: `addDivider(scope)` adds a divider with the default characteristics.

Example: `scope.addDivider()` adds a divider with the default characteristics.

Divider Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DisplayChannel',2,'Name','MyDivider'` specifies that a divider should be added to display channel 2 and named “MyDivider”.

`'DisplayChannel'` — Channel on the display that shows this divider
`NumInputPorts` (default) | scalar numeric value in the range (1,`NumInputPorts`)

Specify as a scalar numeric value the display channel that shows this divider. By default, the divider is added to the end of the display.

Example: `'DisplayChannel',2`

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`'Height'` — Height of the divider
`0` (default) | scalar integer

Specify, in pixels, the height of the divider as a scalar integer in the range 8 to 200. If you choose 0, the value of the `DisplayChannelHeight` property in the Logic Analyzer is used.

Example: `'Height',2`

Data Types: `double`

`'Name'` — The name or label for the divider
`''` (default) | character vector

Specify as a character vector the name that you would like to set for the new divider.

Example: `'Name','MyDivider'`

Data Types: `char`

addWave

System object: dsp.LogicAnalyzer
Package: dsp

Add wave corresponding to specified input

Syntax

```
waveTag = addWave(scope)
waveTag = addWave(scope,'Name',Value)
```

Description

`waveTag = addWave(scope)` adds in a wave corresponding to the specified input channel. A tag value is returned back, which can be used to modify and delete the wave.

`waveTag = addWave(scope,'Name',Value)` adds in a wave with each specified property *`Name`* set to the specified value. You can specify *`Name`*–*`Value`* arguments in any order.

Input Arguments

`scope` — The Logic Analyzer object to which you want to add a wave
dsp.LogicAnalyzer object

Example: `'addWave(scope)'` adds a wave with the default characteristics.

Example: `'scope.addWave()'` adds a wave with the default characteristics.

Wave Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'InputChannel',2,'Color','Blue'` specifies that a wave should be added to input channel 1 and colored blue.

`'Color'` — Color of the wave

`'Default'` (default) | character vector

Color of the wave, specified as an `[R G B]` value or one of the following:

- `'Black'`
- `'Blue'`
- `'Cyan'`
- `'Default'`
- `'Green'`
- `'Magenta'`
- `'Red'`
- `'White'`
- `'Yellow'`

When you choose `'Default'`, the value of the `DisplayChannelColor` property in the Logic Analyzer is used.

Example: `'Color','Blue'`

Example: `'Color',[0,0,1]`

Data Types: `char` | `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`'DisplayChannel'` — Channel on the display that shows this wave

`NumInputPorts` (default) | scalar numeric value in the range (1,`NumInputPorts`)

Specify as a scalar numeric value the display channel that shows this wave. By default, the wave is added to the end of the display.

Example: `'DisplayChannel',2`

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`'FontSize'` — Font size for values in the wave

`0` (default) | scalar nonnegative integer

Specify as a scalar nonnegative integer the font size in points. When you choose 0, the value of the `DisplayChannelFontSize` property in the Logic Analyzer is used.

Example: `'FontSize',8`

Data Types: `double`

`'Format'` — Display format for the wave

`'Default'` (default) | character vector

`'Format'` must be one of the following.

- `'Analog'`
- `'Default'`
- `'Digital'`

When you choose `'Default'`, the value of the `DisplayChannelFormat` property in the Logic Analyzer is used.

Example: `'Format','Digital'`

Data Types: `char`

`'Height'` — Height of the wave

`0` (default) | scalar integer

Specify as a scalar integer the height of the wave in the display in units of 16 pixels. When you choose 0, the value of the `DisplayChannelHeight` property in the Logic Analyzer is used.

Example: `'Height',2`

Data Types: `double`

`'InputChannel'` — Input channel that corresponds to this wave

`1` (default) | scalar integer in the range (1,`NumInputPorts`)

This property specifies the input channel whose data is used for this wave. By default, it will connect the first input to this wave.

Example: `'InputChannel',2`

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`'Name'` — The name or label for the wave
`''` (default) | character vector

Specify as a character vector the name that you would like to set for the new wave.

Example: `'Name','MyWave'`

Data Types: `char`

`'Radix'` — Radix for the wave
`'Default'` (default) | character vector

`'Radix'` must be one of the following.

- `'Binary'`
- `'Default'`
- `'Hexadecimal'`
- `'Octal'`
- `'Signed decimal'`
- `'Unsigned decimal'`

When the input signals are of class double, single, or logical, you should not set this property. When you choose `'Default'`, the value of the `DisplayChannelRadix` property in the Logic Analyzer is used.

Example: `'Radix','Hexadecimal'`

Data Types: `char`

deleteCursor

System object: dsp.LogicAnalyzer
Package: dsp

Delete specified cursor

Syntax

```
deleteCursor(scope,tag)
deleteCursor(scope,'CursorTag',tag)
```

Description

`deleteCursor(scope,tag)` deletes the cursor specified by the input tag.

`deleteCursor(scope,'CursorTag',tag)` deletes the cursor specified by the input tag.

Input Arguments

`scope` — The Logic Analyzer object from which you want to delete a cursor

`dsp.LogicAnalyzer` object handle

The Logic Analyzer object from which you want to delete a cursor, specified as a handle to the `dsp.LogicAnalyzer` object.

`tag` — The tag identifying which cursor to delete

character vector, randomly assigned

The tag identifying which cursor to delete, specified as a randomly assigned character vector.

Example: `'deleteCursor(scope,tag)'` deletes a cursor from Logic Analyzer.

Example: `'scope.deleteCursor(tag)'` deletes a cursor from Logic Analyzer.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'CursorTag','C6'` specifies to delete the divider with tag `'C6'`.

`'CursorTag'` — The tag identifying which cursor to delete

character vector, randomly assigned

The tag identifying which cursor to delete, specified as a randomly assigned character vector.

Example: `'CursorTag','C6'`

Data Types: `char`

deleteDisplayChannel

System object: dsp.LogicAnalyzer
Package: dsp

Delete specified display channel

Syntax

```
deleteDisplayChannel(scope,tag)
deleteDisplayChannel(scope,'DisplayChannelTag',tag)
```

Description

`deleteDisplayChannel(scope,tag)` deletes the display channel, either a wave or a divider, specified by the input tag.

`deleteDisplayChannel(scope,'DisplayChannelTag',tag)` deletes the display channel, either a wave or a divider, specified by the input tag.

Input Arguments

`scope` — The Logic Analyzer object from which you want to delete a display channel

`dsp.LogicAnalyzer` object

The Logic Analyzer object from which you want to delete a display channel, specified as a handle to the `dsp.LogicAnalyzer` object.

`tag` — The tag identifying which display channel to delete

character vector

The tag identifying which display channel to delete, specified as a character vector.

Example: `'deleteDisplayChannel(scope,tag)'` deletes a display channel from Logic Analyzer.

Example: `'scope.deleteDisplayChannel(tag)'` deletes a display channel from Logic Analyzer.

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DisplayChannelTag','C5'` specifies to delete the divider with tag `'C5'`.

`'DisplayChannelTag'` — The tag identifying which display channel to delete
character vector

The tag identifying which display channel to delete, specified as a character vector.

Example: `'DisplayChannelTag','C5'`

Data Types: `char`

getCursorInfo

System object: dsp.LogicAnalyzer
Package: dsp

Return settings for specified cursor

Syntax

```
getCursorInfo(scope,'CursorTag',tag)
```

Description

`getCursorInfo(scope,'CursorTag',tag)` returns the settings for the cursor or cursors, specified by the input tag.

Input Arguments

`scope` — The Logic Analyzer object from which you want to return cursor settings
dsp.LogicAnalyzer object

The Logic Analyzer object from which you want to return cursor settings, specified as a handle to the `dsp.LogicAnalyzer` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'CursorTag','C5'` specifies to return settings about the cursor with tag `'C5'`.

`'CursorTag'` — The tag or tags identifying the cursor or cursors about which to get information
character vector or cell array of character vectors

The tag or tags identifying the cursor or cursors about which to get information, specified as a character vector or cell array of character vectors.

Example: `'CursorTag','C5'`

Example: `'CursorTag',{'C4','C5'}`

Data Types: `char` | `cell`

Output Arguments

`cursorInfo` — Information about settings for the cursor or cursors

struct

The `cursorInfo` struct contains the following fields:

- `Location` — Location of the cursors
- `Color` — Color of the cursors
- `Locked` — Locked status of the cursors
- `Tag` — Tag identifying cursors

getCursorTags

System object: dsp.LogicAnalyzer
Package: dsp

Return all cursor tags

Syntax

```
T = getCursorTags(scope)
```

Description

`T = getCursorTags(scope)` returns all the cursor tags for the Logic Analyzer. You can use these tags to get information about a cursor using the dsp.LogicAnalyzer.getCursorInfo method, to modify a cursor using the dsp.LogicAnalyzer.modifyCursor method, or to delete a cursor using the dsp.LogicAnalyzer.deleteCursor method.

Input Arguments

`scope` — The Logic Analyzer object from which you want to return all cursor tags
`dsp.LogicAnalyzer` object

The Logic Analyzer object from which you want to return all cursor tags, specified as a handle to the `dsp.LogicAnalyzer` object.

Output Arguments

`T` — The cursor tags
cell array of character vectors

The cursor tags, specified as a cell array of character vectors.

Example: `{'C1'}`

Example: `{'C1','C2','C3'}`

Data Types: `cell`

getDisplayChannelInfo

System object: dsp.LogicAnalyzer
Package: dsp

Return settings for specified display channel

Syntax

`channelInfo = getDisplayChannelInfo(scope,'DisplayChannelTag',tag)`

Description

`channelInfo = getDisplayChannelInfo(scope,'DisplayChannelTag',tag)` returns the settings for the display channel or channels, specified by the input tag.

Input Arguments

`scope` — The Logic Analyzer object from which you want to return display channel settings
`dsp.LogicAnalyzer` object

The Logic Analyzer object from which you want to return display channel settings, specified as a handle to the `dsp.LogicAnalyzer` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DisplayChannelTag','W5'` specifies to return display channel settings about the wave with tag `'W5'`.

`'DisplayChannelTag'` — The tag or tags identifying the display channel or channels about which to get information
character vector or cell array of character vectors

The tag or tags identifying the display channel or channels about which to get information, specified as a character vector or cell array of character vectors.

Example: `'DisplayChannelTag','W5'`

Example: `'DisplayChannelTag',{'W4','W5'}`

Data Types: `char` | `cell`

Output Arguments

`channelInfo` — Information about settings for the display channel or channels

struct

The `channelInfo` struct contains the following fields:

- `Color` — Color of the waves.
- `InputChannel` — Channel on the display that corresponds to the specified waves.
- `Radix` — Radix for the waves.
- `FontSize` — Font size for values in the waves. A value of `0` indicates that the waves inherit `FontSize` from the global `DisplayChannelColor` property.
- `Name` — The name or label for the waves.
- `Height` — Height of the wave. A value of `0` indicates that the waves inherit `Height` from the global `DisplayChannelHeight` property.
- `Tag` — Tag for the channel.

getDisplayChannelTags

System object: dsp.LogicAnalyzer
Package: dsp

Return all display channel tags

Syntax

```
T = getDisplayChannelTags(scope)
```

Description

`T = getDisplayChannelTags(scope)` returns all the display channel tags for the Logic Analyzer. These tags are particularly useful for getting information about a wave or divider using the dsp.LogicAnalyzer.getDisplayChannelInfo method, for modifying a wave or divider using the dsp.LogicAnalyzer.modifyDisplayChannel method, or for deleting a wave or divider using the dsp.LogicAnalyzer.deleteDisplayChannel method.

Input Arguments

`scope` — The Logic Analyzer object from which you want to return all display channel tags
`dsp.LogicAnalyzer` object

The Logic Analyzer object from which you want to return all display channel tags, specified as a handle to the `dsp.LogicAnalyzer` object.

Output Arguments

`T` — The display channel tags
cell array of character vectors

The display channel tags, returned as a cell array of character vectors.

Example: `{'W1'}`

Example: `{'W1','W2','W3'}`

Data Types: `cell`

hide

System object: dsp.LogicAnalyzer
Package: dsp

Hide scope window

Syntax

`hide(scope)`

Description

`hide(scope)` hides the window associated with System object, `scope`.

See Also

dsp.LogicAnalyzer.show

modifyCursor

System object: dsp.LogicAnalyzer
Package: dsp

Modify properties of specified cursor

Syntax

```
modifyCursor(scope,tag)
modifyCursor(scope,'CursorTag',tag)
modifyCursor(scope,tag,'Name',Value)
modifyCursor(scope,'CursorTag',tag,'Name',Value)
```

Description

`modifyCursor(scope,tag)` modifies the properties of the cursor specified by the input tag.

`modifyCursor(scope,'CursorTag',tag)` modifies the properties of the cursor specified by the input tag.

`modifyCursor(scope,tag,'Name',Value)` modifies the properties of the cursor specified by the input tag with each specified property *Name* set to the specified value. You can specify *Name*–*Value* arguments in any order.

`modifyCursor(scope,'CursorTag',tag,'Name',Value)` modifies the properties of the cursor specified by the input tag with each specified property *Name* set to the specified value. You can specify *Name*–*Value* arguments in any order.

Input Arguments

`scope` — The Logic Analyzer object for which you want to modify a cursor

`dsp.LogicAnalyzer` object

The Logic Analyzer object for which you want to modify a cursor specified, as a handle to the `dsp.LogicAnalyzer` object.

`tag` — The tag identifying which cursor to modify
character vector, randomly assigned

The tag identifying which cursor to modify specified, as a randomly assigned character vector.

Example: `modifyCursor(scope,tag)` modifies a cursor in Logic Analyzer.

Example: `scope.modifyCursor(tag)` modifies a cursor in Logic Analyzer.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Location',2,'Color','Blue'` specifies that a cursor should be moved to the 2-second mark and colored blue.

`'Color'` — Color of the cursor
`'Yellow'` (default) | character vector | three element vector

Color of the cursor, specified as an `[R G B]` value or one of the following:

- `'Black'`
- `'Blue'`
- `'Cyan'`
- `'Green'`
- `'Magenta'`
- `'White'`
- `'Yellow'`

Example: `'Color','Blue'`

Example: `'Color',[0,0,1]`

Data Types: `char` | `double`

`'Location'` — Location of the cursor
`0` (default) | numeric scalar

Specify as a numeric scalar value, in seconds, the cursor location.

Example: `'Location',1`

Data Types: `double`

`'Locked'` — Locked status of the cursor

`false` (default) | `true`

Locked status of the cursor, specified as `false` or `true`.

- `true` — the cursor's location cannot be changed. Logic Analyzer denotes this by assigning a default color of red.
- `false` — the cursor's location can be changed. Logic Analyzer denotes this by assigning a default color of yellow.

Example: `'Locked',true`

modifyDisplayChannel

System object: dsp.LogicAnalyzer
Package: dsp

Modify properties of specified display channel

Syntax

```
modifyDisplayChannel(scope,tag,'Name',Value)
modifyDisplayChannel(scope,'DisplayChannelTag',tag,'Name',Value)
```

Description

`modifyDisplayChannel(scope,tag,'Name',Value)` modifies the properties of the display channel specified by the input tag with each specified property *Name* set to the specified value. You can specify *Name–Value* arguments in any order.

`modifyDisplayChannel(scope,'DisplayChannelTag',tag,'Name',Value)` modifies the properties of the display channel specified by the input tag with each specified property *Name* set to the specified value. You can specify *Name–Value* arguments in any order.

Input Arguments

`scope` — The Logic Analyzer object for which you want to modify a display channel
`dsp.LogicAnalyzer` object

The Logic Analyzer object for which you want to modify a display channel, specified as a handle to the `dsp.LogicAnalyzer` object.

`tag` — The tag identifying which display channel to modify
character vector, randomly assigned

The tag identifying which display channel to modify, specified as a randomly assigned character vector.

Example: `modifyDisplayChannel(scope,tag)` modifies a display channel in Logic Analyzer.

Example: `scope.modifyDisplayChannel(tag)` modifies a display channel in Logic Analyzer.

The first section on Name-Value Pair Arguments shows the properties you can set if the display channel contains a wave. The second section on Name-Value Pair Arguments shows the properties you can set if the display channel contains a divider.

Wave Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'InputChannel',2,'Color','Blue'` specifies that a wave should be added to input channel 1 and colored blue.

`'Color'` — Color of the wave
`'Default'` (default) | character vector

Color of the wave, specified as an `[R G B]` value or one of the following:

- `'Black'`
- `'Blue'`
- `'Cyan'`
- `'Default'`
- `'Green'`
- `'Magenta'`
- `'Red'`
- `'White'`
- `'Yellow'`

When you choose `'Default'`, the value of the `DisplayChannelColor` property in the Logic Analyzer is used.

Example: `'Color','Blue'`

Example: `'Color',[0,0,1]`

Data Types: `char` | `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`'DisplayChannel'` — Channel on the display that shows this wave
`NumInputPorts` (default) | scalar numeric value in the range (1,`NumInputPorts`)

Specify as a scalar numeric value the display channel that shows this wave. By default, the wave is added to the end of the display.

Example: `'DisplayChannel',2`

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`'FontSize'` — Font size for values in the wave
`0` (default) | scalar nonnegative integer

Specify as a scalar nonnegative integer the font size in points. When you choose 0, the value of the `DisplayChannelFontSize` property in the Logic Analyzer is used.

Example: `'FontSize',8`

Data Types: `double`

`'Format'` — Display format for the wave
`'Default'` (default) | character vector

`'Format'` must be one of the following.

- `'Analog'`
- `'Default'`
- `'Digital'`

When you choose `'Default'`, the value of the `DisplayChannelFormat` property in the Logic Analyzer is used.

Example: `'Format','Digital'`

Data Types: `char`

`'Height'` — Height of the wave
`0` (default) | scalar integer

Specify as a scalar integer the height of the wave in the display in units of 16 pixels. When you choose 0, the value of the `DisplayChannelHeight` property in the Logic Analyzer is used.

Example: `'Height',2`

Data Types: `double`

`'InputChannel'` — Input channel that corresponds to this wave

`1` (default) | scalar integer in the range (1,`NumInputPorts`)

This property specifies the input channel whose data is used for this wave. By default, it will connect the first input to this wave.

Example: `'InputChannel',2`

Data Types: `double` | `single` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`'Name'` — The name or label for the wave

`''` (default) | character vector

Specify as a character vector the name that you would like to set for the new wave.

Example: `'Name','MyWave'`

Data Types: `char`

`'Radix'` — Radix for the wave

`'Default'` (default) | character vector

`'Radix'` must be one of the following.

- `'Binary'`
- `'Default'`
- `'Hexadecimal'`
- `'Octal'`
- `'Signed decimal'`
- `'Unsigned decimal'`

When the input signals are of class double, single, or logical, you should not set this property. When you choose `'Default'`, the value of the `DisplayChannelRadix` property in the Logic Analyzer is used.

Example: 'Radix','Hexadecimal'

Data Types: char

Divider Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'DisplayChannel',2,'Name','MyDivider' specifies that a divider should be added to display channel 2 and named “MyDivider”.

'DisplayChannel' — Channel on the display that shows this divider

NumInputPorts (default) | scalar numeric value in the range (1,NumInputPorts)

Specify as a scalar numeric value the display channel that shows this divider. By default, the divider is added to the end of the display.

Example: 'DisplayChannel',2

Data Types: double | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

'Height' — Height of the divider

0 (default) | scalar integer

Specify, in pixels, the height of the divider as a scalar integer in the range 8 to 200. If you choose 0, the value of the DisplayChannelHeight property in the Logic Analyzer is used.

Example: 'Height',2

Data Types: double

'Name' — The name or label for the divider

'' (default) | character vector

Specify as a character vector the name that you would like to set for the new divider.

Example: 'Name','MyDivider'

Data Types: char

moveDisplayChannel

System object: dsp.LogicAnalyzer
Package: dsp

Move position of specified display channel

Syntax

```
moveDisplayChannel(scope,tag,'DisplayChannel',displayChannelValue)
moveDisplayChannel(
scope,'DisplayChannelTag',tag,'DisplayChannel',displayChannelValue)
```

Description

`moveDisplayChannel(scope,tag,'DisplayChannel',displayChannelValue)` moves the display channel, either a wave or a divider, specified by the input `tag`, to the new location specified by the input `displayChannelValue`.

`moveDisplayChannel(`
`scope,'DisplayChannelTag',tag,'DisplayChannel',displayChannelValue)` moves the display channel, either a wave or a divider, specified by the input `tag`, to the new location specified by the input `displayChannelvValue`.

Input Arguments

`scope` — The Logic Analyzer object in which you want to move a display channel

`dsp.LogicAnalyzer` object

The Logic Analyzer object in which you want to move a display channel, specified as a handle to the `dsp.LogicAnalyzer` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DisplayChannelTag','W1'` specifies to delete the divider with tag `'W1'`.

`'DisplayChannelTag'` — The tag identifying which display channel to move
character vector

The tag identifying which display channel to move, specified as a character vector.

Example: `'DisplayChannelTag','W1'`

Data Types: `char`

`'DisplayChannel'` — The location identifying where the display channel should be moved
scalar integer

The location identifying where the display channel should be moved, specified as a scalar integer.

Example: `'DisplayChannel',2`

Data Types: `double`

reset

System object: dsp.LogicAnalyzer
Package: dsp

Reset internal states of scope object

Syntax

```
reset(scope)
```

Description

`reset(scope)` sets the internal states of the scope object to their initial values.

You should call the `reset` method after calling the `step` method when you want to clear the scope figure displays, prior to releasing system resources. This action enables you to start a simulation from the beginning. When you call the `reset` method, the displays will become blank again. In this sense, its functionality is similar to that of the MATLAB `clf` function. Do not call the `reset` method after calling the `release` method.

Algorithms

In operation, the `reset` method is similar to a consecutive execution of the `mdlTerminate` function and the `mdlInitializeConditions` function.

See Also

dsp.LogicAnalyzer

show

System object: dsp.LogicAnalyzer
Package: dsp

Make scope window visible

Syntax

`show(scope)`

Description

`show(scope)` makes the window associated with System object, `scope`, visible.

See Also

dsp.LogicAnalyzer.hide

step

System object: dsp.LogicAnalyzer
Package: dsp

Display signal in scope figure

Syntax

```
step(scope,X)
step(scope,X1,X2,...,XN)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`step(scope,X)` displays the signal, `X`, in the scope figure.

`step(scope,X1,X2,...,XN)` displays the signals `X1`, `X2`,...,`XN` in the scope figure when you set the `NumInputPorts` property to `N`. In this case, `X1`, `X2`,...,`XN` can have different data types and dimensions.

dsp.LowerTriangularSolver System object

Package: dsp

Solve lower-triangular matrix equation

Description

The `LowerTriangularSolver` object solves $LX = B$ for X when L is a square, lower-triangular matrix with the same number of rows as B.

To solve $LX = B$ for X:

1. Define and set up your linear system solver. See “Construction” on page 4-1170.
2. Call step to solve the equation according to the properties of `dsp.LowerTriangularSolver`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`lowtriang = dsp.LowerTriangularSolver` returns a linear system solver, `lowtriang`, used to solve the linear system $LX = B$, where L is a lower (or unit-lower) triangular matrix.

`lowtriang = dsp.LowerTriangularSolver('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a linear system solver, `lowtriang`, with each specified property set to the specified value.

Properties

OverwriteDiagonal

Replace diagonal elements of input with ones

When you set this property to `true`, the linear system solver replaces the elements on the diagonal of the input, *L*, with ones. Set this property to either `true` or `false`. The default is `false`.

RealDiagonalElements

Indicate that diagonal of complex input is real

When you set this property to `true`, the linear system solver optimizes computation speed if the diagonal elements of complex input, *L*, are real. This property applies only when you set the `OverwriteDiagonal` on page 4-1171 property to `false`. Set this property to either `true` or `false`. The default is `false`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

ProductDataType

Data type of product

Specify the product data type as `Full precision`, `Same as input`, or `Custom`. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `ProductDataType` on page 4-1171 property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Data type of accumulator

Specify the accumulator data type as `Full precision`, `Same as first input`, `Same as product`, or `Custom`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `AccumulatorDataType` on page 4-1172 property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Data type of output

Specify the output data type as `Same as first input` or `Custom`. The default is `Same as first input`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` on page 4-1172 property to `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Solve matrix equation for specified inputs

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Solve a Lower Triangular Matrix

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
lowtriang = dsp.LowerTriangularSolver;
u = tril(rand(4, 4));
b = rand(4, 1);
```

Check that result is the solution to the linear equations.

```
x1 = inv(u)*b
x = lowtriang(u, b)


x1 =

    0.5177
    4.5810
   -3.4853
    9.6147


x =

    0.5177
    4.5810
   -3.4853
    9.6147
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Forward Substitution` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

dsp.UpperTriangularSolver

Introduced in R2012a

step

System object: dsp.LowerTriangularSolver
Package: dsp

Solve matrix equation for specified inputs

Syntax

`X = step(lowtriang,L,B)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`X = step(lowtriang,L,B)` computes the solution, X, of the matrix equation $LX = B$, where L is a square, lower-triangular matrix with the same number of rows as the matrix B.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.LowpassFilter System object

Package: dsp

FIR or IIR lowpass filter

Description

`dsp.LowpassFilter` independently filters each channel of the input over time using the given design specifications. You can set the `FilterType` property of `dsp.LowpassFilter` to `'FIR'` or `'IIR'` to implement the object as a FIR or IIR lowpass filter. This object supports fixed-point operations, HDL code generation, and ARM Cortex code generation.

To filter each channel of your input:

1. Define and set up your lowpass filter. See "Construction" on page 4-1176.
2. Call step to filter each channel of the input signal according to the properties of `dsp.LowpassFilter`. The input signal can be a real- or complex-valued column vector or matrix, with floating point or fixed-point precision. If the input signal is a matrix, each column of the matrix is treated as an independent channel. The number of rows in the input signal denote the channel length.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`LPF = dsp.LowpassFilter` returns a minimum order FIR lowpass filter, `LPF`, with the default filter settings. Calling `step` with the default property settings filters the input data with a passband frequency of `8` kHz, a stopband frequency of `12` kHz, a passband ripple of `0.1` dB, and a stopband attenuation of `80` dB.

`LPF = dsp.LowpassFilter(Name,Value)` returns a lowpass filter, with additional properties specified by one, or more `Name,Value` pair arguments. `Name` is the property name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Properties

`SampleRate` — Input sample rate
`44100` (default) | real positive scalar

Input sample rate in Hz, specified as the comma-separated pair consisting of `'SampleRate'` and a real positive scalar.

`FilterType` — Filter type
`'FIR'` (default) | `'IIR'`

Filter type, specified as one of the following options:

- `'FIR'` — The object designs an FIR lowpass filter.
- `'IIR'` — The object designs an IIR lowpass (biquad) filter.

`DesignForMinimumOrder` — Minimum order filter design
`true` (default) | `false`

Minimum order filter design, specified as the comma-separated pair consisting of `'DesignForMinimumOrder'` and a logical value. If this property is `true`, then `dsp.LowpassFilter` designs filters with the minimum order that meets the passband frequency, stopband frequency, passband ripple, and stopband attenuation specifications. Set these specifications using the corresponding properties. If this property is `false`, then the object designs filters with the order that you specify in the `FilterOrder` property. This filter design meets the passband frequency, passband ripple, and stopband attenuation specifications that you set using the respective properties.

`FilterOrder` — Order of the FIR or IIR filter
`50` (default) | positive integer scalar

Order of the FIR or IIR filter, specified as the comma-separated pair consisting of `'FilterOrder'` and a positive integer scalar. Specifying a filter order is only valid when the value of `'DesignForMinimumOrder'` is `false`.

`PassbandFrequency` — Filter passband edge frequency
`8000` (default) | real positive scalar

Filter passband edge frequency in Hz, specified as the comma-separated pair of 'PassbandFrequency' and a real positive scalar. The value of the passband edge frequency in Hz must be less than half the SampleRate.

StopbandFrequency — Filter stopband edge frequency
12000 (default) | real positive scalar

Filter stopband edge frequency in Hz, specified as the comma-separated pair consisting of 'StopbandFrequency' and a real positive scalar. The value of the stopband edge frequency in Hz must be less than half the SampleRate. You can specify the stopband edge frequency only when 'DesignForMinimumOrder' is true.

PassbandRipple — Maximum ripple of filter response in the passband
0.1 (default) | real positive scalar

Maximum ripple of filter response in the passband, in dB, specified as the comma-separated pair consisting of 'PassbandRipple' and a real positive scalar. Maximum ripple of filter response defaults to 0.1 dB.

StopbandAttenuation — Minimum attenuation in the stopband
80 (default) | real positive scalar

Minimum attenuation in the stopband in dB, specified as the comma-separated pair consisting of 'StopbandAttenuation' and a real positive scalar. Minimum attenuation in the stopband defaults to 80 dB.

Fixed-Point Properties

RoundingMethod — Rounding method for output fixed-point operations
'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

Rounding method for output fixed-point operations, specified as a character vector. For more information on the rounding modes, see "Precision and Range".

CoefficientsDataType — Word and fraction lengths of coefficients
numerictype(1,16) (default) | numerictype object

Word and fraction lengths of coefficients, specified as a numerictype object. The default, numerictype(1,16) corresponds to a signed numeric type object with 16-bit coefficients and a fraction length determined based on the coefficient values, to give the best possible precision.

This property is not tunable.

Word length of the output is same as the word length of the input. Fraction length of the output is computed such that the entire dynamic range of the output can be represented without overflow. For details on how the fraction length of the output is computed, see “Fixed-Point Precision Rules for Avoiding Overflow in FIR Filters”.

Methods

measure	Measure frequency response characteristics of `dsp.LowpassFilter` System object.
reset	Reset internal states of lowpass filter
step	Filter input using FIR or IIR lowpass filter

For additional methods, see “Analysis Methods for Filter System Objects” on page 3-2.

For a complete list of analysis methods supported for the `dsp.LowpassFilter` object, enter `dsp.LowpassFilter.helpFilterAnalysis` at the MATLAB command prompt.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Impulse and Frequency Response of FIR and IIR Lowpass Filters

Create a minimum order FIR lowpass filter for data sampled at 44.1 kHz. Specify a passband frequency of 8 kHz, a stopband frequency of 12 kHz, a passband ripple of 0.1 dB, and a stopband attenuation of 80 dB.

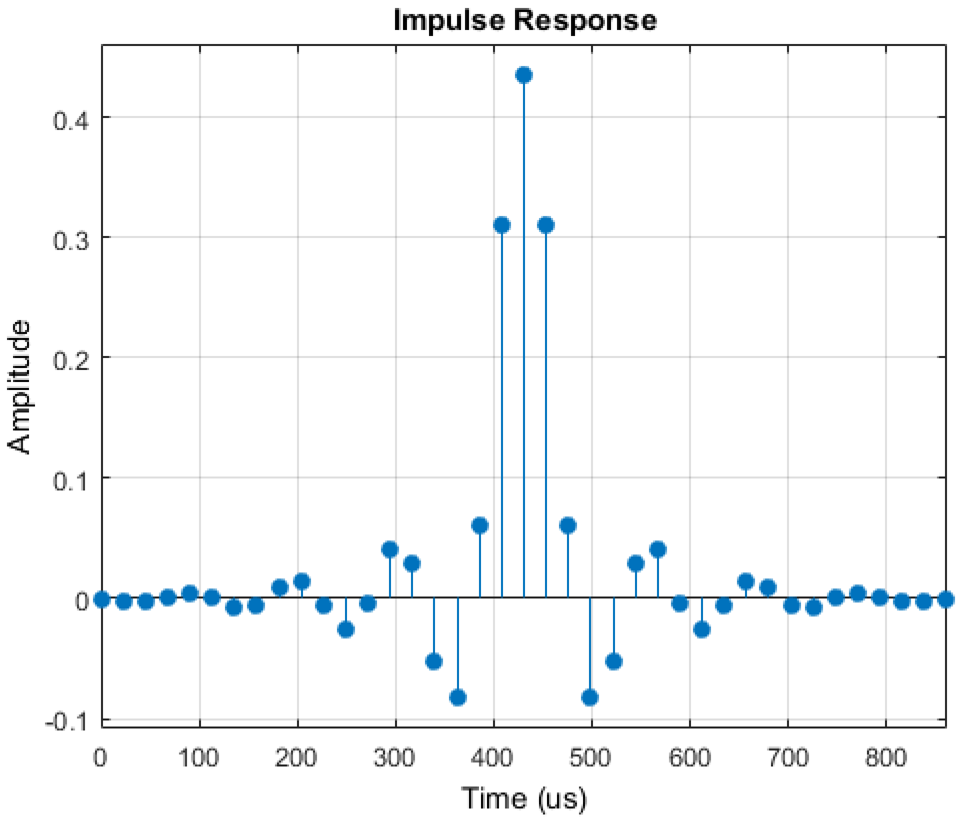
```
Fs = 44.1e3;
filtertype = 'FIR';
Fpass = 8e3;
Fstop = 12e3;
Rp = 0.1;
Astop = 80;
FIRLPF = dsp.LowpassFilter('SampleRate',Fs,...
                             'FilterType',filtertype,...
                             'PassbandFrequency',Fpass,...
                             'StopbandFrequency',Fstop,...
                             'PassbandRipple',Rp,...
                             'StopbandAttenuation',Astop);
```

Design a minimum order IIR lowpass filter with the same properties as the FIR lowpass filter. Use `clone` to create a system object with the same properties as the FIR lowpass filter. Change the `FilterType` property of the cloned filter to `IIR`.

```
IIRLPF = clone(FIRLPF);
IIRLPF.FilterType = 'IIR';
```

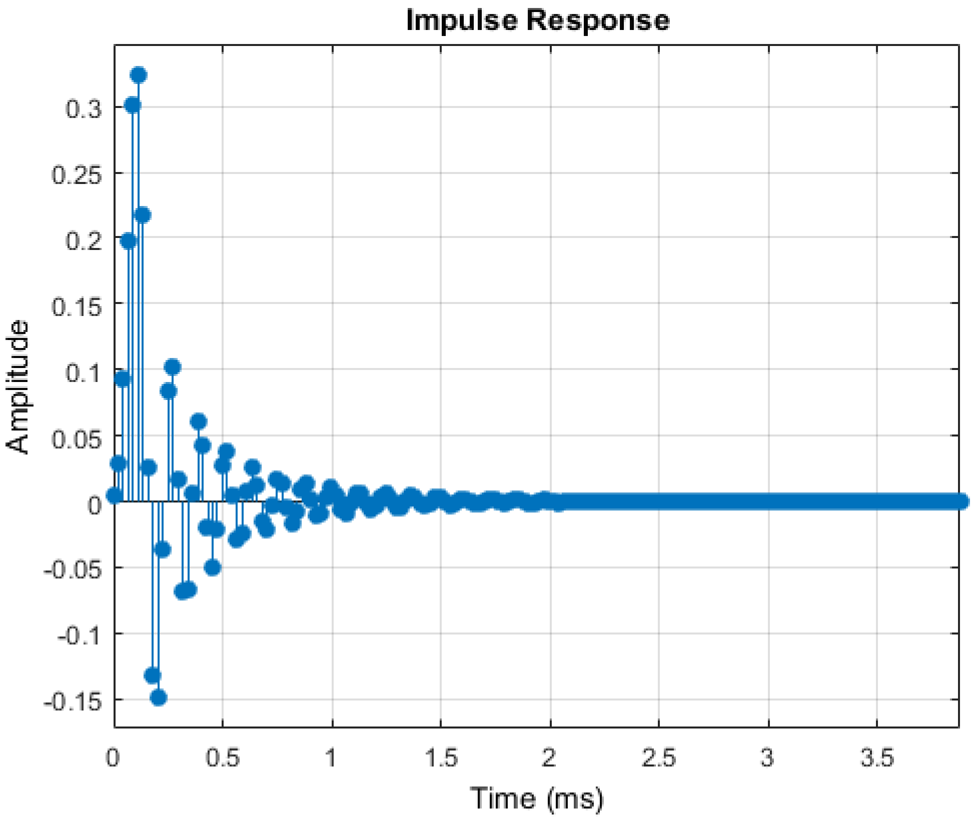
Plot the impulse response of the FIR lowpass filter. The zeroth order coefficient is delayed by 19 samples, which is equal to the group delay of the filter. The FIR lowpass filter is a causal FIR filter.

```
fvtool(FIRLPF,'Analysis','impulse')
```



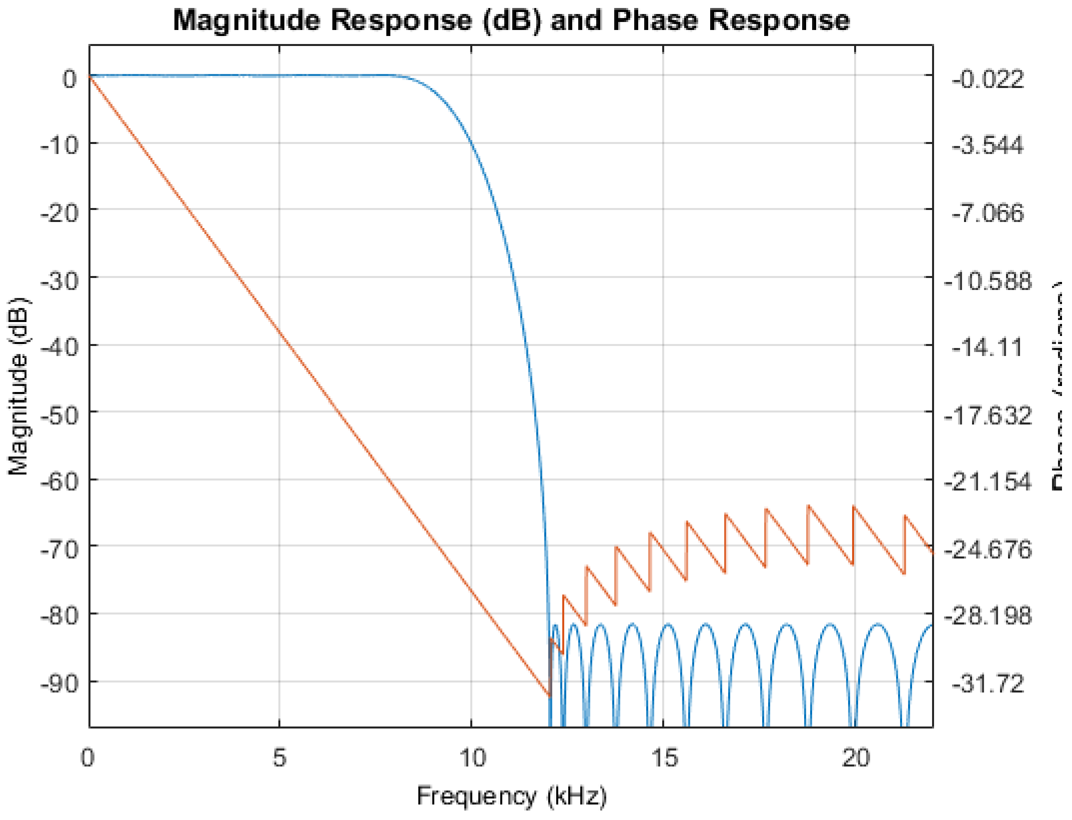
Plot the impulse response of the IIR lowpass filter.

```
fvtool(IIRLPF,'Analysis','impulse')
```



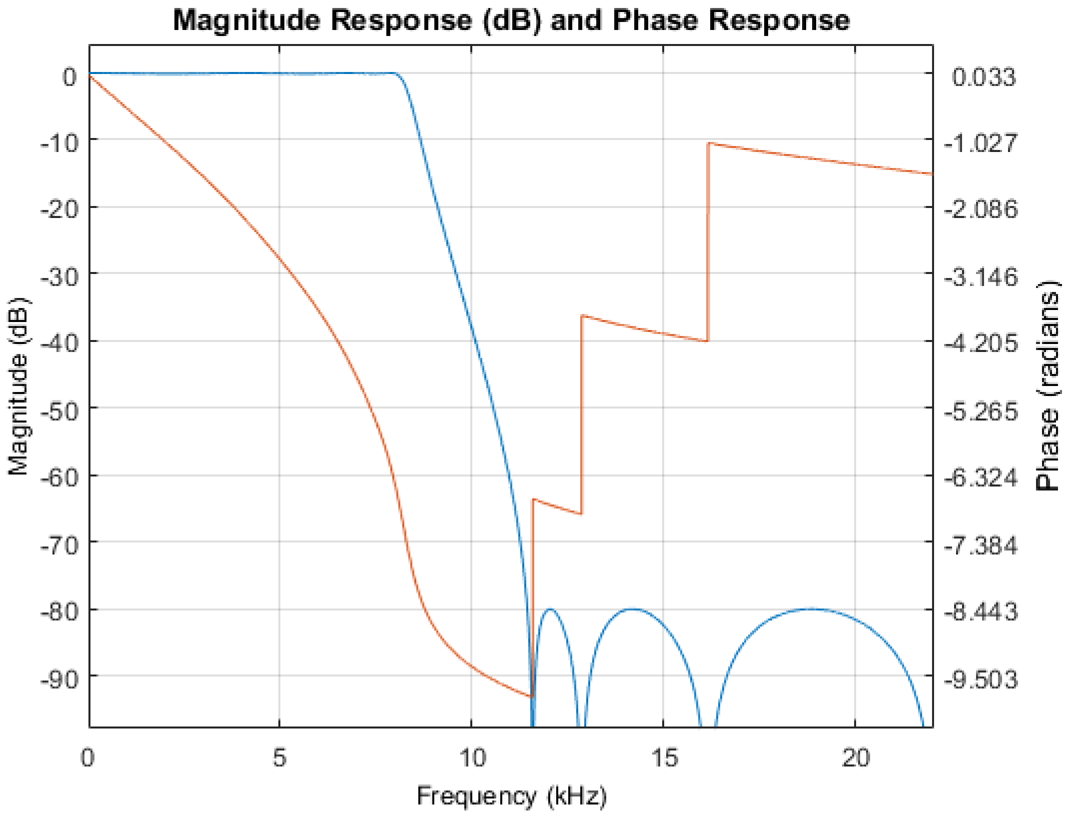
Plot the magnitude and phase response of the FIR lowpass filter.

```
fvtool(FIRLPF,'Analysis','freq')
```



Plot the magnitude and phase response of the IIR lowpass filter.

```
fvtool(IIRLPF,'Analysis','freq')
```



Calculate the cost of implementing the FIR lowpass filter.

```
cost(FIRLPF)
```

```
ans =

  struct with fields:

                NumCoefficients: 39
                      NumStates: 38
    MultiplicationsPerInputSample: 39
          AdditionsPerInputSample: 38
```

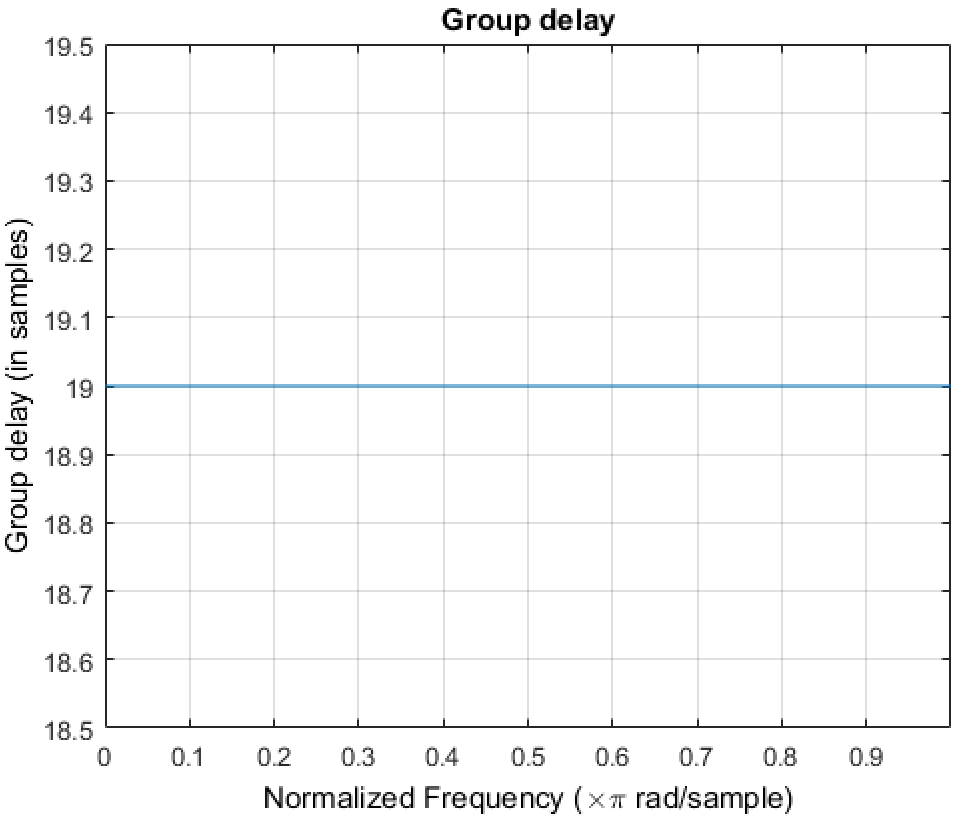
Calculate the cost of implementing the IIR lowpass filter. The IIR filter is more efficient to implement than its FIR counterpart.

```
cost(IIRLPF)
```

```
ans = 

  struct with fields:

                  NumCoefficients: 18
                        NumStates: 14
    MultiplicationsPerInputSample: 18
          AdditionsPerInputSample: 14
```

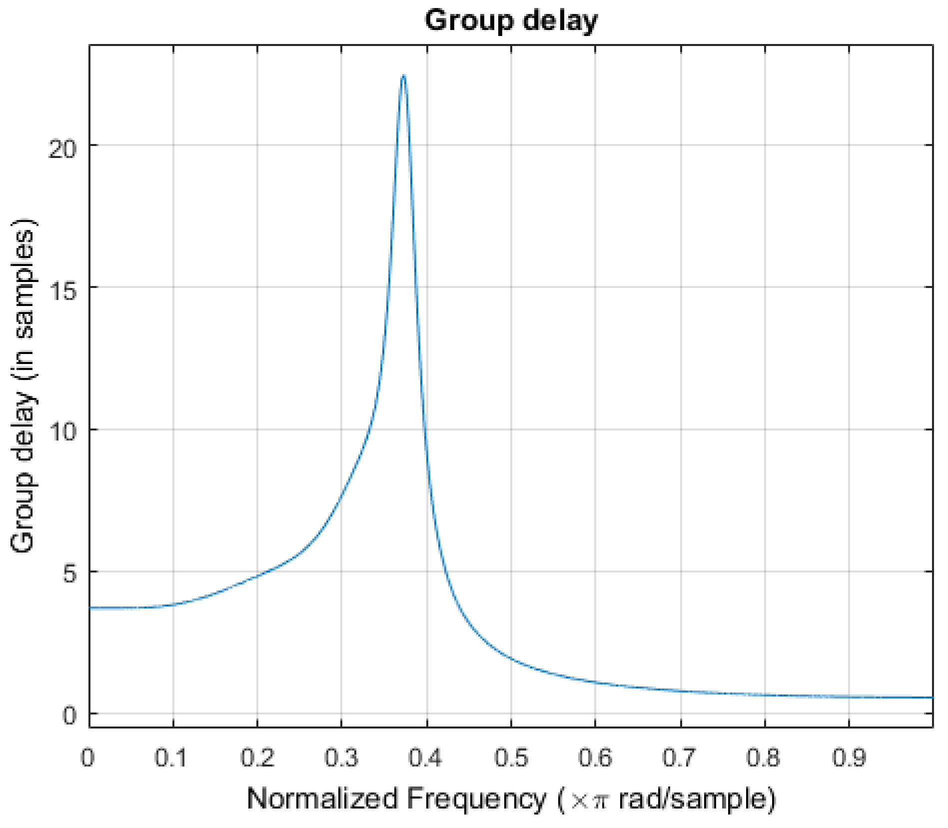
Calculate the group delay of the FIR lowpass filter.

```
grpdelay(FIRLPF)
```



Calculate the group delay of the IIR lowpass filter. The FIR filter has a constant group delay (linear phase), while its IIR counterpart does not.

```
grpdelay(IIRLPF)
```



Filter White Gaussian Noise

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

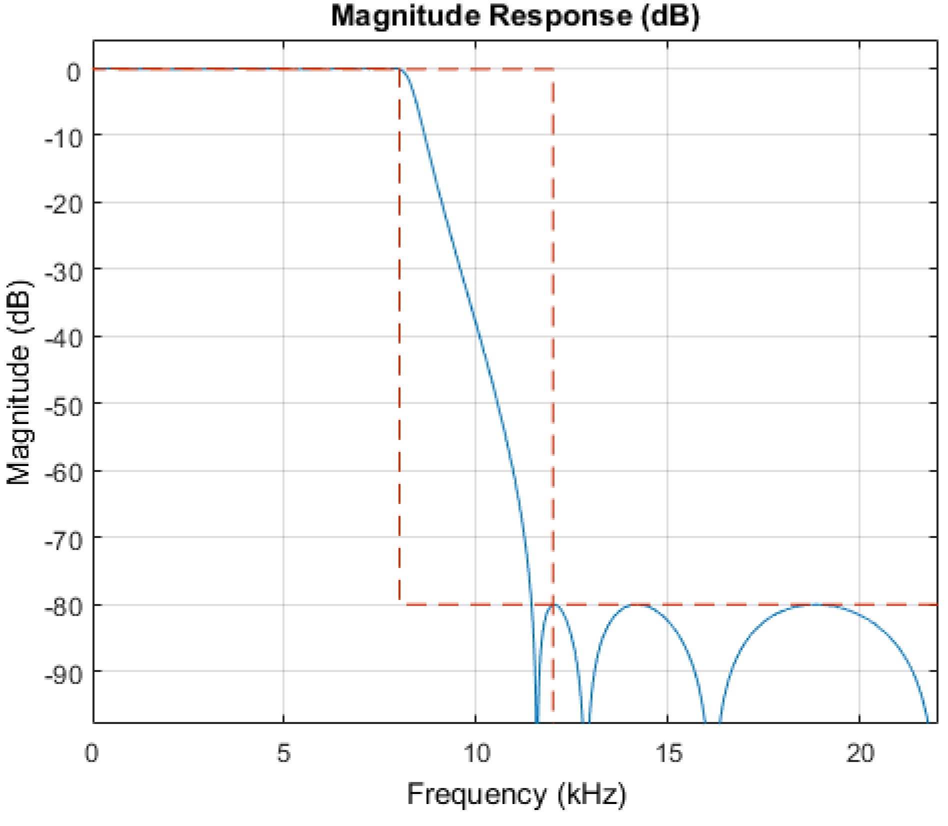
Set up the IIR lowpass filter. The sampling rate of the white Gaussian noise is 44,100 Hz. The passband frequency of the filter is 8 kHz, the stopband frequency is 12 kHz, the passband ripple is 0.1 dB, and the stopband attenuation is 80 dB.

```
Fs = 44.1e3;
filtertype = 'IIR';
Fpass = 8e3;
Fstop = 12e3;
```

```
Rp = 0.1;
Astop = 80;
LPF = dsp.LowpassFilter('SampleRate',Fs,...
                        'FilterType',filtertype,...
                        'PassbandFrequency',Fpass,...
                        'StopbandFrequency',Fstop,...
                        'PassbandRipple',Rp,...
                        'StopbandAttenuation',Astop);
```

View the magnitude response of the lowpass filter.

```
fvtool(LPF)
```

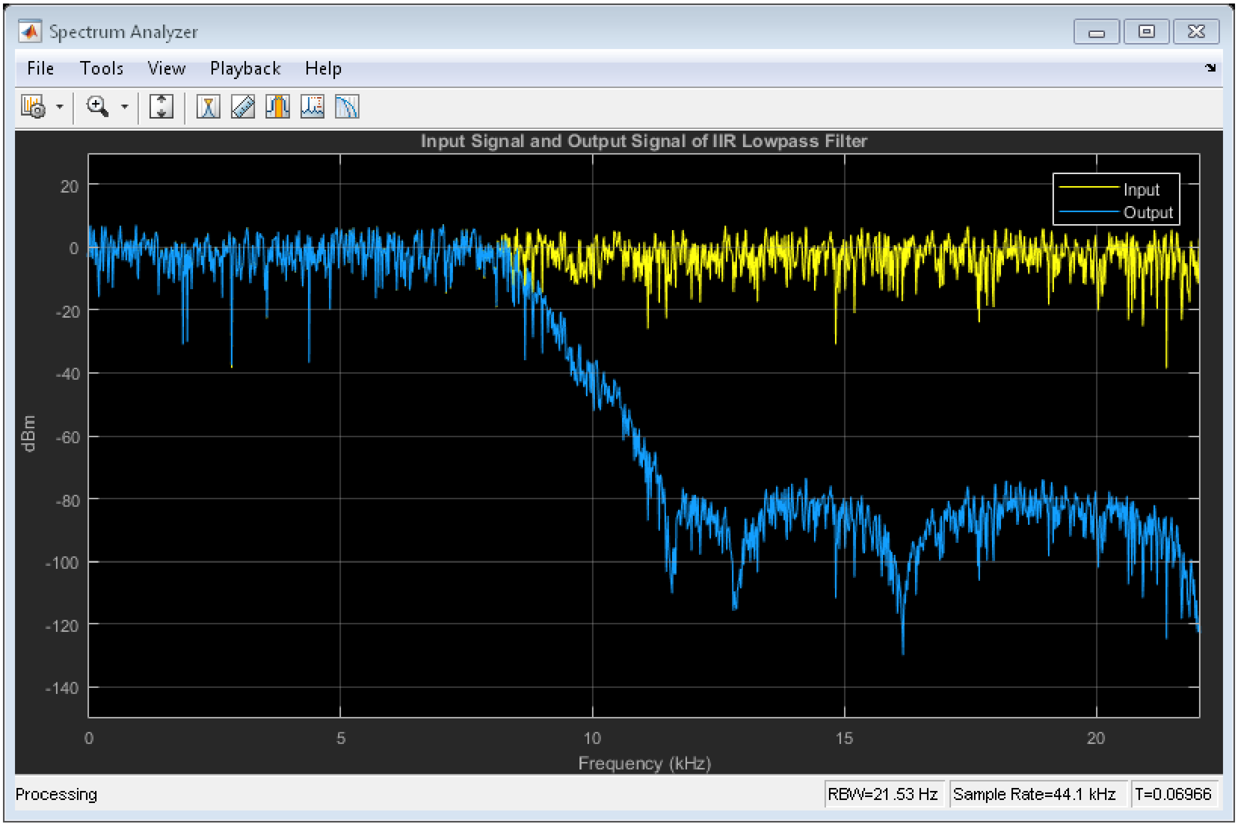


Create a spectrum analyzer object.

```
hSA = dsp.SpectrumAnalyzer('SampleRate',44.1e3,...
    'PlotAsTwoSidedSpectrum',false,'ShowLegend',true,'YLimits',...
           [-150 30],...
           'Title',...
           'Input Signal and Output Signal of IIR Lowpass Filter');
hSA.ChannelNames = {'Input','Output'};
```

Filter the white Gaussian noisy input signal. View the input and output signals using the spectrum analyzer.

```
for k = 1:100
    Input = randn(1024,1);

    Output = LPF(Input);

    hSA([Input,Output]);
end
```



Supported Data Types

Argument	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point
Output	• Double-precision floating point • Single-precision floating point • Fixed point

Algorithms

FIR Lowpass Filter

When the `FilterType` property is set to `'FIR'`, the `dsp.LowpassFilter` object acts as a FIR lowpass filter.

In this configuration, `dsp.LowpassFilter` is an alternative to using `firceqrip` and `firgr` with `dsp.FIRFilter`. This object condenses the two-step process into one. For the minimum order design, the object uses generalized Remez FIR filter design algorithm. For the specified order design, the object uses the constrained equiripple FIR filter design algorithm. The designed filter is then implemented as a linear phase Type-1 filter with a `Direct form` structure. You can use dsp.LowpassFilter.measure to verify that the design meets the prescribed specifications.

IIR Lowpass Filter

When the `FilterType` property is set to `'IIR'`, the `dsp.LowpassFilter` object acts as an IIR lowpass filter. In this configuration, the object uses the elliptic design method to compute the SOS and scale values required to meet the filter design specifications. The object uses the SOS and scale values to setup a `Direct form I` biquadratic IIR filter, which forms the basis of the IIR version of the `dsp.LowpassFilter` System object. You can use dsp.LowpassFilter.measure to verify that the design meets the prescribed specifications.

References

[1] Shpak, D.J., and A. Antoniou. "A generalized Remez method for the design of FIR digital filters." *IEEE Transactions on Circuits and Systems*. Vol. 37, Issue 2, Feb. 1990, pp. 161–174.

[2] Selesnick, I.W., and C. S. Burrus. "Exchange algorithms that complement the Parks-McClellan algorithm for linear-phase FIR filter design." *IEEE Transactions on Circuits and Systems*. Vol. 44, Issue 2, Feb. 1997, pp. 137–143.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

dsp.HighpassFilter | “Lowpass Filter Design in MATLAB”

Introduced in R2015a

measure

System object: dsp.LowpassFilter
Package: dsp

Measure frequency response characteristics of `dsp.LowpassFilter` System object.

Syntax

`LPFMeas = measure(LPF)`

Description

`LPFMeas = measure(LPF)` measures the frequency response characteristics of the `dsp.LowpassFilter` System object, `LPF`.

Input Arguments

`LPF` — FIR or IIR lowpass filter

`LowpassFilter` System object

FIR or IIR lowpass filter, specified as a `LowpassFilter` System object.

Output Arguments

`LPFMeas` — Measurements of frequency response characteristics of the lowpass filter

`LowpassFilter` System object

Measurements of frequency response characteristics of the lowpass filter, returned as an `fdesign.lowpassmeas` object.

Examples

Measure Frequency Response Characteristics of Lowpass Filter

Measure the frequency response characteristics of a lowpass filter. Create a `dsp.LowpassFilter` System object with default properties. Measure the frequency response characteristics of the filter.

```
LPF = dsp.LowpassFilter
LPFMeas = measure(LPF)
```

```
LPF = 

  dsp.LowpassFilter with properties:

               FilterType: 'FIR'
    DesignForMinimumOrder: true
        PassbandFrequency: 8000
        StopbandFrequency: 12000
           PassbandRipple: 0.1000
      StopbandAttenuation: 80
               SampleRate: 44100

  Use get to show all properties


LPFMeas = 

Sample Rate      : 44.1 kHz
Passband Edge    : 8 kHz
3-dB Point       : 9.1311 kHz
6-dB Point       : 9.5723 kHz
Stopband Edge    : 12 kHz
Passband Ripple  : 0.08289 dB
Stopband Atten.  : 81.6141 dB
Transition Width : 4 kHz
```

Introduced in R2015a

reset

System object: dsp.LowpassFilter
Package: dsp

Reset internal states of lowpass filter

Syntax

`reset(LPF)`

Description

`reset(LPF)` resets the filter states of the lowpass filter, `LPF`, to their initial values. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the lowpass filter to nonzero input data, the filter might have nonzero states. If you Invoke the `step` method again without first invoking the `reset` method, the object might produce different outputs for an identical input.

Input Arguments

`LPF` — FIR or IIR lowpass filter

`LowpassFilter` System object

FIR or IIR lowpass filter, specified as a `LowpassFilter` System object.

Examples

Reset a Lowpass Filter

Create a lowpass filter with default properties.

```
LPF = dsp.LowpassFilter;
```

Create a two-channel random signal. Apply the `step` method twice on the signal.

```
x = randn(10,2);

y1 = step(LPF,x);
y2 = step(LPF,x);

no = all(y2==y1)

no =

  1×2 logical array

   0   0
```

The output is different because the internal states of `LPF` have changed. Use `reset` to reset the lowpass filter and apply `step` again. Verify that the output is unchanged.

```
reset(LPF)

y3 = step(LPF,x);

yes = all(y3==y1)

yes =

  1×2 logical array

   1   1
```

Introduced in R2015a

step

System object: dsp.LowpassFilter
Package: dsp

Filter input using FIR or IIR lowpass filter

Syntax

`Y = step(LPF,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(LPF,X)` filters the real or complex input signal, `X`, using the lowpass filter, `LPF`. `Y` is a lowpass-filtered version of `X`. The input `X` must be a column vector or matrix. If `X` is a matrix, each column is treated as an independent channel. `X` can be a floating point or a fixed-point input.

Input Arguments

`LPF` — FIR or IIR lowpass filter

`LowpassFilter` System object

FIR or IIR lowpass filter, specified as a `LowpassFilter` System object.

`X` — Input signal

vector | matrix

Input signal, specified as a vector or matrix with floating point or fixed-point precision. The row length of `X` is the frame size, that is, channel length. Each column of `X` is treated as a separate channel. The column length of `X` is the number of channels.

Output Arguments

Y — Lowpass-filtered signal

vector | matrix

Lowpass-filtered signal, returned as a vector or matrix of the same size, data type, and complexity as the input signal, X.

Examples

Filter White Gaussian Noise Signal With Lowpass Filter

Create a lowpass filter with default properties.

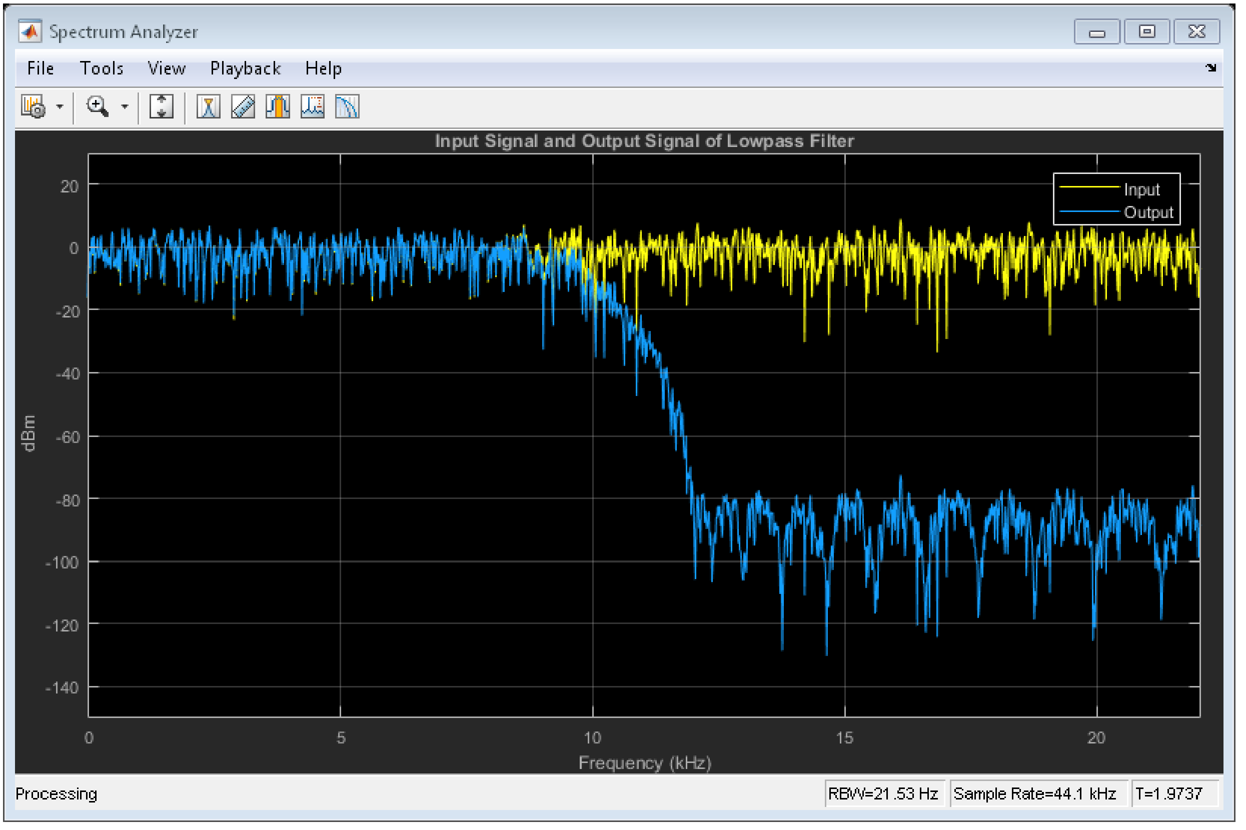
```
LPF = dsp.LowpassFilter;
```

Create a spectrum analyzer object.

```
hSA = dsp.SpectrumAnalyzer('SampleRate',44.1e3,...
    'PlotAsTwoSidedSpectrum',false,'ShowLegend',true,'YLimits',...
           [-150 30],...
           'Title',...
           'Input Signal and Output Signal of Lowpass Filter');
hSA.ChannelNames = {'Input','Output'};
```

Implement step on LPF to filter the white Gaussian noisy input signal. View the input and output signals using the spectrum analyzer.

```
for k = 1:100
    Input = randn(1024,1);

    Output = step(LPF,Input);

    step(hSA,[Input,Output]);
end
```



Introduced in R2015a

dsp.LPCToAutocorrelation System object

Package: dsp

Convert linear prediction coefficients to autocorrelation coefficients

Description

The `LPCToAutocorrelation` System object converts linear prediction coefficients to autocorrelation coefficients.

To convert LPC to autocorrelation coefficients:

1 Define and set up your LPC to autocorrelation object. See "Construction" on page 4-1200.
2 Call step to convert LPC according to the properties of `dsp.LPCToAutocorrelation`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`lpc2ac = dsp.LPCToAutocorrelation` returns an LPC to autocorrelation System object, `lpc2ac`, that converts linear prediction coefficients (LPC) to autocorrelation coefficients.

`lpc2ac = dsp.LPCToAutocorrelation('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an `LPC` to autocorrelation conversion object, `lpc2ac`, with each specified property set to the specified value.

Properties

`PredictionErrorInputPort`

Enable prediction error power input

Choose how to select the prediction error power. When you set this property to `true`, you must specify the prediction error power as a second input to the `step` method. When you set this property to `false`, the object assumes that the prediction error power is `1`. The default is `false`.

`NonUnityFirstCoefficientAction`

Action to take when first LPC coefficient is not `1`

Specify the action that the object takes when the first coefficient of each channel of the LPC input is not `1`. Select `Replace with 1` or `Normalize`. The default is `Replace with 1`.

Methods

step	Autocorrelation coefficients from LPC coefficients

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Convert LPC To Autocorrelation Coefficients

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Convert the linear prediction coefficients to autocorrelation coefficients.

```
a = [1.0 -1.4978 1.4282 -1.3930 0.9076 -0.3855 0.0711].';
lpc2ac = dsp.LPCToAutocorrelation;
ac = lpc2ac(a);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `LPC/RC to Autocorrelation` block reference page. The object properties correspond to the block parameters, except:
The object does not have a property that corresponds to the **Type of Conversion** block parameter. The object's behavior corresponds to the block's behavior when you set the **Type of Conversion** parameter to `LPC to autocorrelation`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also
dsp.LPCToRC | dsp.LPCToLSF | dsp.LPCToCepstral | dsp.RCToAutocorrelation

Introduced in R2012a

step

System object: dsp.LPCToAutocorrelation
Package: dsp

Autocorrelation coefficients from LPC coefficients

Syntax

```
AC = step(lpc2ac,A)
AC = step(lpc2ac,A,P)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`AC = step(lpc2ac,A)` converts the columns of the linear prediction coefficients, `A`, to autocorrelation coefficients, `AC`. The object assumes a prediction error power of `1`.

`AC = step(lpc2ac,A,P)` when you set the `PredictionErrorInputPort` property to `true`, converts the columns of the linear prediction coefficients, `A`, to autocorrelation coefficients, `AC`. This conversion uses `P` as the prediction error power. `P` must be a row vector with same number of columns as `A`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.LPCToCepstral System object

Package: dsp

Convert linear prediction coefficients to cepstral coefficients

Description

The `LPCToCepstral` object converts linear prediction coefficients to cepstral coefficients.

To convert LPC to cepstral coefficients:

1. Define and set up your LPC to cepstral converter. See "Construction" on page 4-1204.
2. Call step to convert LPC according to the properties of `dsp.LPCToCepstral`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`lpc2cc = dsp.LPCToCepstral` returns an LPC to cepstral converter object, `lpc2cc`, that converts linear prediction coefficients (LPCs) to cepstral coefficients (CCs).

`lpc2cc = dsp.LPCToCepstral('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an LPC to cepstral converter object, `lpc2cc`, with each specified property set to the specified value.

Properties

`PredictionErrorInputPort`

Enable prediction error power input

Choose how to set the prediction error power. When you set this property to `true`, you must specify the prediction error as a second input to the `step` method. When you set this property to `false`, the object assumes the prediction error power is 1. The default is `false`.

CepstrumLengthSource

Source of cepstrum length

Select how to specify the length of cepstral coefficients: `Auto` or `Property`. The default is `Auto`. When this property is set to `Auto`, the length of each channel of the cepstral coefficients output is the same as the length of each channel of the input LPC coefficients. The default is `Property`.

CepstrumLength

Number of output cepstral coefficients

Set the length of the output cepstral coefficients vector as a scalar numeric integer. This property applies when you set the `CepstrumLengthSource on page 4-` property to `Property`. The default is `10`.

NonUnityFirstCoefficientAction

LPC coefficient nonunity action

Specify the action that the object takes when the first coefficient of each channel of the LPC input is not `1`. Select `Replace with 1` or `Normalize`. The default is `Replace with 1`.

Methods

step	Cepstral coefficients from columns of input LPC coefficients

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object

Common to All System Objects	
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Convert LPC To Cepstral Coefficients

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
levinson = dsp.LevinsonSolver;
levinson.AOutputPort = true; % Output polynomial coefficients
ac = dsp.Autocorrelator;
ac.MaximumLagSource = 'Property';
ac.MaximumLag = 9; % Compute autocorrelation lags between [0:9]
lpc2cc = dsp.LPCToCepstral;
x = (1:100)';
a = ac(x);
```

Compute LPC coefficients.

```
A = levinson(a);
```

Convert LPC to CC.

```
CC = lpc2cc(A);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `LPC to/from Cepstral Coefficients` block reference page. The object properties correspond to the block parameters, except:
The object does not have a property that corresponds to the **Type of Conversion** block parameter. The object's behavior corresponds to the block's behavior when you set the **Type of Conversion** parameter to `LPCs to cepstral coefficients`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.LPCToLSF | dsp.CepstralToLPC | dsp.LPCToRC

Introduced in R2012a

step

System object: dsp.LPCToCepstral
Package: dsp

Cepstral coefficients from columns of input LPC coefficients

Syntax

```
CC = step(lpc2cc,A)
CC = step(lpc2cc,A,P)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`CC = step(lpc2cc,A)` computes the cepstral coefficients, `CC`, from the columns of input linear prediction coefficients, `A`. The object assumes the prediction error power is `1`.

`CC = step(lpc2cc,A,P)` when you set the `PredictionErrorInputPort` property to `true`, computes the cepstral coefficients, `CC`, from the columns of input linear prediction coefficients, `A`. This conversion uses `P` as the prediction error power.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.LPCToLSF System object

Package: dsp

Convert linear prediction coefficients to line spectral frequencies

Description

The `LPCToLSF` object converts linear prediction coefficients to line spectral frequencies.

To convert LPC to LSF:

1 Define and set up your LPC to LSF converter. See "Construction" on page 4-1209.
2 Call step to convert LPC according to the properties of `dsp.LPCToLSF`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`lpc2lsf = dsp.LPCToLSF` returns a System object, `lpc2lsf`, that converts linear prediction coefficients (LPCs) to line spectral frequencies (LSFs).

`lpc2lsf = dsp.LPCToLSF('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an LPC to LSF System object, `lpc2lsf`, with each specified property set to the specified value.

Properties

`NumCoarseGridPoints`

Number of coarse subintervals used for finding roots (LSP values)

Specify the number of coarse subintervals, `n`, used for finding line spectral pairs (LSP) values as a positive scalar integer. LSPs, which are the roots of two particular polynomials related to the input LPC polynomial, always lie in the range (–1, 1). The System object finds these roots using the Chebyshev polynomial root finding method. To compute LSF outputs, the object computes the arc cosine of the LSPs, outputting values ranging from 0 to pi radians. The object divides the interval (–1, 1) into n subintervals and looks for roots in each subinterval. If you set *n* too small in relation to the LPC polynomial order, the object can fail to find some of the roots. The default is `64`. This property is tunable.

NumBisects

Value of bisection refinement used for finding roots

Specify the root bisection refinement value, `k`, used in the Chebyshev polynomial root finding method, where each line spectral pair (LSP) output is within

$$\frac{1}{(n \cdot 2^k)}$$

of the actual LSP value. Here *n* is the value of the `NumCoarseGridPoints on page 4-` property, and the object searches a maximum of $k \cdot (n-1)$ points for finding the roots. You must set the `NumBisects` property value *k*, to a positive scalar integer. The default is `4`. This property is tunable.

ExceptionOutputPort

Produces output with validity status of LSF output

Set this property to `true` to return a second output that indicates whether the computed LSF values are valid. The output is a vector with a length equal to the number of channels. A logical value of `1` indicates valid output. A logical value of `0` indicates invalid output. The LSF outputs are invalid when the object fails to find all the LSF values or when the input LPCs are unstable. The default is `false`.

OverwriteInvalidOutput

Enable overwriting invalid output with previous output

Specify the action that the System object should take for invalid LSF outputs. When you set this property to `true`, the object overwrites the invalid output with the previous output. When you set this property to `false`, the object does not take any action on invalid outputs and ignores the outputs.

FirstOutputValuesSource

Source of values for first output when output is invalid

Specify the source of values for the first output when the output is invalid as `Auto` or `Property`. This property applies when you set the `OverwriteInvalidOutput on page 4-` property to `true`. The default is `Auto`. When you set this property to `Auto`, the object uses a default value for the first output. The default value corresponds to the LSF representation of an allpass filter.

FirstOutputValues

Value of the first output

Specify a numeric vector of LSF values for overwriting an invalid first output. The length of this vector must be one less than the length of the input LPC vector. For multichannel inputs, you can set this property to a matrix with the same number of channels as the input, or one vector that is applied to every channel. The default is an empty vector. This property applies when you set the `OverwriteInvalidOutput on page 4-` property to `true` and the `FirstOutputValuesSource on page 4-` property to `Property`.

NonUnityFirstCoefficientAction

Action to take when first LPC coefficient is not `1`

Specify the action the object takes when the first coefficient of each channel of the LPC input is not `1` as `Replace with 1` or `Normalize`. The default is `Replace with 1`.

Methods

reset	Reset values for overwriting invalid outputs to their initial values
step	Convert LPC coefficients to line spectral frequencies

Common to All System Objects	
`clone`	Create System object with same property values

Common to All System Objects	
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Convert LPC Coefficients To LSF Coefficients

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Convert LPC to LSF coefficients

```
a = [1.0000 0.6149 0.9899 0.0000 0.0031 -0.0082]';
lpc2lsf = dsp.LPCToLSF;
y = lpc2lsf(a);
display(y);
```

```
y =

    0.7842
    1.5605
    1.8776
    1.8984
    2.3593
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `LPC to LSF/LSP Conversion` block reference page. The object properties correspond to the block parameters, except:
There is no object property that corresponds to the **Output** block parameter. The object only supports LSF outputs in the range $(0, \Pi)$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.LPCToLSP | dsp.LSFToLPC

Introduced in R2012a

reset

Reset values for overwriting invalid outputs to their initial values

Syntax

```
reset(lpc2lsf)
```

Description

`reset(lpc2lsf)` resets the values for overwriting the invalid outputs to their initial values.

step

System object: dsp.LPCToLSF
Package: dsp

Convert LPC coefficients to line spectral frequencies

Syntax

```
LSF = step(lpc2lsf,A)
[..., STATUS] = step(lpc2lsf,A)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`LSF = step(lpc2lsf,A)` converts the `LPC` coefficients, `A` , to line spectral frequencies, `LSF`, in the range (0 pi). The System object operates along the columns of the input `A`.

`[..., STATUS] = step(lpc2lsf,A)` also returns the status flag, `STATUS`, indicating if the current output is valid when the `ExceptionOutputPort` property is `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.LPCToLSP System object

Package: dsp

Convert linear prediction coefficients to line spectral pairs

Description

The `LPCToLSP` object converts linear prediction coefficients to line spectral pairs.

To convert LPC to LSP:

1. Define and set up your LPC to LSP converter. See "Construction" on page 4-1216.
2. Call step to convert LPC according to the properties of `dsp.LPCToLSP`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`lpc2lsp = dsp.LPCToLSP` returns a System object, `lpc2lsp`, that converts linear prediction coefficients (LPCs) to line spectral pairs (LSPs).

`lpc2lsp = dsp.LPCToLSP('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an `LPC` to `LSF` System object, `lpc2lsp`, with each specified property set to the specified value.

Properties

`NumCoarseGridPoints`

Number of coarse subintervals used for finding roots (LSP values)

Specify the number of coarse subintervals, *n*, used for finding line spectral pairs (LSP) values, as a positive scalar integer. LSPs, which are the roots of two particular polynomials related to the input LPC polynomial, always lie in the range (–1, 1). The System object finds these roots using the Chebyshev polynomial root finding method. The object divides the interval (–1, 1) into *n* subintervals and looks for roots in each subinterval. If *n* is set to too small a number in relation to the LPC polynomial order, the object can fail to find some of the roots. The default is `64`. This property is tunable.

`NumBisects`

Value of bisection refinement used for finding roots

Specify the root bisection refinement value, *k*, that the Chebyshev polynomial uses in the root finding method. For each line spectral pair (LSP) the output is within

$$\frac{1}{\mathrm{n} \cdot 2^{\mathrm{k}}}$$

of the actual LSP value. Here *n* is the value of the `NumCoarseGridPoints` on page 4- property and the object searches a maximum of

$$\mathrm{k} \cdot (\mathrm{n}\text{-}1)$$

points for finding the roots. The `NumBisects` on page 4- property value *k*, must be a positive scalar integer. The default is `4`. This property is tunable.

`ExceptionOutputPort`

Produces output with validity status of LSP output

Set this property to `true` to return a second output that indicates whether the computed LSP values are valid. The object outputs a vector length equal to the number of channels. A logical value of `1` indicates the output is valid. A logical value of `0` indicates the output is invalid. The LSP outputs are invalid when the object fails to find all the LSP values or when the input LPCs are unstable. The default is `false`.

`OverwriteInvalidOutput`

Enable overwriting invalid output with previous output

Specify the action that the object takes for invalid LSP outputs. When you set this property to `true`, the object overwrites the invalid output with the previous output.

When you set this property to `false`, the object takes no action on invalid outputs and ignores the outputs.

`FirstOutputValuesSource`

Source of values for first output when output is invalid

Specify the source of values for the first output when the output is invalid as `Auto` or `Property`. This property applies only when you set the `OverwriteInvalidOutput on page 4-` property to `true`. The default is `Auto`. When this property is `Auto`, the object uses a default value for the first output. The default value corresponds to the LSP representation of an allpass filter.

`FirstOutputValues`

Value of first output

Specify a numeric vector of LSP values for overwriting an invalid first output. The length of this vector must be one less than the length of the input LPC vector. For multichannel inputs, set this property can to a matrix with the same number of channels as the input or one vector that you apply to every channel. The default is an empty vector. This property applies only when you set the `OverwriteInvalidOutput on page 4-` property to `true` and the `FirstOutputValuesSource on page 4-` property to `Property`.

`NonUnityFirstCoefficientAction`

First coefficient nonunity action

Specify the action that the object takes when the first coefficient of each channel of the LPC input is not equal to `1`. Specify as one of `Replace with 1` or `Normalize`. The default is `Replace with 1`.

Methods

reset	Reset values for overwriting invalid outputs to their initial values
step	Convert linear prediction coefficients to line spectral pairs

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Convert LPC To LSP Coefficients

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Convert the linear prediction coefficients to line spectral pairs.

```
a = [1.0000 0.6149 0.9899 0.0000 0.0031 -0.0082]';
lpc2lsp = dsp.LPCToLSP;
y = lpc2lsp(a); % Convert to LSP coefficients
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `LPC to LSF/LSP Conversion` block reference page. The object properties correspond to the block parameters, except:
No object property corresponds to the **Output** block parameter. The object only supports LSP outputs.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.LPCToLSF | dsp.LSPToLPC

Introduced in R2012a

reset

Reset values for overwriting invalid outputs to their initial values

Syntax

```
reset(H)
```

Description

`reset(H)` resets the values for overwriting the invalid outputs to their initial values.

step

System object: dsp.LPCToLSP
Package: dsp

Convert linear prediction coefficients to line spectral pairs

Syntax

```
LSF = step(lpc2lsp,A)
[..., STATUS] = step(lpc2lsp,A)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`LSF = step(lpc2lsp,A)` converts the `LPC` coefficients, `A` , to line spectral pairs normalized in the range (–1 1), `LSP`. The object operates along the columns of the input `A`.

`[..., STATUS] = step(lpc2lsp,A)` also returns the status flag, `STATUS`, indicating if the current output is valid when the `ExceptionOutputPort` property is `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.LPCToRC System object

Package: dsp

Convert linear prediction coefficients to reflection coefficients

Description

The `LPCToRC` object converts linear prediction coefficients to reflection coefficients.

To convert LPC to reflection coefficients:

1. Define and set up your LPC to RC converter. See "Construction" on page 4-1223.
2. Call step to convert LPC according to the properties of `dsp.LPCToRC`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`lpc2rc = dsp.LPCToRC` returns an LPC to RC System object, `lpc2rc`, that converts linear prediction coefficients (LPC) to reflection coefficients (RC).

`lpc2rc = dsp.LPCToRC('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an LPC to RC conversion object, `lpc2rc`, with each specified property set to the specified value.

Properties

`PredictionErrorOutputPort`

Enable normalized prediction error power output

Set this property to `true` to return the normalized error power as a vector with one element per input channel. Each element varies between zero and one. The default is `true`.

ExceptionOutputPort

Produces output with stability status of filter represented by LPC coefficients

Set this property to `true` to return the stability status of the filter. A logical value of `1` indicate a stable filter. A logical value of `0` indicate an unstable filter. The default is `false`.

NonUnityFirstCoefficientAction

Action to take when first LPC coefficient is not 1

Specify the action that the object takes when the first coefficient of each channel of the LPC input is not 1. Select `Replace with 1` or `Normalize`. The default is `Replace with 1`.

Methods

step	Convert columns of linear prediction coefficients to reflection coefficients

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Convert LPC Coefficients To RC Coefficients

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Convert the linear prediction coefficients to reflection coefficients:

```
load mtlb

levinson = dsp.LevinsonSolver;
levinson.AOutputPort = true;
levinson.KOutputPort = false;

ac = dsp.Autocorrelator;
lpc2rc = dsp.LPCToRC;
ac.MaximumLagSource = 'Property';
```

Compute autocorrelation for lags between [0:10]

```
ac.MaximumLag = 10;
a = ac(mtlb);
A = levinson(a); % Compute LPC coefficients
[K, P] = lpc2rc(A); % Convert to RC
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `LPC to/from RC` block reference page. The object properties correspond to the block parameters, except:

- There is no object property that corresponds to the **Type of conversion** block parameter. The object always converts LPC to RC.
- The `NonUnityFirstCoefficientAction` object property corresponds to the **If first input value is not 1** block parameter. There is neither a `Normalize and warn` nor an `Error` option for the object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.RCToLPC | dsp.LPCToAutocorrelation

Introduced in R2012a

step

System object: dsp.LPCToRC
Package: dsp

Convert columns of linear prediction coefficients to reflection coefficients

Syntax

```
[K,P] = step(lpc2rc,A)
K = step(lpc2rc,A)
[..., S] = step(lpc2rc,A)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[K,P] = step(lpc2rc,A)` converts the columns of linear prediction coefficients, `A`, to reflection coefficients `K` and outputs the normalized prediction error power, `P`.

`K = step(lpc2rc,A)` when you set the `PredictionErrorOutputPort` property to `false`, converts the columns of linear prediction coefficients, `A`, to reflection coefficients `K`.

`[..., S] = step(lpc2rc,A)` also outputs the `LPC` filter stability, `S`, when you set the `ExceptionOutputPort` property to `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.LSFToLPC System object

Package: dsp

Convert line spectral frequencies to linear prediction coefficients

Description

The `LSFToLPC` object converts line spectral frequencies to linear prediction coefficients.

To convert LSF to LPC:

1. Define and set up your LSF to LPC converter. See "Construction" on page 4-1229.
2. Call step to convert LSF according to the properties of `dsp.LSFToLPC`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`lsf2lpc = dsp.LSFToLPC` returns an LSF to LPC System object, `lsf2lpc`, which converts line spectral frequencies (LSFs) to linear prediction coefficients (LPCs).

Methods

step	Convert input line spectral frequencies to linear prediction coefficients

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object

Common to All System Objects	
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Convert LSF Coefficients to LPC Coefficients

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
a = [1.0000 0.6149 0.9899 0.0000 0.0031 -0.0082]'
lpc2lsf = dsp.LPCToLSF;
ylsf = lpc2lsf(a);
lsf2lpc = dsp.LSFToLPC;
ylpc = lsf2lpc(ylsf);
```

```
a =

    1.0000
    0.6149
    0.9899
         0
    0.0031
   -0.0082
```

Check if values in `ylpc` are the same as in `a`.

```
display(ylpc);
```

```
ylpc =

    1.0000
    0.6149
    0.9899
    0.0000
```

```
  0.0031
 -0.0082
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `LSF/LSP to LPC Conversion` block reference page. The object properties correspond to the block parameters, except:
The object does not have a property that corresponds to the **Input** block parameter. The object's behavior corresponds to the block's behavior when you set the **Input** parameter to `LSF in range (0 pi)`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.LSPToLPC | dsp.LPCToLSF

Introduced in R2012a

step

System object: dsp.LSFToLPC
Package: dsp

Convert input line spectral frequencies to linear prediction coefficients

Syntax

```
A = step(lsf2lpc,LSF)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`A = step(lsf2lpc,LSF)` converts the input line spectral frequencies, (LSF), in the range `(0,pi)`, `LSF` , to linear prediction coefficients, `A`. The input can be a vector or a matrix, where each column of the matrix is treated as a separate channel.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.LSPToLPC System object

Package: dsp

Convert line spectral pairs to linear prediction coefficients

Description

The `LSPToLPC` object converts line spectral pairs to linear prediction coefficients.

To convert LSP to LPC:

1 Define and set up your LSP to LPC converter. See "Construction" on page 4-1233.
2 Call step to convert LSP according to the properties of `dsp.LSPToLPC`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = dsp.LSPToLPC` returns an LSP to LPC System object, `H`, which converts line spectral pairs (LSPs) to linear prediction coefficients (LPCs).

Methods

step	Convert input line spectral pairs to linear prediction coefficients

Common to All System Objects	
`clone`	Create System object with same property values

Common to All System Objects	
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Convert LSP To LPC Coefficients

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Convert line spectral pairs to linear prediction coefficients.

```
ylsp = [0.7080 0.0103 -0.3021 -0.3218 -0.7093]';
lsp2lpc = dsp.LSPToLPC;
ylpc = lsp2lpc(ylsp)
```

```
ylpc =

    1.0000
    0.6149
    0.9898
   -0.0000
    0.0030
   -0.0081
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `LSF/LSP to LPC Conversion` block reference page. The object properties correspond to the block parameters, except:
No object property corresponds to the **Input** block parameter. The object converts LSP in the range (–1, 1) to LPC.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also
dsp.LSFToLPC | dsp.LPCToLSP

Introduced in R2012a

step

System object: dsp.LSPToLPC
Package: dsp

Convert input line spectral pairs to linear prediction coefficients

Syntax

`A = step(lsp2lpc,LSP)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`A = step(lsp2lpc,LSP)` converts the input line spectral pairs in the range (–1,1), `LSP`, to linear prediction coefficients, `A`. The input can be a vector or a matrix, where each column of the matrix is treated as a separate channel.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.LUFactor System object

Package: dsp

Factor square matrix into lower and upper triangular matrices

Description

The `LUFactor` object factors a square matrix into lower and upper triangular matrices.

To factor a square matrix into lower and upper triangular matrices:

1 Define and set up your System object. See "Construction" on page 4-1237.
2 Call step to factor the square matrix according to the properties of `dsp.LUFactor`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`lu = dsp.LUFactor` returns an `LUFactor` System object, `lu`, which factors a row permutation of a square input matrix `A` as $A_p = L \cdot U$, where `L` is the unit-lower triangular matrix, and `U` is the upper triangular matrix. The row-pivoted matrix A_p contains the rows of A permuted as indicated by the permutation index vector P. The equivalent MATLAB code is `Ap = A(P,:)`.

`lu = dsp.LUFactor('PropertyName',PropertyValue,...)` returns an `LUFactor` object, `lu`, with each specified property set to the specified value.

Properties

ExceptionOutputPort

Set to `true` to output singularity of input

Set this property to `true` to output the singularity of the input as logical data type values of `true` or `false`. An output of `true` indicates that the current input is singular, and an output of `false` indicates the current input is nonsingular.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `|Ceiling|Convergent|Floor|Nearest |Round | Simplest | Zero|`. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as `Full precision`, `Same as input` or `Custom`. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType on page 4-1238` property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Full precision`, `Same as input`, `Same as product` or `Custom`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType on page 4-1238` property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType on page 4-1239` property to `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Decompose matrix into lower and upper triangular matrices

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Decompose a Square Matrix

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Decompose a square matrix into the lower and upper components.

```
lu = dsp.LUFactor;
x = rand(4)
[LU, P] = lu(x);
L = tril(LU,-1)+diag(ones(size(LU,1),1));
U = triu(LU);
y = L*U
```

```
x =

    0.8147    0.6324    0.9575    0.9572
    0.9058    0.0975    0.9649    0.4854
    0.1270    0.2785    0.1576    0.8003
    0.9134    0.5469    0.9706    0.1419


y =

    0.9134    0.5469    0.9706    0.1419
    0.9058    0.0975    0.9649    0.4854
    0.8147    0.6324    0.9575    0.9572
    0.1270    0.2785    0.1576    0.8003
```

Check back whether `y` equals the permuted `x`

```
xp = x(P,:)
```

```
xp =

    0.9134    0.5469    0.9706    0.1419
    0.9058    0.0975    0.9649    0.4854
    0.8147    0.6324    0.9575    0.9572
```

```
    0.1270    0.2785    0.1576    0.8003
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `LU Factorization` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

dsp.LDLFactor

Introduced in R2012a

step

System object: dsp.LUFactor
Package: dsp

Decompose matrix into lower and upper triangular matrices

Syntax

```
[LU,P] = step(lu,A)
[LU,P,S] = step(lu,A)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[LU,P] = step(lu,A)` decomposes the matrix `A` into lower and upper triangular matrices. The output `LU` is a composite matrix with lower triangle elements from `L` and upper triangle elements from `U` . The permutation vector `P` is the second output.

`[LU,P,S] = step(lu,A)` returns an additional output `S` indicating if the input is singular when the `ExceptionOutputPort` property is set to `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.MatFileReader System object

Package: dsp

Read MAT file

Description

The `MatFileReader` object reads V7.3 MAT files.

To read V7.3 MAT files:

1. Define and set up your System object. See "Construction" on page 4-1243.
2. Call step to read the MAT file according to the properties of `dsp.MatFileReader`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`mfr = dsp.MatFileReader` returns a System object, `mfr`, to read scalar stream data from a V7.3 MAT file.

`mfr = dsp.MatFileReader(`*`FILENAME`*`,`*`VARIABLENAME`*`,`*`FRAMESIZE`*`)` reads frames of MAT file data, using the specified file name, variable name, and frame size.

`mfr = dsp.MatFileReader('`*`PropertyName`*`',`*`PropertyValue`*`,...)` reads MAT file data with each specified property set to the specified value.

`mfr =`
`dsp.MatFileReader(`*`FILENAME`*`,`*`VARIABLENAME`*`,`*`FRAMESIZE`*`,'`*`PropertyName`*`',`*`PropertyValue`*
reads frames of MAT file data, using the specified file name, variable name, and frame size, and other specified properties set to the specified values.

Properties

`Filename`

Name of MAT file from which to read

Specify the name of a MAT file as a character vector. Specify the full path for the file only if the file is not on the MATLAB path.

Default: `Untitled.mat`

`VariableName`

Name of the variable to read

Name of the variable stored in and read from the MAT file.

Default: `x`

`SamplesPerFrame`

Number of samples per output frame

Specify the number of elements (samples per frame) to read from the MAT file each time the `step` method is called.

Default: 1

Methods

isDone	End-of-file status
reset	Reset internal states of multimedia file reader to read from beginning of file
step	Read data from a variable in the MAT file

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object

Common to All System Objects	
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Read a MAT File

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Read a MAT file using the `MatFileReader` object.

```
filename = [tempname '.mat'];  % Create variable name
originalData = rand(40,2);
save(filename,'originalData','-v7.3'); % Write to MAT file

mfr = dsp.MatFileReader(filename,'VariableName',...
   'originalData','SamplesPerFrame', 4);
while ~isDone(mfr)          % Stream data into MATLAB
    finalData = mfr();
end
```

See Also

dsp.MatFileWriter

isDone

System object: dsp.MatFileReader
Package: dsp

End-of-file status

Syntax

```
STATUS = isDone(mfr)
```

Description

`STATUS = isDone(mfr)` returns a logical value, `STATUS`. When the `MatFileReader` object, `mfr` , reaches the end of the MAT file, `STATUS` is `true`.

reset

System object: dsp.MatFileReader
Package: dsp

Reset internal states of multimedia file reader to read from beginning of file

Syntax

`reset(mfr)`

Description

`reset(mfr)` resets the `MatFileReader` object to read from the beginning of the file.

step

System object: dsp.MatFileReader
Package: dsp

Read data from a variable in the MAT file

Syntax

```
X = step(mfr)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`X = step(mfr)` reads data, `X`, from a variable stored in a MAT-file. The variable is assumed to be *N*-dimensional and a MATLAB built-in datatype. The data is read into MATLAB by reading along the first dimension.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.MatFileWriter System object

Package: dsp

Write MAT file

Description

The `MatFileWriter` object writes data to a V7.3 MAT file.

To write data to a V7.3 MAT file:

1. Define and set up your System object. See “Construction” on page 4-1249.
2. Call step to write data according to the properties of `dsp.MatFileWriter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`mfw = dsp.MatFileWriter` returns a MAT file writer System object, `mfw`, that writes data to a V7.3 MAT file.

`mfw = dsp.MatFileWriter('PropertyName',PropertyValue,...)` returns a MAT file writer System object, `mfw`, with each specified property set to the specified value.

`mfw = dsp.MatFileWriter(FILENAME,'PropertyName',PropertyValue,...)` returns a MAT file writer System object, `mfw`, with `Filename` property set to `FILENAME` and other specified properties set to the specified values.

Properties

`Filename`

Name of MAT file to write

Specify the name of a MAT file as a character vector. Specify the full path for the file only if the file is not on the MATLAB path. The default file name is `Untitled.mat`.

`VariableName`

Name of the variable to write

Name of the variable to which to write. This variable is stored in the MAT file. The default variable name is `x`. You cannot overwrite a variable that is already in an existing MAT file.

Methods

step	Write one frame of MAT file data

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Write Data Into a MAT File

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

First, create a variable name.

```
filename = [tempname '.mat'];
```

Next, write that variable to a MAT-file.

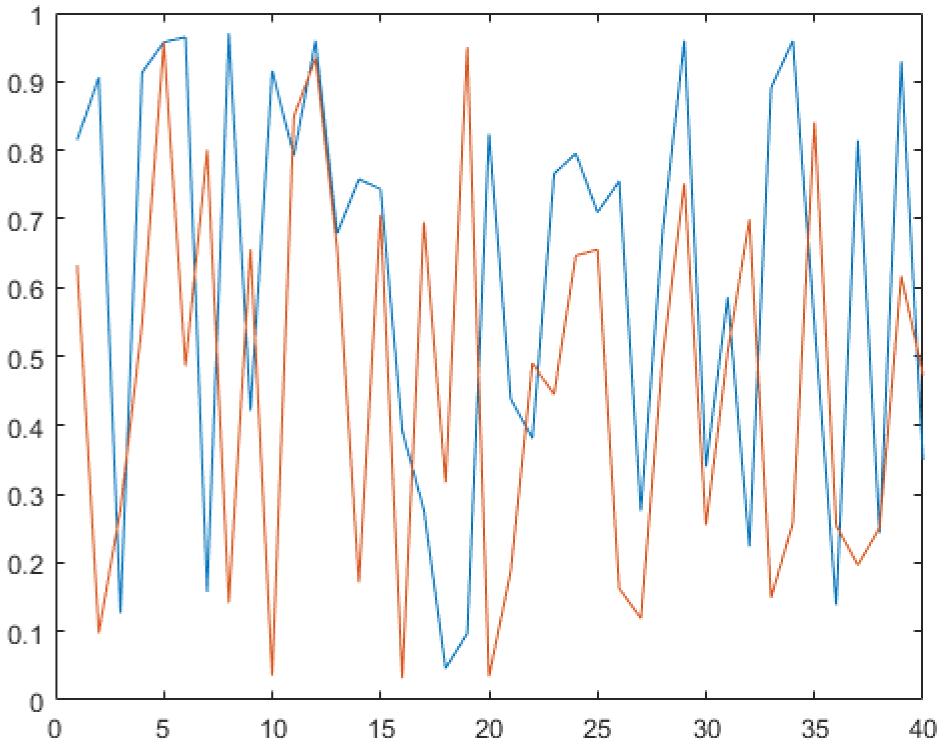
```
mfw = dsp.MatFileWriter(filename, 'VariableName', 'originalData');
for i = 1:10
    originalData = rand(4,2);
    mfw(originalData);
end
release(mfw); % This will close the MAT file
```

Finally, load the variable back into MATLAB.

```
data = load(filename, 'originalData');
```

Plot the data.

```
plot(data.originalData);
```



See Also

dsp.MatFileReader

Introduced in R2012b

step

System object: dsp.MatFileWriter
Package: dsp

Write one frame of MAT file data

Syntax

`step(mfw,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`step(mfw,X)` writes one frame of data, `X` , to the variable stored in the MAT file. The variable is assumed to be N-dimensional and a MATLAB built-in data type. The data is written to the file by concatenating along the first dimension.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.Maximum System object

Package: dsp

Find maximum value of input or sequence of inputs

Description

The `dsp.Maximum` object finds the maximum values of an input or sequence of inputs.

To compute the maximum value of an input or sequence of inputs:

1. Define and set up your System object. See "Construction" on page 4-1254.
2. Call step to find the maximum according to the properties of `dsp.Maximum`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Note: The `Running` mode in the `dsp.Maximum` System object will be removed in a future release. To compute the running maximum in MATLAB, use the dsp.MovingMaximum System object instead.

Construction

`max = dsp.Maximum` returns an object, `max`, that computes the value and index of the maximum elements in an input or a sequence of inputs along the specified `Dimension` on page 4-1256.

`max = dsp.Maximum('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a maximum-finding object, `max`, with each specified property set to the specified value.

Properties

ValueOutputPort

Output maximum value

Set this property to `true` in order to output the maximum of the input. This property applies only when you set the `RunningMaximum` on page 4-1255 property to false. The default is `true`.

RunningMaximum

Calculate over single input or multiple inputs

When you set this property to `true`, the object computes the maximum value over successive calls to the `step` method. When you set this property to `false`, the object computes the maximum value over the current input. The default is `false`.

IndexOutputPort

Output index of maximum value

Set this property to `true` to output the index of the maximum value of the input. This property applies only when you set the `RunningMaximum` on page 4-1255 property to `false`. The default is `true`.

ResetInputPort

Additional input to enable resetting of running maximum

Set this property to `true` to enable resetting the running maximum. When you set this property to `true`, you must specify a reset input to the `step` method to reset the running maximum. This property applies only when you set the `RunningMaximum` on page 4-1255 property to `true`. The default is `false`.

ResetCondition

Condition that triggers resetting of running maximum

Specify the event that resets the running maximum as one of `| Rising edge | Falling edge | Either edge | Non-zero |`. This property applies only when you set the `ResetInputPort` on page 4-1255 property to `true`. The default is `Non-zero`.

IndexBase

Numbering base for index of maximum value

Specify whether to start the index numbering from `One` or `Zero` when computing the index of the maximum value. This property applies only when you set the `IndexOutputPort` on page 4-1255 property to `true`. The default is `One`.

Dimension

Dimension to operate along

Specify how the maximum calculation is performed over the data as one of | `All` | `Row` | `Column` | `Custom` |. This property applies when you set the `RunningMaximum` on page 4-1255 property to `false`. The default is `Column`.

CustomDimension

Numerical dimension to calculate over

Specify the integer dimension of the input signal over which the object finds the maximum. The cannot exceed the number of dimensions in the input signal. This property only applies when you set the `Dimension` on page 4-1256 property to `Custom`. The default is `1`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero` |. The default is `Floor`.

OverflowAction

Action to take when integer input is out-of-range

Specify the overflow action as one of | `Wrap` | `Saturate` |. The default is `Wrap`.

ProductDataType

Data type of product

Specify the product fixed-point data type as one of | `Same as input` | `Custom` |. The default is `Same as input`.

`CustomProductDataType`

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `AccumulatorDataType` on page 4-1257 property to `Custom`. The default is `numerictype([],32,30)`.

`AccumulatorDataType`

Data type of accumulator

Specify the accumulator fixed-point data type as one of | `Same as product` | `Same as input` | `Custom` |. The default is `Same as product`.

`CustomAccumulatorDataType`

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `AccumulatorDataType` on page 4-1257 property to `Custom`. The default is `numerictype([],32,30)`.

Methods

reset	Reset computation of running maximum
step	Maximum value

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object

Common to All System Objects	
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Compute the Maximum and Running Maximum of a Signal

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

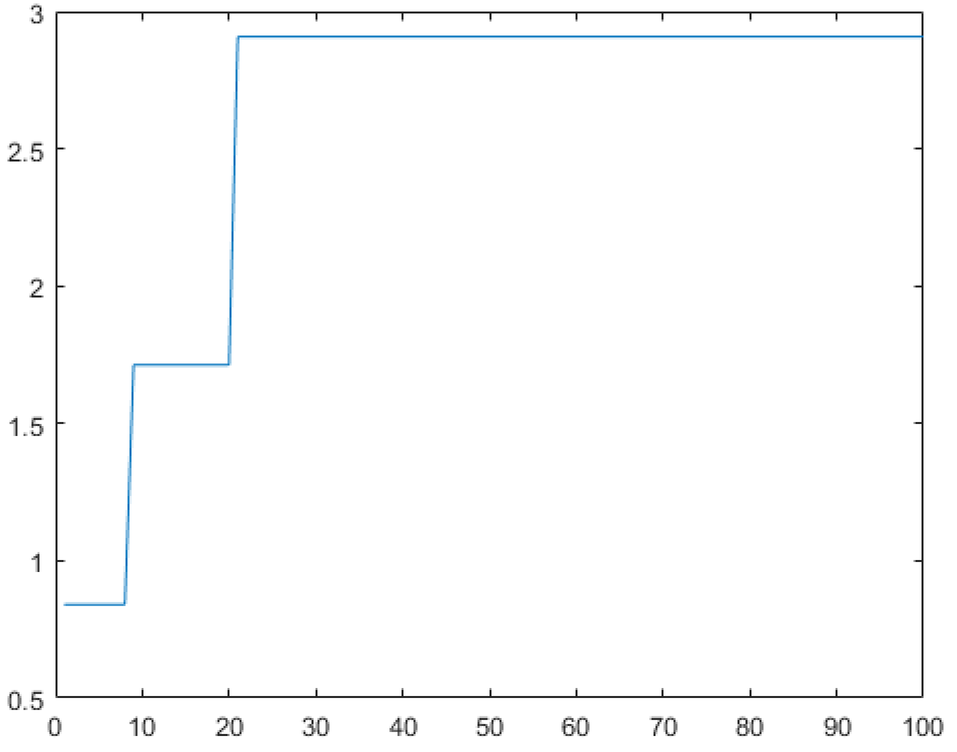
Find a maximum value and its index.

```
max1 = dsp.Maximum;
x = randn(100,1);
[y, I] = max1(x)
```

```
y =

    3.5784


I =

     9
```

Compute a running maximum.

```
max2 = dsp.Maximum;
max2.RunningMaximum = true;
x = randn(100,1);
z = max2(x);
plot(z);
```



y(i) is the minimum of all values in the vector x(1:i).

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Maximum` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

System Objects
dsp.MovingMaximum | dsp.Mean | dsp.Minimum

Blocks
`Maximum` | `Moving Maximum`

Introduced in R2012a

reset

System object: dsp.Maximum
Package: dsp

Reset computation of running maximum

Syntax

```
reset(max)
```

Description

`reset(max)` resets the computation of the running maximum for the `Maximum` object `max`.

step

System object: dsp.Maximum
Package: dsp

Maximum value

Syntax

```
[VAL,IND] = step(max,X)
VAL = step(max,X)
IND = step(max,X)
VAL = step(max,X,R)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[VAL,IND] = step(max,X)` returns the maximum value, `VAL`, and the index or position of the maximum value, `IND`, along the specified `Dimension` of `X`.

`VAL = step(max,X)` returns the maximum value, `VAL`, of the input `X`. When the `RunningMaximum` property is `true`, `VAL` corresponds to the maximum value over successive calls to the `step` method.

`IND = step(max,X)` returns the zero- or one-based index `IND` of the maximum value. To enable this type of processing, set the `IndexOutputPort` property to `true` and the `ValueOutputPort` and `RunningMaximum` properties to `false`.

`VAL = step(max,X,R)` resets the state of `max` based on the value of reset signal, `R`, and the `ResetCondition` property. To enable this type of processing, set the `RunningMaximum` property to `true` and the `ResetInputPort` property to `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.Mean System object

Package: dsp

Find mean value of input or sequence of inputs

Description

The `dsp.Mean` object finds the mean of an input or sequence of inputs.

To compute the mean of an input or sequence of inputs:

1 Define and set up your System object. See "Construction" on page 4-1264.
2 Call step to compute the mean according to the properties of `dsp.Mean`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Note: The `Running` mode in the `dsp.Mean` System object will be removed in a future release. To compute the running mean in MATLAB, use the dsp.MovingAverage System object instead.

Construction

`mn = dsp.Mean` returns an object, `mn`, that computes the mean of an input or a sequence of inputs.

`mn = dsp.Mean('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a mean-finding object, `mn`, with each specified property set to the specified value.

Properties

RunningMean

Calculate over single input or multiple inputs

When you set this property to `true`, the object calculates the mean over successive calls to the `step` method. When you set this property to `false`, the object computes the mean over the current input. The default is `false`.

ResetInputPort

Additional input to enable resetting of running mean

Set this property to `true` to enable resetting of the running mean. When you set this property to `true`, you must specify a reset input to the `step` method to reset the running mean. This property applies only when you set the `RunningMean` on page 4-1265 property to `true`. The default is `false`.

ResetCondition

Condition that triggers resetting of running mean

Specify the event that resets the running maximum as one of | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero` |. This property applies only when you set the `ResetInputPort` on page 4-1265 property to `true`. The default is `Non-zero`.

Dimension

Dimension to operate along

Specify how the mean calculation is performed over the data as one of | `All` | `Row` | `Column` | `Custom` |. This property applies when you set the `RunningMean` on page 4-1265 property to `false`. The default is `Column`.

CustomDimension

Numerical dimension to calculate over

Specify the integer dimension, indexed from one, of the input signal over which the object calculates the mean. The value cannot exceed the number of dimensions in the input signal. This property only applies when you set the `Dimension` on page 4-1265 property to `Custom`. The default is `1`.

Fixed-Point Properties

`RoundingMethod`

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero` |. The default is `Floor`.

`OverflowAction`

Action to take when integer input is out-of-range

Specify the overflow action as one of | `Wrap` | `Saturate` |. The default is `Wrap`.

`AccumulatorDataType`

Data type of accumulator

Specify the accumulator fixed-point data type as one of | `Same as input` | `Custom` |. The default is `Same as input`.

`CustomAccumulatorDataType`

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `AccumulatorDataType` on page 4-1266 property to `Custom`. The default is `numerictype([],32,30)`.

`OutputDataType`

Data type of output

Specify the output fixed-point data type as one of | `Same as accumulator` | `Same as input` | `Custom` |. The default is `Same as accumulator`.

`CustomOutputDataType`

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` on page 4-1266 property to `Custom`. The default is `numerictype([],32,30)`.

Methods

reset	Reset internal states of mean-finding object
step	Mean value

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Compute the Mean and Running Mean of a Signal

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Mean

```
mean1 = dsp.Mean;
x = randn(100,1);
y = mean1(x);
```

Running Mean

```
mean2 = dsp.Mean;
mean2.RunningMean = true;
x = randn(100,1);
```

```
yrmean = mean2(x);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Mean` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

System Objects
dsp.MovingAverage | dsp.Minimum

Blocks
`Mean` | `Moving Average`

Introduced in R2012a

reset

System object: dsp.Mean
Package: dsp

Reset internal states of mean-finding object

Syntax

```
reset(mn)
```

Description

`reset(mn)` sets the internal states of the `Mean` object `mn` to their initial values.

step

System object: dsp.Mean
Package: dsp

Mean value

Syntax

```
Y = step(mn,X)
Y = step(mn,X,R)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(mn,X)` computes the mean of `X`. When you set the `RunningMean` property to `true`, `Y` corresponds to the mean successive calls to the `step` method.

`Y = step(mn,X,R)` resets the computation of the running mean based on the value of the reset signal, `R`, and the `ResetCondition` property. To enable this type of processing, set the `RunningMean` property to `true` and the `ResetInputPort` property to `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.Median System object

Package: dsp

Median value of input

Description

The `Median` object computes the median value of the input. The object can compute the median along each dimension (row or column) of the input or of the entire input.

To compute the median of the input:

1. Define and set up your median System object. See "Construction" on page 4-1271.
2. Call step to compute the median according to the properties of `dsp.Median`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`med = dsp.Median` returns a median System object, `med`, that computes the median along the columns of the input using the `Quick sort` sorting method.

`med = dsp.Median('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a median System object, `med`, with each property set to the value you specify.

Properties

`SortMethod`

Sort method

Specify the method the object should use to sort the data before computing the median. You can specify `Quick sort` or `Insertion sort`. The quick sort algorithm uses a recursive sort method and is faster at sorting more than 32 elements. The insertion sort algorithm uses a nonrecursive method and is faster at sorting less than 32 elements. If you are using the `Median` object to generate code, you should use the insertion sort algorithm to prevent recursive function calls in your generated code. The default is `Quick sort`.

Dimension

Dimension to operate along

Specify the dimension along which the object computes the median values. You can specify one of | `All` | `Row` | `Column` | `Custom` |. The default is `Column`.

CustomDimension

Numerical dimension to operate along

Specify the dimension of the input signal (as a one-based value), over which the object computes the median. The cannot exceed the number of dimensions in the input signal. This property applies only when you set the `Dimension` on page 4-1272 property to `Custom`. The default is `1`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero` |. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | `Wrap` | `Saturate` |. The default is `Wrap`.

ProductDataType

Product word and fraction lengths

Specify the product data type as one of | `Same as input` | `Custom` |. The default is `Same as input`.

`CustomProductDataType`

Product word and fraction lengths

Specify the product data type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `ProductDataType` on page 4-1272 property to `Custom`. The default is `numerictype([],32,30)`.

`AccumulatorDataType`

Accumulator word and fraction lengths

Specify the accumulator data type as one of | `Same as product` | `Same as input` | `Custom` |. The default is `Same as product`.

`CustomAccumulatorDataType`

Accumulator word and fraction lengths

Specify the fixed-point accumulator data type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `AccumulatorDataType` on page 4-1273 property to `Custom`. The default is `numerictype([],32,30)`.

`OutputDataType`

Output word and fraction lengths

Specify the output data type as one of | `Same as accumulator` | `Same as product` | `Same as input` | `Custom` |. The default is `Same as accumulator`.

`CustomOutputDataType`

Output word and fraction lengths

Specify the data type of the output as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` on page 4-1273 property to `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Median value

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Compute The Median

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Compute the median value of the input column using the `dsp.Median` object.

```
med = dsp.Median;
x = [7 -9 0 -1 2 0 3 5 -9]';
y = med(x)
```

```
y =

     0
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Median` block reference page. The object properties correspond to the block properties, except:
Treat sample-based row input as a column block parameter is not supported by the `dsp.Median` System object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.Minimum | dsp.Variance | dsp.Mean | dsp.Maximum

Introduced in R2012a

step

System object: dsp.Median
Package: dsp

Median value

Syntax

`Y = step(med,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(med,X)` computes the median value of the input `X` and returns the result in `Y`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.MedianFilter System object

Package: dsp

Median filter

Description

The `dsp.MedianFilter` System object computes the moving median of the input signal along each channel, independently over time. The object uses the sliding window method to compute the moving median. In this method, a window of specified length is moved over each channel, sample by sample, and the object computes the median of the data in the window. For more details, see "Algorithms" on page 4-1284.

The object accepts multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$, and $n > 1$. m is the number of samples in each frame (or channel), and n is the number of channels. The object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel. However, the number of channels cannot change. This object supports C and C++ code generation.

To compute the moving median of the input:

1 Create a `dsp.MedianFilter` object and set the properties of the object.
2 Call step to compute the moving median.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`medFilt = dsp.MedianFilter` returns a median filter object, `medFilt`, using the default properties.

`medFilt = dsp.MedianFilter(Len)` sets the `WindowLength` property to `Len`.

`medFilt = dsp.MedianFilter(Name,Value)` specifies properties using `Name,Value` pairs. Unspecified properties have default values.

Example:

```
movMin = dsp.MedianFilter('WindowLength',5);
```

Properties

`WindowLength` — Length of the sliding window

5 (default) | positive scalar integer

Length of the sliding window, specified as a positive scalar integer.

Methods

reset	Reset internal states of System object
step	Moving median of input signal

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Remove High-Frequency Noise Using Median Filter

Filter high-frequency noise from a noisy sine wave signal using a median filter. Compare the performance of the median filter with an averaging filter.

Initialization

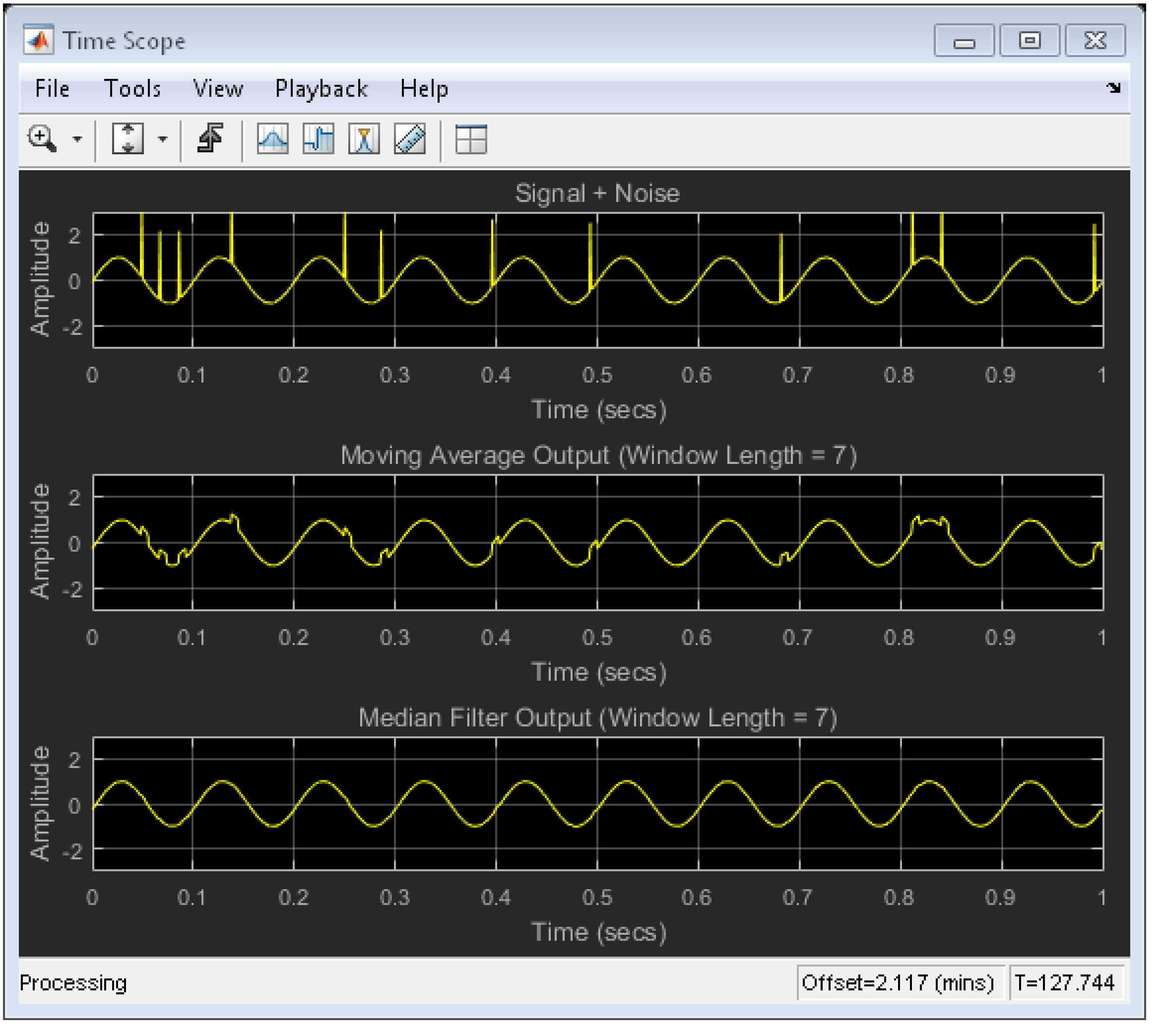
Set up a `dsp.MedianFilter` object, `medFilt`, and a `dsp.MovingAverage` object, `movavgWin`. These objects use the sliding window method with a window length of 7. Create a time scope for viewing the output.

```
Fs = 1000;
medFilt = dsp.MedianFilter(7);
movavgWin = dsp.MovingAverage(7);
scope  = dsp.TimeScope('SampleRate',Fs,...
    'TimeSpanOverrunAction','Scroll',...
    'TimeSpan',1,'ShowGrid',true,...
    'YLimits',[-3 3],...
    'LayoutDimensions',[3 1],...
    'NumInputPorts',3);
scope.ActiveDisplay = 1;
scope.Title = 'Signal + Noise';
scope.ActiveDisplay = 2;
scope.Title = 'Moving Average Output (Window Length = 7)';
scope.ActiveDisplay = 3;
scope.Title = 'Median Filter Output (Window Length = 7)';

FrameLength = 256;
count = 1;
sine = dsp.SineWave('SampleRate',Fs,'Frequency',10,...
    'SamplesPerFrame',FrameLength);
```

Filter the Noisy Sine Wave

Generate a noisy sine wave signal with a frequency of 10 Hz. Apply the median filter and the moving average object to the signal. View the output on the time scope.

```
for i = 1:500
    hfn = 3 * (rand(FrameLength,1) < 0.02);
    x = sine() + 1e-2 * randn(FrameLength,1) + hfn;
    y1 = movavgWin(x);
    y2 = medFilt(x);
    scope(x,y1,y2);
end
```



The median filter removes the high-frequency noise more effectively than the moving average object does.

Remove High-Frequency Noise from Gyroscope Data

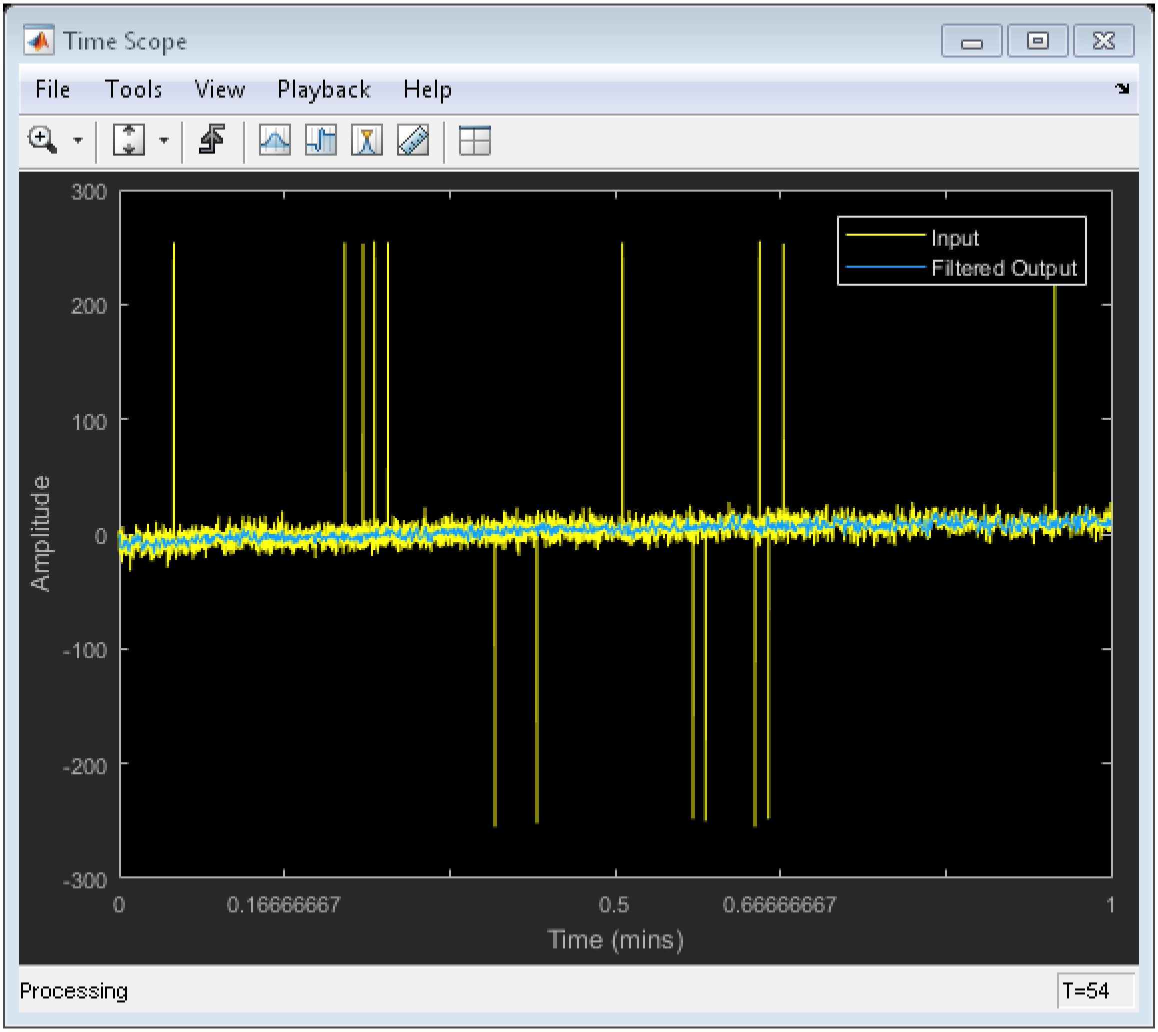
This example shows how to remove the high-frequency outliers from a streaming signal using the `dsp.MedianFilter` System object™.

Use the dsp.MatFileReader System object to read the gyroscope MAT file. The gyroscope MAT file contains 3 columns of data, with each column containing 7140 samples. The three columns represent the *X*-axis, *Y*-axis, and *Z*-axis data from the gyroscope motion sensor. Choose a frame size of 714 samples so that each column of the the data contains 10 frames. The dsp.MedianFilter System object uses a window length of 10. Create a dsp.TimeScope object to view the filtered output.

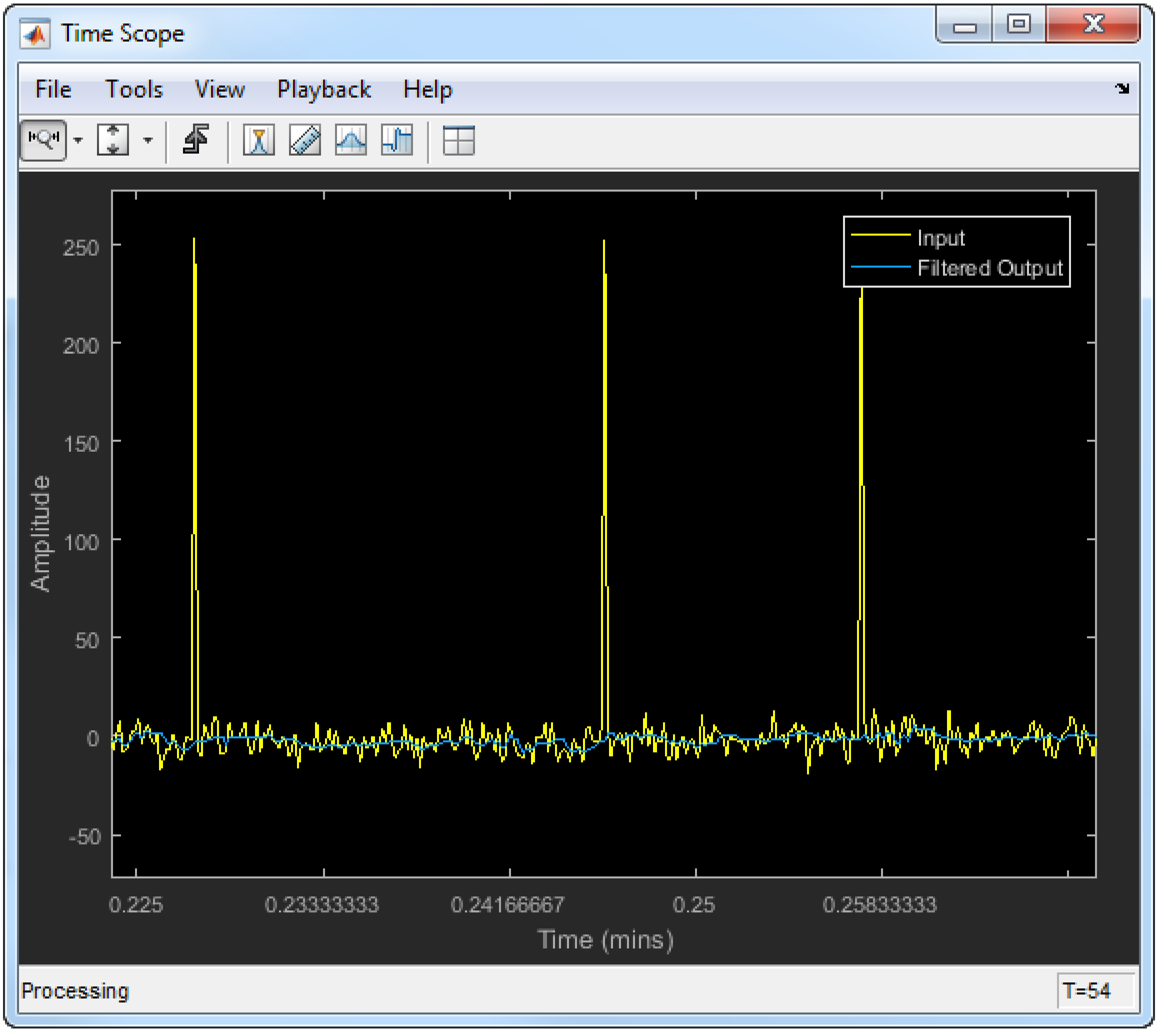
```
reader = dsp.MatFileReader('SamplesPerFrame',714,'Filename','LSM9DS1gyroData73.mat',...
    'VariableName','data');
medFilt = dsp.MedianFilter(10);
scope = dsp.TimeScope('NumInputPorts',1,'SampleRate',119,'YLimits',[-300 300],...
    'ChannelNames',{'Input','Filtered Output'},'TimeSpan',60,'ShowLegend',true);
```

Filter the gyroscope data using the dsp.MedianFilter System object. View the filtered *Z*-axis data in the time scope.

```
for i = 1:10
    gyroData = reader();
    filteredData = medFilt(gyroData);
    scope([gyroData(:,3),filteredData(:,3)]);
end
```



The original data contains several outliers. Zoom in on the data to confirm that the median filter removes all the outliers.



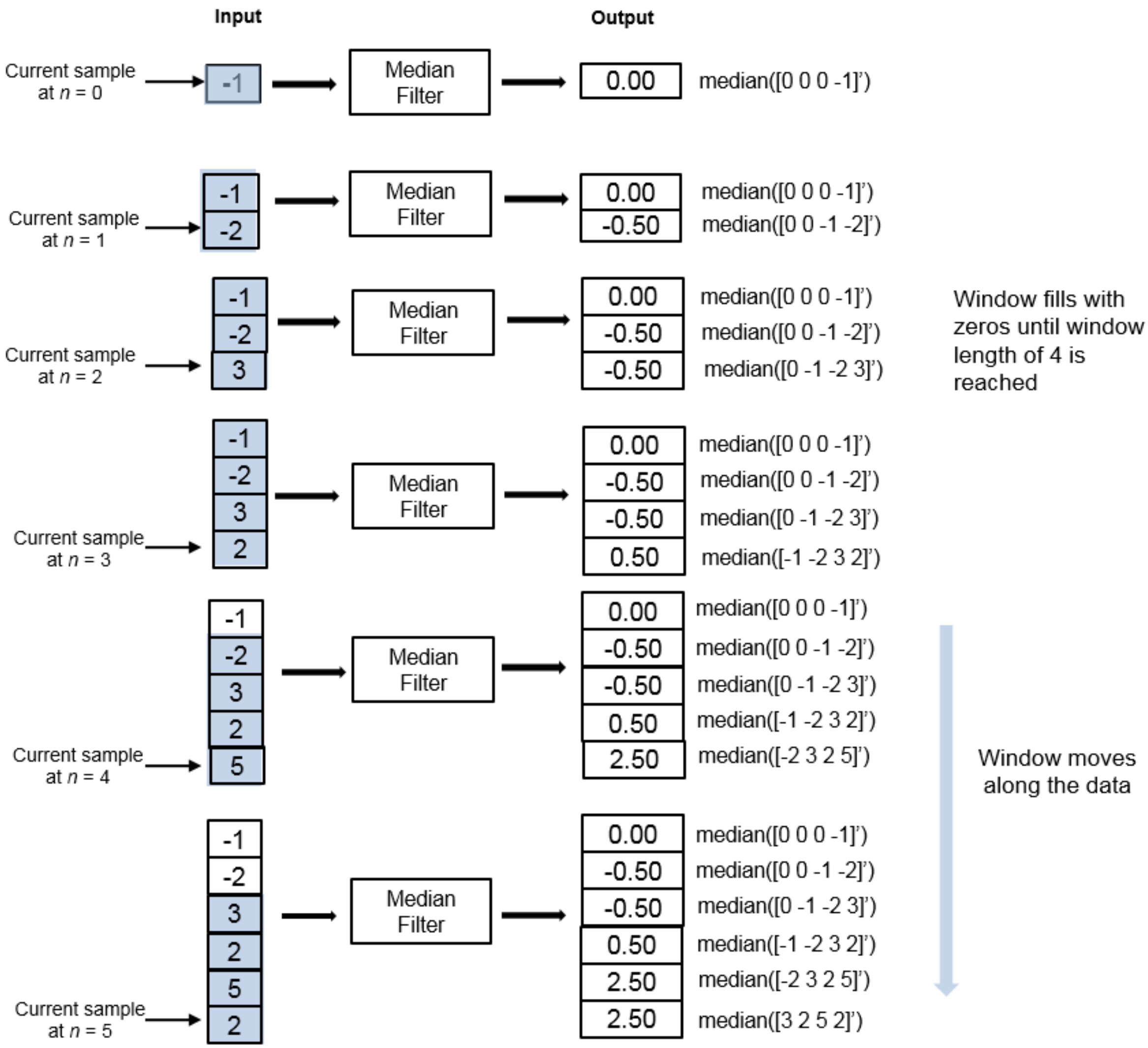
- "What Are Moving Statistics?"
- "Signal Statistics"
- "Remove High-Frequency Noise from Gyroscope Data"

Algorithms

Sliding Window Method

In the sliding window method, the output for each input sample is the median of the current sample and the *Len* - 1 previous samples. *Len* is the length of the window. To compute the first *Len* - 1 outputs, when the window does not have enough data yet, the algorithm fills the window with zeros. As an example, to compute the median value when the second input sample comes in, the algorithm fills the window with *Len* - 2 zeros. The data vector, *x*, is then the two data samples followed by *Len* - 2 zeros. This object performs median filtering on the input data over time.

Consider an example of computing the moving median of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.


Input
Output
Current sample at n = 0
-1
Median Filter
0.00
median([0 0 0 -1]')
Current sample at n = 1
-1
-2
0.00
-0.50
median([0 0 0 -1]')
median([0 0 -1 -2]')
Current sample at n = 2
3
median([0 -1 -2 3]')
Window fills with zeros until window length of 4 is reached
Current sample at n = 3
2
0.50
median([-1 -2 3 2]')
Current sample at n = 4
5
2.50
median([-2 3 2 5]')
Window moves along the data
Current sample at n = 5
median([3 2 5 2]')

References

[1] Bodenham, Dean. "Adaptive Filtering and Change Detection for Streaming Data." PH.D. Thesis. Imperial College, London, 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

System Objects

dsp.Median | dsp.MovingMaximum | dsp.MovingMinimum | dsp.MovingAverage | dsp.MovingRMS | dsp.MovingVariance | dsp.MovingStandardDeviation

Blocks

`Median` | `Median Filter` | `Moving Average` | `Moving Maximum` | `Moving Minimum` | `Moving RMS` | `Moving Standard Deviation` | `Moving Variance`

Topics

"What Are Moving Statistics?"
"Signal Statistics"
"Remove High-Frequency Noise from Gyroscope Data"

Introduced in R2016b

reset

System object: dsp.MedianFilter
Package: dsp

Reset internal states of System object

Syntax

`reset(medFilt)`

Description

`reset(medFilt)` resets the states of the input object to their initial values. The initial state values correspond to the initial conditions for the difference equation defining the object. After you call the object using nonzero input data, the filter might have nonzero states. If you continue to pass nonzero input data to the object without first calling `reset`, the object might produce different outputs for an identical input.

Introduced in R2016b

step

System object: dsp.MedianFilter
Package: dsp

Moving median of input signal

Syntax

`y = step(medFilt,x)`

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(medFilt,x)` computes the moving median of the input signal, `x`, using the sliding window method of the input `dsp.MedianFilter` System object, `medFilt`. The input `x` can be a row vector, a column vector, or a matrix. If `x` is a matrix, each column is treated as an independent channel. The moving median is computed along each channel. The input `x` can be complex and supports single and double data types.

Introduced in R2016b

dsp.Minimum System object

Package: dsp

Find minimum values of input or sequence of inputs

Description

The `dsp.Minimum` object finds the minimum value of an input or sequence of inputs.

To compute the minimum value of an input or sequence of inputs:

1. Define and set up your System object. See "Construction" on page 4-1289.
2. Call step to compute the minimum according to the properties of `dsp.Minimum`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Note: The `Running` mode in the `dsp.Minimum` System object will be removed in a future release. To compute the running minimum in MATLAB, use the dsp.MovingMinimum System object instead.

Construction

`min = dsp.Minimum` returns an object, `min`, that computes the value and/or index of the minimum elements in an input or a sequence of inputs over the specified `Dimension`.

`min = dsp.Minimum('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a minimum-finding object, `min`, with each specified property set to the specified value.

Properties

ValueOutputPort

Output minimum value

Set this property to `true` in order to output the minimum value of the input. This property applies only when you set the `RunningMinimum` on page 4-1290 property to false. The default is `true`.

RunningMinimum

Calculate over single input or multiple inputs

When you set this property to `true`, the object computes the minimum value over successive calls to the `step` method. When you set this property to `false`, the object computes the minimum value over the current input. The default is `false`.

IndexOutputPort

Output index of minimum value

Set this property to `true` to output the index of the minimum value of the input. This property applies only when you set the `RunningMinimum` on page 4-1290 property to `false`. The default is `true`.

ResetInputPort

Additional input to enable resetting of running minimum

Set this property to `true` to enable resetting of the running minimum. When you set this property to `true`, you must specify a reset input to the `step` method to reset the running minimum. This property applies only when you set the `RunningMinimum` on page 4-1290 property to `true`. The default is `false`.

ResetCondition

Condition that triggers resetting of running minimum

Specify the event that resets the running minimum as one of `| Rising edge | Falling edge | Either edge | Non-zero |`. This property applies only when you set the `ResetInputPort` on page 4-1290 property to `true`. The default is `Non-zero`.

IndexBase

Numbering base for index of minimum value

Specify the numbering used when computing the index of the minimum value as starting from either `One` or `Zero`. This property applies only when you set the `IndexOutputPort` on page 4-1290 property to `true`. The default is `One`.

Dimension

Dimension to operate along

Specify how the minimum calculation is performed over the data as one of `| All | Row | Column | Custom |`. This property applies when you set the `RunningMinimum` on page 4-1290 property to `false`. The default is `Column`.

CustomDimension

Numerical dimension to calculate over

Specify the integer dimension of the input signal over which the object finds the minimum. The cannot exceed the number of dimensions in the input signal. This property only applies when you set the `Dimension` on page 4-1291 property to `Custom`. The default is `1`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of `| Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |`. The default is `Floor`.

OverflowAction

Action to take when integer input is out-of-range

Specify the overflow action as one of `| Wrap | Saturate |`. The default is `Wrap`.

ProductDataType

Data type of product

Specify the product fixed-point data type as one of | Same as input | Custom |. The default is Same as input.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the AccumulatorDataType on page 4-1292 property to Custom. The default is numerictype([],32,30).

AccumulatorDataType

Data type of accumulator

Specify the accumulator fixed-point data type as one of | Same as product | Same as input | Custom |. The default is Same as product.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the AccumulatorDataType on page 4-1292 property to Custom. The default is numerictype([],32,30).

Methods

reset	Reset internal states of minimum-finding object
step	Operate on inputs to calculate outputs

Common to All System Objects	
clone	Create System object with same property values

Common to All System Objects	
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Compute the Minimum and Running Minimum of a Signal

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

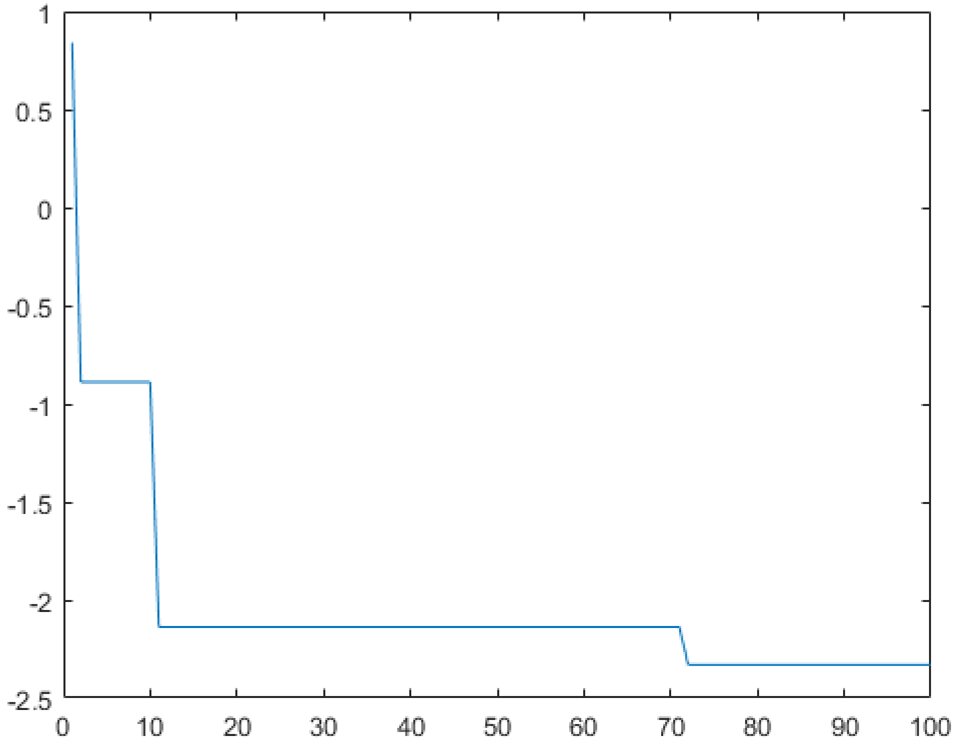
Find a minimum value and its index.

```
min1 = dsp.Minimum;
x = randn(100,1);
[y, I] = min1(x)
```

```
y =

   -2.9443


I =

    35
```

Compute a running minimum.

```
min2 = dsp.Minimum;
min2.RunningMinimum = true;
x = randn(100,1);
y = min2(x);
plot(y);
```



y(i) is the minimum of all values in the vector x(1:i);

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Minimum` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

System Objects
dsp.MovingMinimum | dsp.Mean | dsp.Minimum

Blocks
`Minimum` | `Moving Minimum`

Introduced in R2012a

reset

System object: dsp.Minimum
Package: dsp

Reset internal states of minimum-finding object

Syntax

`reset(min)`

Description

`reset(min)` sets the internal states of the `Minimum` object `min` to their initial values.

step

System object: dsp.Minimum
Package: dsp

Operate on inputs to calculate outputs

Syntax

```
[VAL,IND] = step(min,X)
VAL = step(min,X)
IND = step(min,X)
VAL = step(min,X,R)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[VAL,IND] = step(min,X)` returns the minimum value, `VAL`, and the index or position of the minimum value, `IND`, along the specified `Dimension` of `X`.

`VAL = step(min,X)` returns the minimum value, `VAL`, of the input `X`. When the `RunningMinimum` property is `true`, `VAL` corresponds to the minimum value over successive calls to the `step` method.

`IND = step(min,X)` returns the zero- or one-based index `IND` of the minimum value when the `IndexOutputPort` property is `true` and the `ValueOutputPort` property is `false`. You must set the `RunningMinimum` property to `false` to use this syntax.

`VAL = step(min,X,R)` resets the state of `min` based on the value of reset signal, `R`, and the `ResetCondition` property. To enable this type of processing, set the `RunningMinimum` property to `true` and the `ResetInputPort` property to `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.MovingAverage System object

Package: dsp

Moving average

Description

The `dsp.MovingAverage` System object computes the moving average of the input signal along each channel, independently over time. The object uses either the sliding window method or the exponential weighting method to compute the moving average. In the sliding window method, a window of specified length is moved over the data, sample by sample, and the average is computed over the data in the window. In the exponential weighting method, the object multiplies the data samples with a set of weighting factors. The average is computed by summing the weighted data. For more details on these methods, see "Algorithms" on page 2-1246.

The object accepts multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$, and $n > 1$. The object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel. However, the number of channels cannot change. This object supports C and C++ code generation.

To compute the moving average of the input:

1. Create a `dsp.MovingAverage` object and set the properties of the object.
2. Call step to compute the moving average.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`movAvg = dsp.MovingAverage` returns a moving average object, `movAvg`, using the default properties.

`movAvg = dsp.MovingAverage(Len)` sets the `WindowLength` property to `Len`.

`movAvg = dsp.MovingAverage(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

Example:

```
movAvg = dsp.MovingAverage('Method','Exponential weighting','ForgettingFactor',0.9);
```

Properties

`Method` — Averaging method

`'Sliding window'` (default) | `'Exponential weighting'`

Averaging method, specified as `'Sliding window'` or `'Exponential weighting'`.

- `'Sliding window'` — A window of length specified by `SpecifyWindowLength` is moved over the input data along each channel. For every sample the window moves by, the object computes the average over the data in the window.
- `'Exponential weighting'` — The object multiplies the samples with a set of weighting factors. The magnitude of the weighting factors decreases exponentially as the age of the data increases, never reaching zero. To compute the average, the algorithm sums the weighted data.

For more details on these methods, see “Algorithms” on page 2-1246.

`SpecifyWindowLength` — Specify window length

true (default) | false

Flag to specify a window length, specified as a scalar boolean.

- `true` — The length of the sliding window is equal to the value you specify in the `WindowLength` property.
- `false` — The length of the sliding window is infinite. In this mode, the average is computed using the current sample and all the past samples.

This property applies when you set `Method` to `'Sliding window'`.

`WindowLength` — Length of the sliding window

4 (default) | positive scalar integer

Length of the sliding window, specified as a positive scalar integer. This property applies when you set `Method` to `'Sliding window'` and `SpecifyWindowLength` to `true`.

`ForgettingFactor` — Exponential weighting factor

0.9 (default) | positive real scalar in the range (0,1]

Exponential weighting factor, specified as a positive real scalar in the range (0,1]. This property applies when you set `Method` to `'Exponential weighting'`. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the past samples are given an equal weight. This property is tunable. You can change its value even when the object is locked.

Methods

reset	Reset internal states of System object
step	Moving average of input signal

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Moving Average of Noisy Ramp Signal

Compute the moving average of a noisy ramp signal using the `dsp.MovingAverage` object.

Initialization

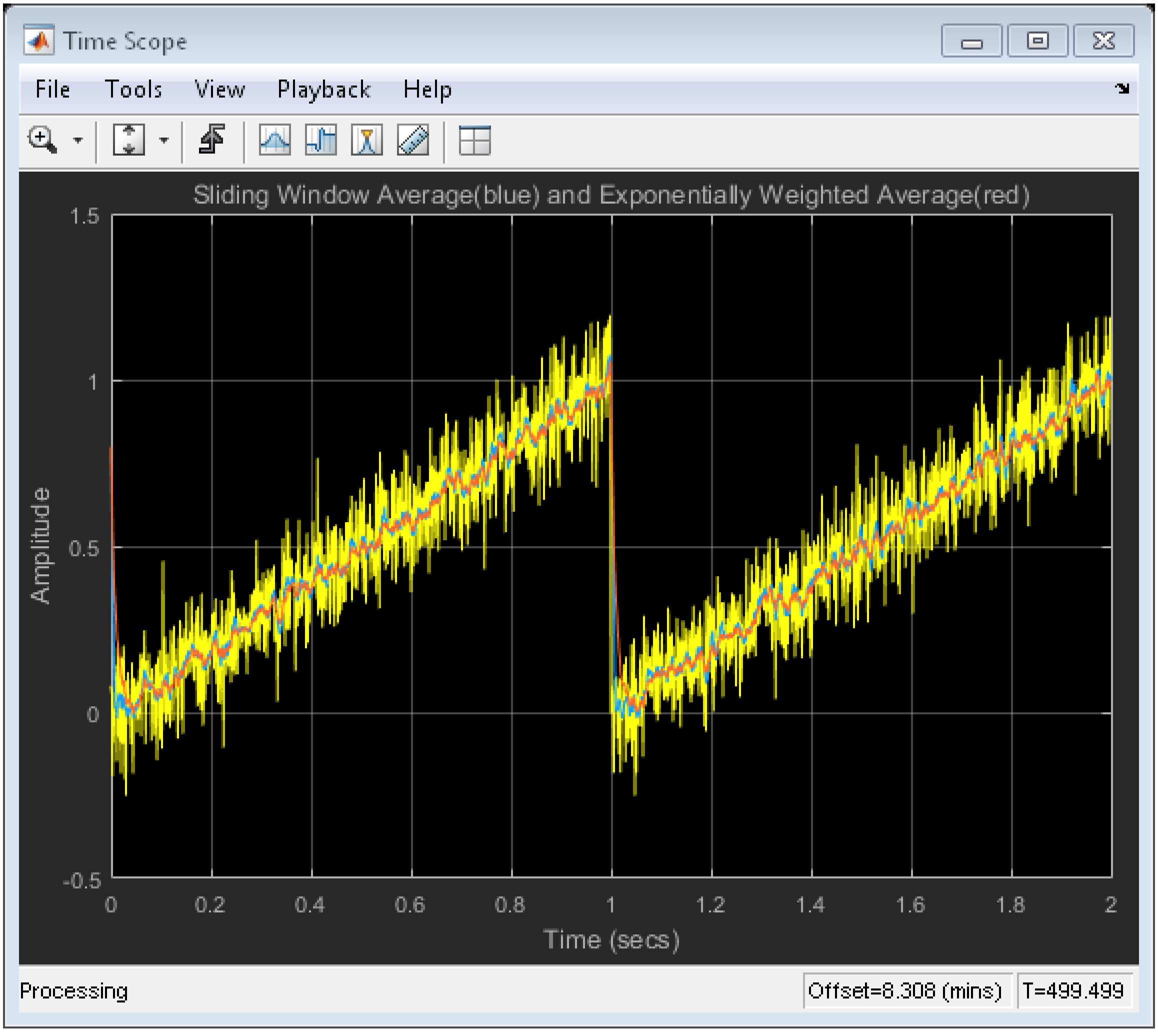
Set up `movavgWindow` and `movavgExp` objects. `movavgWindow` uses the sliding window method with a window length of 10. `movavgExp` uses the exponentially weighting method with a forgetting factor of 0.9. Create a time scope for viewing the output.

```
FrameLength = 1001;
Fs = 1000;
movavgWindow = dsp.MovingAverage(10);
movavgExp = dsp.MovingAverage('Method','Exponential weighting',...
    'ForgettingFactor',0.9);
scope  = dsp.TimeScope('SampleRate',Fs,...
    'TimeSpanOverrunAction','Scroll',...
    'TimeSpan',2,...
    'ShowGrid',true,...
    'YLimits',[-0.5 1.5]);
title = 'Sliding Window Average(blue) and Exponentially Weighted Average(red)';
scope.Title = title;
```

Compute the Average

Generate a ramp signal with an amplitude of 1.0 and a time span of 2 seconds. Apply the sliding window average and exponentially weighted average to the ramp. View the output on the time scope.

```
for i = 1:500
    t = (0:0.001:1)';
    unitstep = t>=0;
    ramp = t.*unitstep;
    x = ramp + 0.1 * randn(FrameLength,1);
    y1 = movavgWindow(x);
    y2 = movavgExp(x);
    scope([x,y1,y2]);
end
```



- "What Are Moving Statistics?"
- "Sliding Window Method and Exponential Weighting Method"
- "How Is a Moving Average Filter Different from an FIR Filter?"
- "Measure Statistics of Streaming Signals"
- "Signal Statistics"

Algorithms

Sliding Window Method

In the sliding window method, the output for each input sample is the average of the current sample and the *Len* - 1 previous samples. *Len* is the length of the window. To compute the first *Len* - 1 outputs, when the window does not have enough data yet, the algorithm fills the window with zeros. As an example, to compute the average when the second input sample comes in, the algorithm fills the window with *Len* - 2 zeros. The data vector, x, is then the two data samples followed by *Len* - 2 zeros.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving average of the current sample and all the previous samples in the channel.

For an example, see "Sliding Window Method and Exponential Weighting Method".

Exponential Weighting Method

In the exponential weighting method, the moving average is computed recursively using these formulas:

$$w_{N,\lambda} = \lambda w_{N-1,\lambda} + 1,$$

$$\bar{x}_{N,\lambda} = \left(1 - \frac{1}{w_{N,\lambda}}\right)\bar{x}_{N-1,\lambda} + \left(\frac{1}{w_{N,\lambda}}\right)x_N$$

- $\bar{x}_{N,\lambda}$ — Moving average at the current sample
- x_N — Current data input sample
- $\bar{x}_{N-1,\lambda}$ — Moving average at the previous sample
- λ — Forgetting factor
- $w_{N,\lambda}$ — Weighting factor applied to the current data sample
- $\left(1 - \frac{1}{w_{N,\lambda}}\right)\bar{x}_{N-1,\lambda}$ — Effect of the previous data on the average

For the first sample, where $N = 1$, the algorithm chooses $w_{N,\lambda} = 1$. For the next sample, the weighting factor is updated and used to compute the average, as per the recursive equation. As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the current average than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the previous samples are given an equal weight.

For an example, see “Sliding Window Method and Exponential Weighting Method”.

References

[1] Bodenham, Dean. “Adaptive Filtering and Change Detection for Streaming Data.” PH.D. Thesis. Imperial College, London, 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

System Objects

dsp.MovingRMS | dsp.MovingMaximum | dsp.MovingMinimum | dsp.MovingStandardDeviation | dsp.MovingVariance | dsp.MedianFilter

Blocks
`Median Filter` | `Moving Average` | `Moving Maximum` | `Moving Minimum` | `Moving RMS` | `Moving Standard Deviation` | `Moving Variance`

Topics
"What Are Moving Statistics?"
"Sliding Window Method and Exponential Weighting Method"
"How Is a Moving Average Filter Different from an FIR Filter?"
"Measure Statistics of Streaming Signals"
"Signal Statistics"

Introduced in R2016b

reset

System object: dsp.MovingAverage
Package: dsp

Reset internal states of System object

Syntax

`reset(movAvg)`

Description

`reset(movAvg)` resets the states of the input object to their initial values. The initial state values correspond to the initial conditions for the difference equation defining the object. After you call the object using nonzero input data, the filter might have nonzero states. If you continue to pass nonzero input data to the object without first calling `reset`, the object might produce different outputs for an identical input.

Introduced in R2016b

step

System object: dsp.MovingAverage
Package: dsp

Moving average of input signal

Syntax

```
y = step(movAvg,x)
```

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(movAvg,x)` computes the moving average of the input signal, `x`, using either the sliding window method or exponential weighting method of the input `dsp.MovingAverage` System object, `movAvg`. The input `x` can be a row vector, a column vector, or a matrix. If `x` is a matrix, each column is treated as an independent channel. The moving average is computed along each channel. The input `x` can be complex and supports single and double data types.

Introduced in R2016b

dsp.MovingMaximum System object

Package: dsp

Moving maximum

Description

The `dsp.MovingMaximum` System object determines the moving maximum of the input signal along each channel, independently over time. The object uses the sliding window method to determine the moving maximum. In this method, a window of specified length is moved over each channel, sample by sample, and the object determines the maximum of the data in the window. For more details, see "Algorithms" on page 4-1313.

The object accepts multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$, and $n > 1$. The object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel. However, the number of channels cannot change. This object supports C and C++ code generation.

To determine the moving maximum of the input:

1. Create a `dsp.MovingMaximum` object and set the properties of the object.
2. Call step to compute the moving maximum.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`movMax = dsp.MovingMaximum` returns a moving maximum object, `movMax`, using the default properties.

`movMax = dsp.MovingMaximum(Len)` sets the `WindowLength` property to `Len`.

`movMax = dsp.MovingMaximum(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

Example:

```
movMax = dsp.MovingMaximum('SpecifyWindowLength',1,'WindowLength',10);
```

Properties

`SpecifyWindowLength` — Specify window length

true (default) | false

Flag to specify a window length, specified as a scalar boolean.

- `true` — The length of the sliding window is equal to the value you specify in the `WindowLength` property.
- `false` — The length of the sliding window is infinite. In this mode, the object determines the maximum of the current sample and all the past samples.

`WindowLength` — Length of the sliding window

4 (default) | positive scalar integer

Length of the sliding window, specified as a positive scalar integer. This property applies when you set `SpecifyWindowLength` to `true`.

Methods

reset	Reset internal states of System object
step	Moving maximum of input signal

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Moving Maximum of Sine Wave Signal

Compute the moving maximum of a sum of three sine waves with varying amplitude. Use a sliding window of length 30.

Initialization

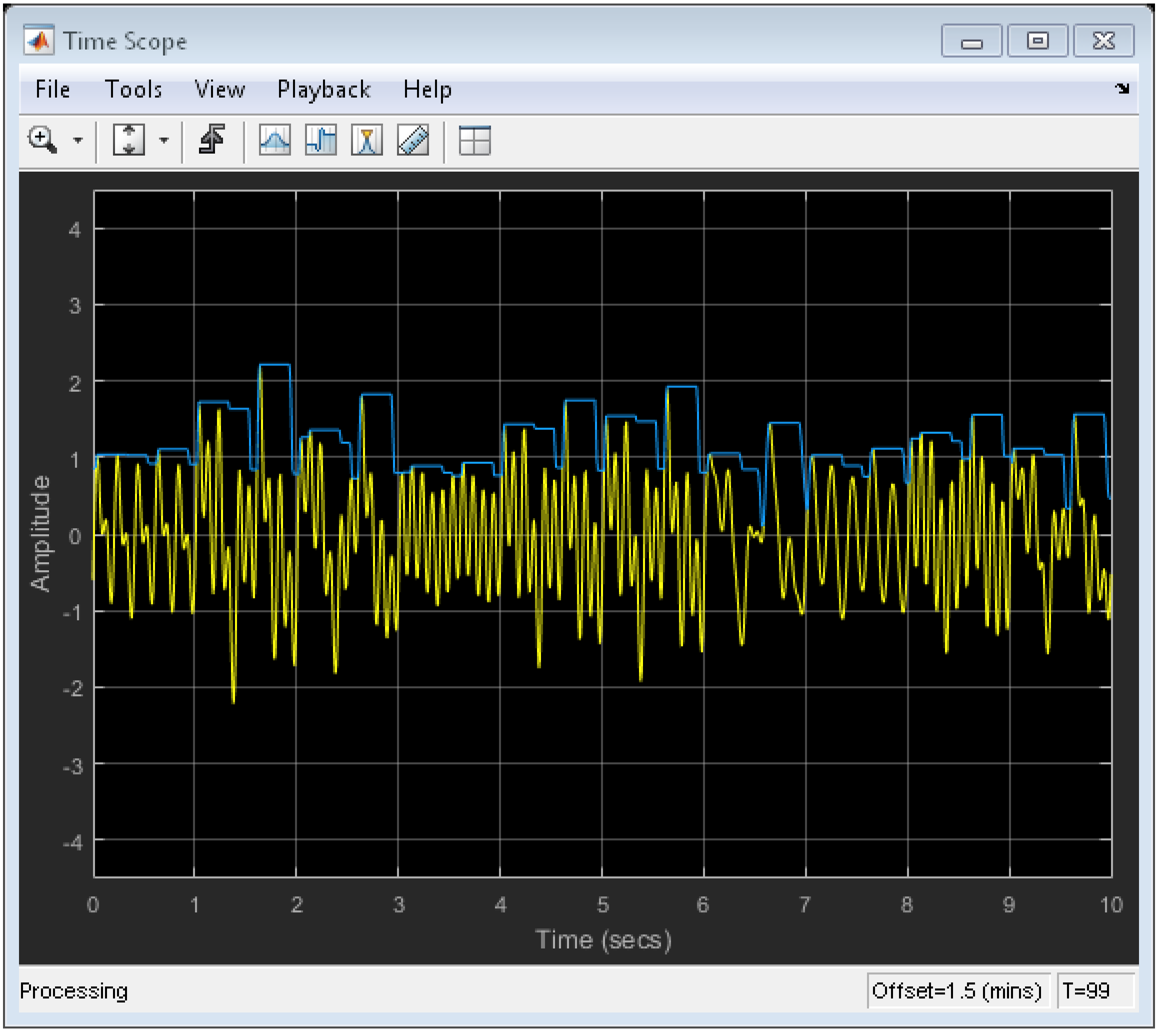
Set up an input signal that is a sum of three sine waves with frequences at 2 Hz, 5 Hz, and 10 Hz. The sampling frequency is 100 Hz. Create a `dsp.MovingMaximum` object with a window length of 30. Create a time scope for viewing the output.

```
sin = dsp.SineWave('SampleRate',100,...
    'Frequency',[2 5 10],...
    'SamplesPerFrame',100);
movMax = dsp.MovingMaximum(30);
scope  = dsp.TimeScope('SampleRate',100,...
    'TimeSpanOverrunAction','Scroll',...
    'TimeSpan',10,'ShowGrid',true,...
    'YLimits',[-4.5 4.5]);
```

Compute the Moving Maximum

Each sine wave component of the input signal has a different amplitude that varies with the iteration. Use the `movMax` object to determine the maximum value of the current sample and the past 29 samples of the input signal.

```
for index = 1:100
    sin.Amplitude = rand(1,3);
    x = sum(sin(),2);
    xmax = movMax(x);
    scope([x,xmax])
end
```



- "Signal Statistics"
- "What Are Moving Statistics?"

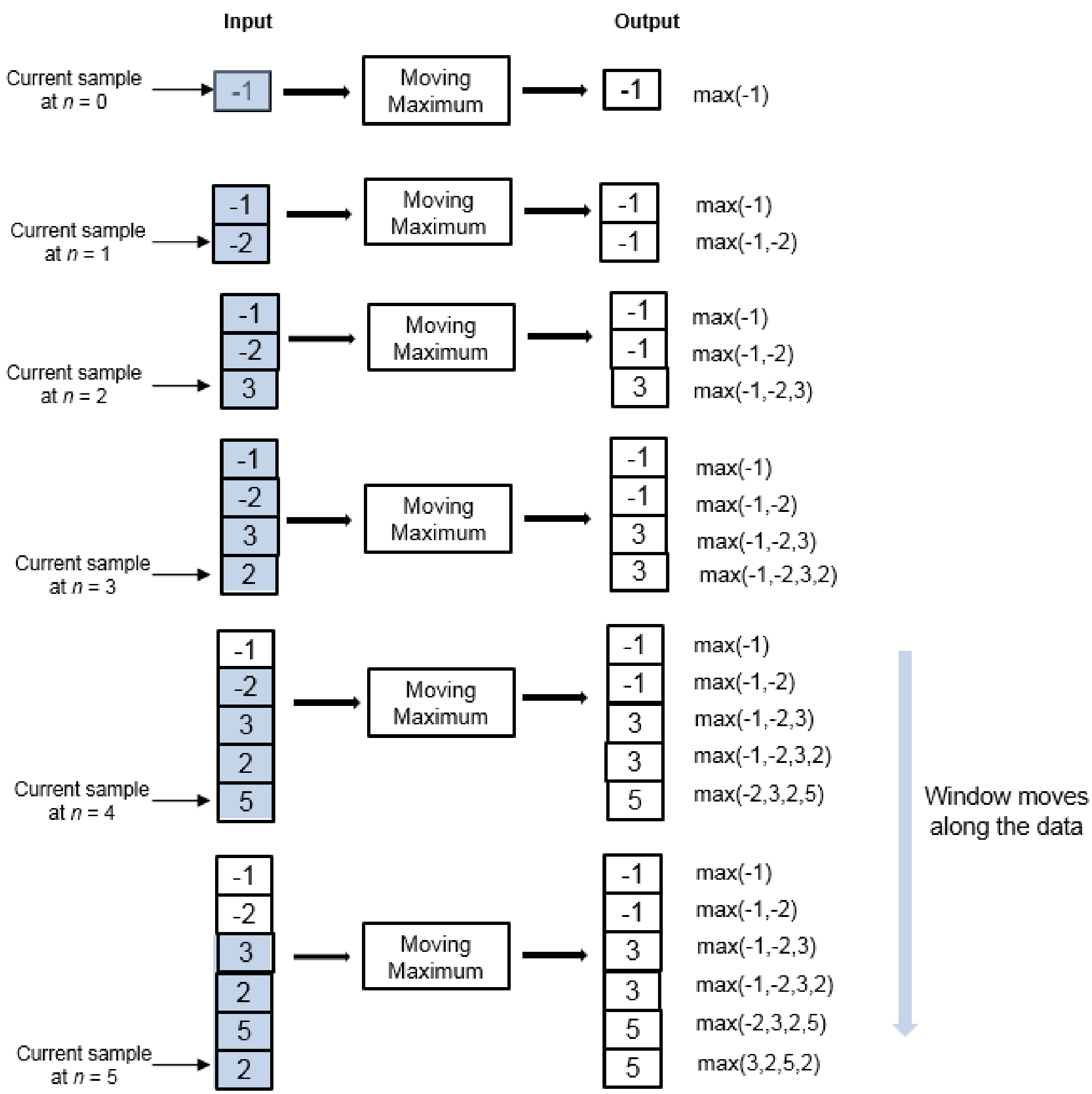
Algorithms

Sliding Window Method

In the sliding window method, the output for each input sample is the maximum of the current sample and the *Len* - 1 previous samples. *Len* is the length of the window. When the algorithm computes the first *Len* - 1 outputs, the length of the window is the length of the data that is available.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the maximum of the current sample and all the previous samples in the channel.

Consider an example of computing the moving maximum of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.


Input
Output
Current sample at n = 0
-1
Moving Maximum
-1
max(-1)
Current sample at n = 1
-1
-2
Moving Maximum
-1
-1
max(-1)
max(-1,-2)
Current sample at n = 2
-1
-2
3
Moving Maximum
-1
-1
3
max(-1)
max(-1,-2)
max(-1,-2,3)
Current sample at n = 3
-1
-2
3
2
Moving Maximum
-1
-1
3
3
max(-1)
max(-1,-2)
max(-1,-2,3)
max(-1,-2,3,2)
Current sample at n = 4
-1
-2
3
2
5
Moving Maximum
-1
-1
3
3
5
max(-1)
max(-1,-2)
max(-1,-2,3)
max(-1,-2,3,2)
max(-2,3,2,5)
Window moves along the data
Current sample at n = 5
-1
-2
3
2
5
2
Moving Maximum
-1
-1
3
3
5
5
max(-1)
max(-1,-2)
max(-1,-2,3)
max(-1,-2,3,2)
max(-2,3,2,5)
max(3,2,5,2)

References

[1] Bodenham, Dean. “Adaptive Filtering and Change Detection for Streaming Data.” PH.D. Thesis. Imperial College, London, 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

System Objects

dsp.Maximum | dsp.MovingMinimum | dsp.MovingAverage | dsp.MovingRMS | dsp.MovingStandardDeviation | dsp.MovingVariance | dsp.MedianFilter

Blocks

Median Filter | Moving Average | Moving Maximum | Moving Minimum | Moving RMS | Moving Standard Deviation | Moving Variance

Topics

“Signal Statistics”
“What Are Moving Statistics?”

Introduced in R2016b

reset

System object: dsp.MovingMaximum
Package: dsp

Reset internal states of System object

Syntax

`reset(movMax)`

Description

`reset(movMax)` resets the states of the input object to their initial values. The initial state values correspond to the initial conditions for the difference equation defining the object. After you call the object using nonzero input data, the filter might have nonzero states. If you continue to pass nonzero input data to the object without first calling `reset`, the object might produce different outputs for an identical input.

Introduced in R2016b

step

System object: dsp.MovingMaximum
Package: dsp

Moving maximum of input signal

Syntax

`y = step(movMax,x)`

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(movMax,x)` determines the moving maximum of the input signal, `x`, using the sliding window method of the input `dsp.MovingMaximum` System object, `movMax`. The input `x` can be a row vector, a column vector, or a matrix. If `x` is a matrix, each column is treated as an independent channel. The moving maximum is determined along each channel. The input `x` can be complex and supports single and double data types.

Introduced in R2016b

dsp.MovingMinimum System object

Package: dsp

Moving minimum

Description

The `dsp.MovingMinimum` System object determines the moving minimum of the input signal along each channel, independently over time. The object uses the sliding window method to determine the moving minimum. In this method, a window of specified length is moved over each channel, sample by sample, and the object determines the minimum of the data in the window. For more details, see "Algorithms" on page 4-1322.

The object accepts multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$, and $n > 1$. The object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel. However, the number of channels cannot change. This object supports C and C++ code generation.

To determine the moving minimum of the input:

1 Create a `dsp.MovingMinimum` object and set the properties of the object.
2 Call step to compute the moving minimum.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`movMin = dsp.MovingMinimum` returns a moving minimum object, `movMin`, using the default properties.

`movMin = dsp.MovingMinimum(Len)` sets the `WindowLength` property to `Len`.

`movMin = dsp.MovingMinimum(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

Example:

```
movMin = dsp.MovingMinimum('SpecifyWindowLength',1,'WindowLength',10);
```

Properties

`SpecifyWindowLength` — Specify window length

true (default) | false

Flag to specify a window length, specified as a scalar boolean.

- `true` — The length of the sliding window is equal to the value you specify in the `WindowLength` property.
- `false` — The length of the sliding window is infinite. In this mode, the object determines the minimum of the current sample and all the past samples.

`WindowLength` — Length of the sliding window

4 (default) | positive scalar integer

Length of the sliding window, specified as a positive scalar integer. This property applies when you set `SpecifyWindowLength` to `true`.

Methods

reset	Reset internal states of System object
step	Moving minimum of input signal

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Moving Minimum of Sine Wave Signal

Compute the moving minimum of a sum of three sine waves with varying amplitude. Use a sliding window of length 30.

Initialization

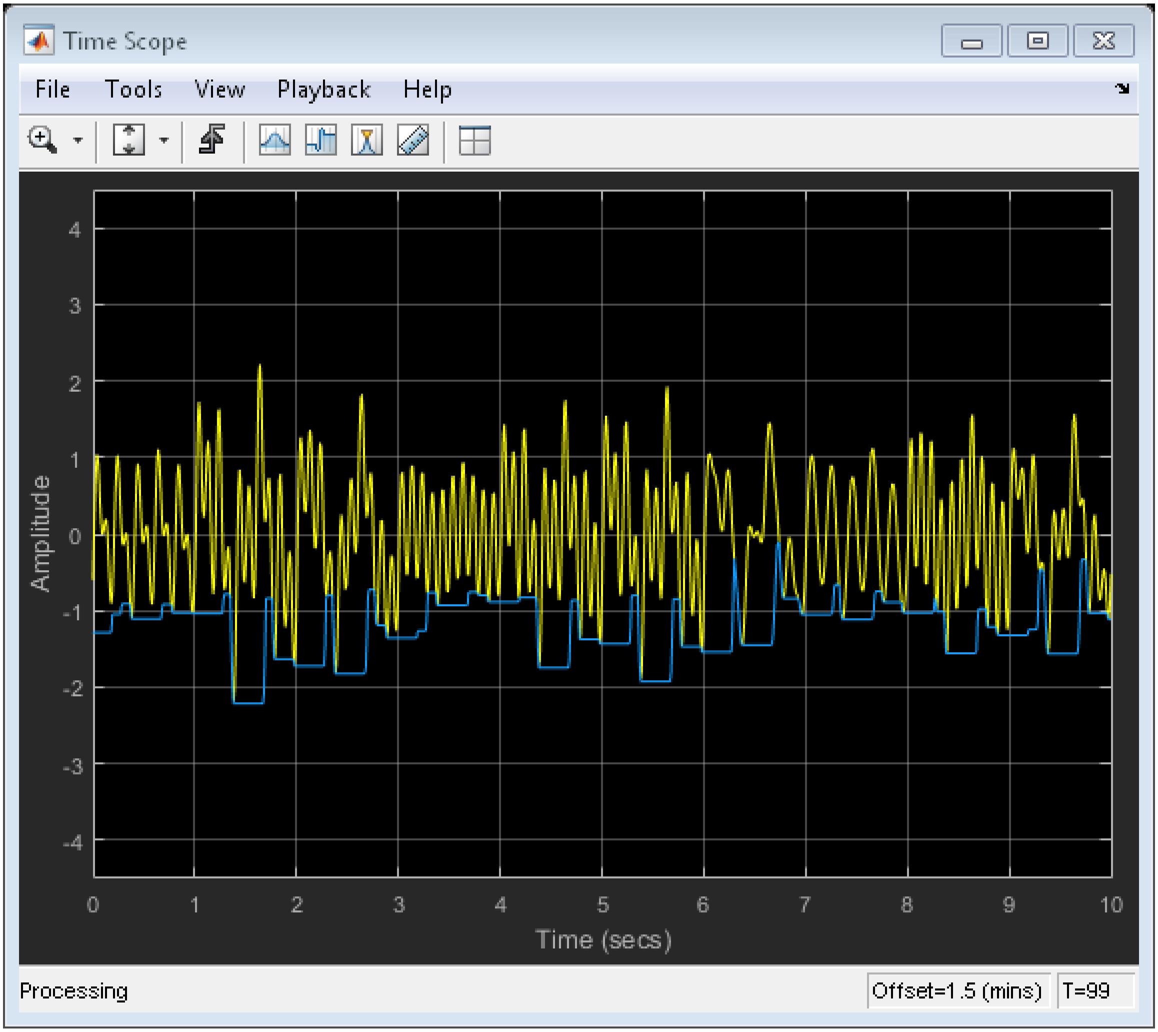
Set up an input signal that is a sum of three sine waves with frequences at 2 Hz, 5 Hz, and 10 Hz. The sampling frequency is 100 Hz. Create a `dsp.MovingMinimum` object with a window length of 30. Create a time scope for viewing the output.

```
sin = dsp.SineWave('SampleRate',100,...
    'Frequency',[2 5 10],...
    'SamplesPerFrame',100);
movMin = dsp.MovingMinimum(30);
scope  = dsp.TimeScope('SampleRate',100,...
    'TimeSpanOverrunAction','Scroll',...
    'TimeSpan',10,'ShowGrid',true,...
    'YLimits',[-4.5 4.5]);
```

Compute the Moving Minimum

Each sine wave component of the input signal has a different amplitude that varies with the iteration. Use the `movMin` object to determine the minimum value of the current sample and the past 29 samples of the input signal.

```
for index = 1:100
    sin.Amplitude = rand(1,3);
    x = sum(sin(),2);
    xmin = movMin(x);
    scope([x,xmin])
end
```



- "Signal Statistics"
- "What Are Moving Statistics?"

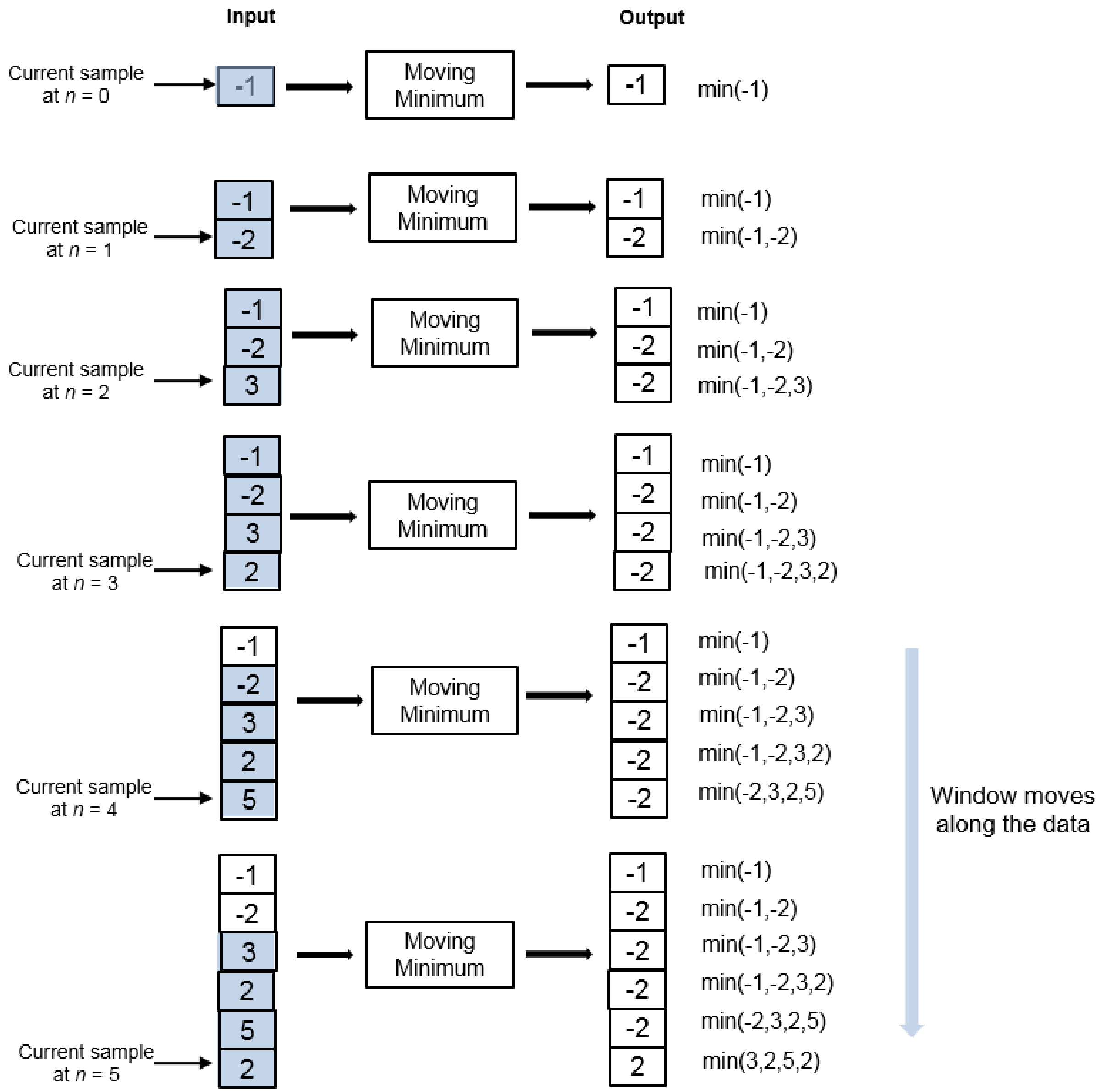
Algorithms

Sliding Window Method

In the sliding window method, the output for each input sample is the minimum of the current sample and the *Len* - 1 previous samples. *Len* is the length of the window. When the algorithm computes the first *Len* - 1 outputs, the length of the window is the length of the data that is available.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the minimum of the current sample and all the previous samples in the channel.

Consider an example of computing the moving minimum of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.


Input
Output
Current sample at n = 0
-1
Moving Minimum
-1
min(-1)
-1
-2
Current sample at n = 1
Moving Minimum
-1
-2
min(-1)
min(-1,-2)
-1
-2
3
Current sample at n = 2
Moving Minimum
-1
-2
-2
min(-1)
min(-1,-2)
min(-1,-2,3)
-1
-2
3
2
Current sample at n = 3
Moving Minimum
-1
-2
-2
-2
min(-1)
min(-1,-2)
min(-1,-2,3)
min(-1,-2,3,2)
-1
-2
3
2
5
Current sample at n = 4
Moving Minimum
-1
-2
-2
-2
-2
min(-1)
min(-1,-2)
min(-1,-2,3)
min(-1,-2,3,2)
min(-2,3,2,5)
Window moves along the data
-1
-2
3
2
5
2
Current sample at n = 5
Moving Minimum
-1
-2
-2
-2
-2
2
min(-1)
min(-1,-2)
min(-1,-2,3)
min(-1,-2,3,2)
min(-2,3,2,5)
min(3,2,5,2)

References

[1] Bodenham, Dean. "Adaptive Filtering and Change Detection for Streaming Data." PH.D. Thesis. Imperial College, London, 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

System Objects

dsp.Minimum | dsp.MovingMaximum | dsp.MovingAverage | dsp.MovingRMS | dsp.MovingStandardDeviation | dsp.MovingVariance | dsp.MedianFilter

Blocks

`Median Filter` | `Minimum` | `Moving Average` | `Moving Maximum` | `Moving Minimum` | `Moving RMS` | `Moving Standard Deviation` | `Moving Variance`

Topics

"Signal Statistics"
"What Are Moving Statistics?"

Introduced in R2016b

reset

System object: dsp.MovingMinimum
Package: dsp

Reset internal states of System object

Syntax

`reset(movMin)`

Description

`reset(movMin)` resets the states of the input object to their initial values. The initial state values correspond to the initial conditions for the difference equation defining the object. After you call the object using nonzero input data, the filter might have nonzero states. If you continue to pass nonzero input data to the object without first calling `reset`, the object might produce different outputs for an identical input.

Introduced in R2016b

step

System object: dsp.MovingMinimum
Package: dsp

Moving minimum of input signal

Syntax

```
y = step(movMin,x)
```

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(movMin,x)` determines the moving minimum of the input signal, `x`, using the sliding window method of the input `dsp.MovingMinimum` System object, `movMin`. The input `x` can be a row vector, a column vector, or a matrix. If `x` is a matrix, each column is treated as an independent channel. The moving minimum is determined along each channel. The input `x` can be complex and supports single and double data types.

Introduced in R2016b

dsp.MovingRMS System object

Package: dsp

Moving Root Mean Square

Description

The `dsp.MovingRMS` System object computes the moving Root Mean Square (RMS) of the input signal along each channel, independently over time. The object uses either the sliding window method or the exponential weighting method to compute the moving RMS. In the sliding window method, a window of specified length is moved over the data, sample by sample, and the RMS is computed over the data in the window. In the exponential weighting method, the object squares the data samples, multiplies them with a set of weighting factors, and sums the weighed data. The object then computes the RMS by taking the square root of the sum. For more details on these methods, see "Algorithms" on page 2-1261.

The object accepts multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$, and $n > 1$. The object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel. However, the number of channels cannot change. This object supports C and C++ code generation.

To compute the moving RMS of the input:

1 Create a `dsp.MovingRMS` object and set the properties of the object.
2 Call step to compute the moving RMS.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`movRMS = dsp.MovingRMS` returns a moving RMS object, `movRMS`, using the default properties.

`movRMS = dsp.MovingRMS(Len)` sets the `WindowLength` property to `Len`.

`movRMS = dsp.MovingRMS(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

Example:

```
movRMS = dsp.MovingRMS('Method','Exponential weighting','ForgettingFactor',0.9);
```

Properties

`Method` — Moving RMS method

`'Sliding window'` (default) | `'Exponential weighting'`

Moving RMS method, specified as `'Sliding window'` or `'Exponential weighting'`.

- `'Sliding window'` — A window of length specified by `SpecifyWindowLength` is moved over the input data along each channel. For every sample the window moves by, the object computes the RMS over the data in the window.
- `'Exponential weighting'` — The object multiplies the squares of the samples with a set of weighting factors. The magnitude of the weighting factors decreases exponentially as the age of the data increases, never reaching zero. To compute the RMS, the algorithm sums the weighted data, and takes a square root of the sum.

For more details on these methods, see “Algorithms” on page 2-1261.

`SpecifyWindowLength` — Specify window length

true (default) | false

Flag to specify a window length, specified as a scalar Boolean.

- `true` — The length of the sliding window is equal to the value you specify in the `WindowLength` property.
- `false` — The length of the sliding window is infinite. In this mode, the RMS is computed using the current sample and all past samples.

This property applies when you set `Method` to `'Sliding window'`.

`WindowLength` — Length of the sliding window

4 (default) | positive scalar integer

Length of the sliding window, specified as a positive scalar integer. This property applies when you set `Method` to `'Sliding window'` and `SpecifyWindowLength` to `true`.

`ForgettingFactor` — Exponential weighting factor
0.9 (default) | positive real scalar in the range (0,1]

Exponential weighting factor, specified as a positive real scalar in the range (0,1]. This property applies when you set `Method` to `'Exponential weighting'`.

A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the past samples are given an equal weight.

This property is tunable. You can change its value even when the object is locked.

Methods

reset	Reset internal states of System object
step	Moving RMS of input signal

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Moving RMS of Noisy Square Wave Signal

Compute the moving RMS of a noisy square wave signal with varying amplitude using the `dsp.MovingRMS` object.

Initialization

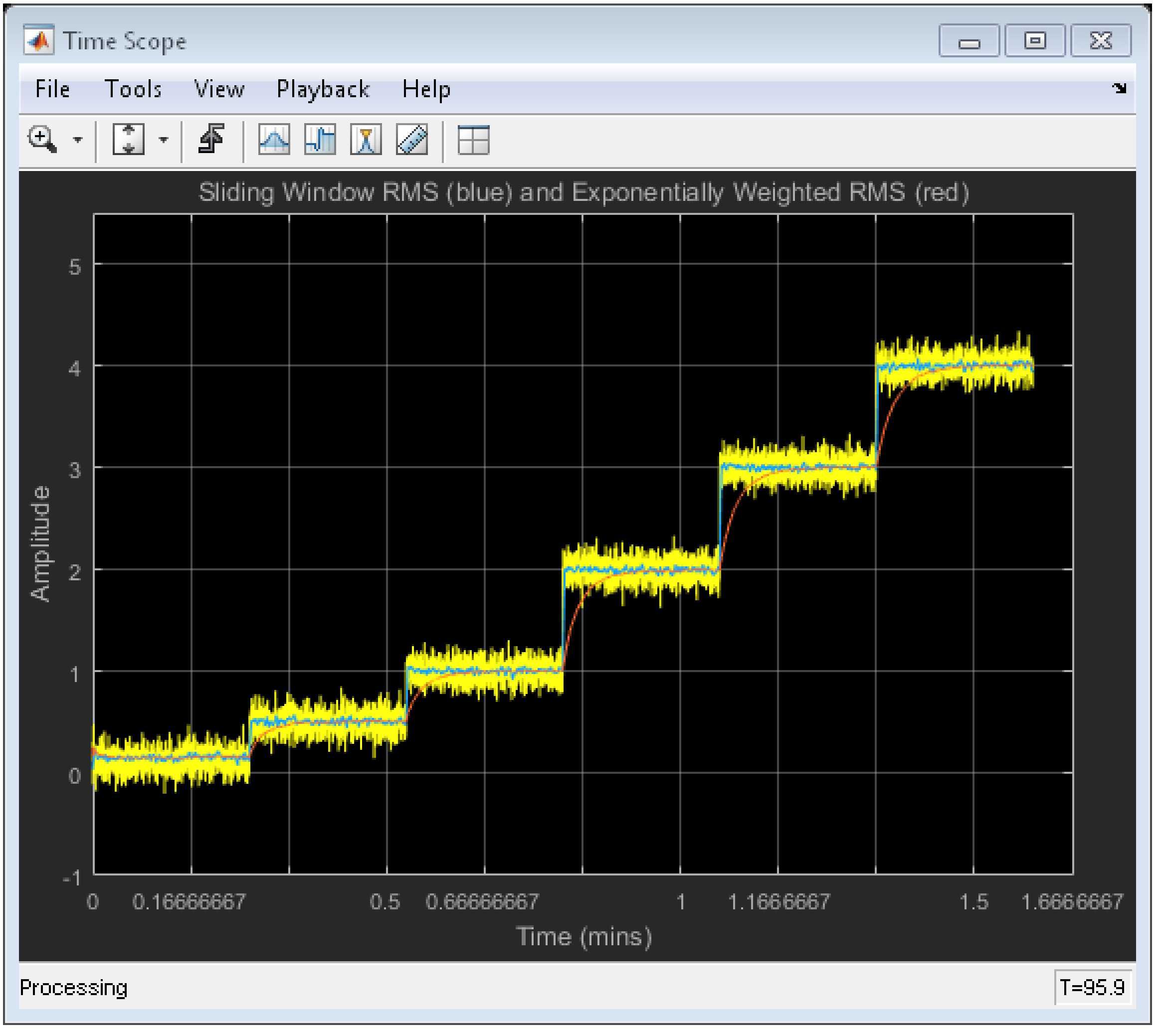
Set up movrmsWin and movrmsExp objects. movrmsWin uses the sliding window method with a window length of 20. movrmsExp uses the exponential weighting method with a forgetting factor of 0.995. Create a time scope for viewing the output.

```
FrameLength = 10;
Fs = 100;
movrmsWin = dsp.MovingRMS(20);
movrmsExp = dsp.MovingRMS('Method','Exponential weighting',...
    'ForgettingFactor',0.995);
scope  = dsp.TimeScope('SampleRate',Fs,...
    'TimeSpanOverrunAction','Scroll',...
    'TimeSpan',100,...
    'ShowGrid',true,...
    'YLimits',[-1.0 5.5]);
title = 'Sliding Window RMS (blue) and Exponentially Weighted RMS (red)';
scope.Title = title;
```

Compute the RMS

Generate a noisy square wave signal. Vary the amplitude of the square wave after a given number of frames. Apply the sliding window method and the exponential weighting method on this signal. View the output on the time scope.

```
count = 1;
Vect = [1/8 1/2 1 2 3 4];
index = 1;
for index = 1:length(Vect)
    V = Vect(index);
    for i = 1:160
        x = V + 0.1 * randn(FrameLength,1);
        y1 = movrmsWin(x);
        y2 = movrmsExp(x);
        scope([x,y1,y2]);
    end
end
```



- "What Are Moving Statistics?"
- "Signal Statistics"
- "Sliding Window Method and Exponential Weighting Method"
- "Energy Detection in the Time Domain"

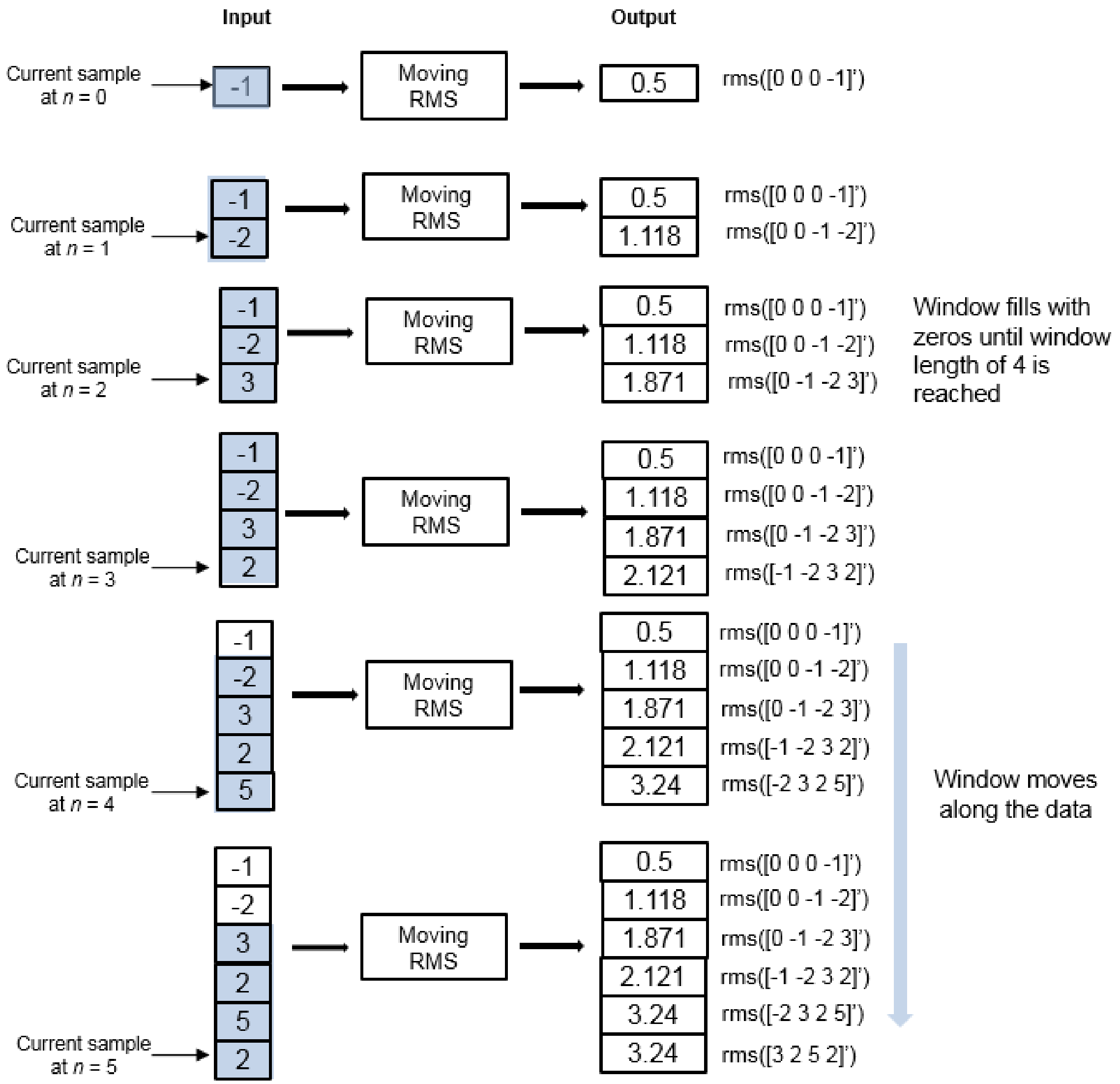
Algorithms

Sliding Window Method

In the sliding window method, the output for each input sample is the RMS of the current sample and the *Len* - 1 previous samples. *Len* is the length of the window. To compute the first *Len* - 1 outputs, when the window does not have enough data yet, the algorithm fills the window with zeros. As an example, to compute the RMS when the second input sample comes in, the algorithm fills the window with *Len* - 2 zeros. The data vector, x, is then the two data samples followed by *Len* - 2 zeros.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving RMS of the current sample and all the previous samples in the channel.

Consider an example of computing the moving RMS of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.


Input
Output
Current sample at n = 0
-1
Moving RMS
0.5
rms([0 0 0 -1]')
Current sample at n = 1
-1
-2
Moving RMS
0.5
1.118
rms([0 0 0 -1]')
rms([0 0 -1 -2]')
Current sample at n = 2
-1
-2
3
Moving RMS
0.5
1.118
1.871
rms([0 0 0 -1]')
rms([0 0 -1 -2]')
rms([0 -1 -2 3]')
Window fills with zeros until window length of 4 is reached
Current sample at n = 3
-1
-2
3
2
Moving RMS
0.5
1.118
1.871
2.121
rms([0 0 0 -1]')
rms([0 0 -1 -2]')
rms([0 -1 -2 3]')
rms([-1 -2 3 2]')
Current sample at n = 4
-1
-2
3
2
5
Moving RMS
0.5
1.118
1.871
2.121
3.24
rms([0 0 0 -1]')
rms([0 0 -1 -2]')
rms([0 -1 -2 3]')
rms([-1 -2 3 2]')
rms([-2 3 2 5]')
Window moves along the data
Current sample at n = 5
-1
-2
3
2
5
2
Moving RMS
0.5
1.118
1.871
2.121
3.24
3.24
rms([0 0 0 -1]')
rms([0 0 -1 -2]')
rms([0 -1 -2 3]')
rms([-1 -2 3 2]')
rms([-2 3 2 5]')
rms([3 2 5 2]')

Exponential Weighting Method

In the exponential weighting method, the moving RMS is computed recursively using these formulas:

$$w_{N,\lambda} = \lambda w_{N-1,\lambda} + 1,$$

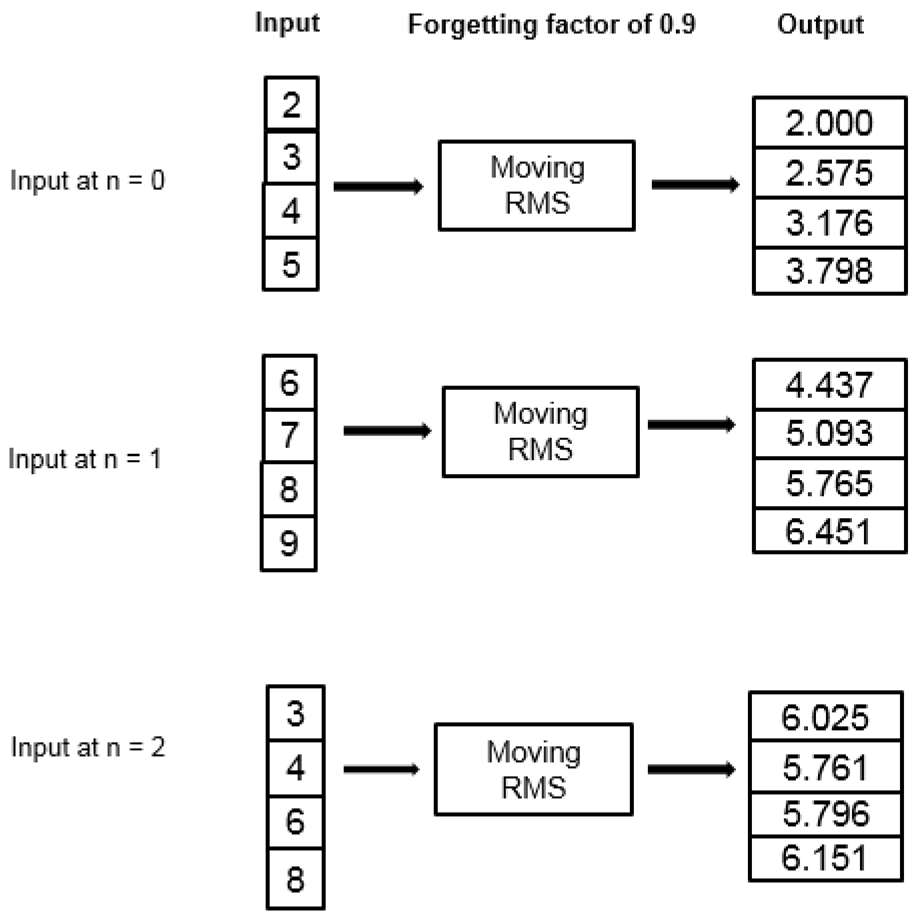
$$x\_rms_{N,\lambda} = \sqrt{\left(1 - \frac{1}{w_{N,\lambda}}\right) x\_rms_{N-1,\lambda} + \left(\frac{1}{w_{N,\lambda}}\right) x^2{}_N}$$

- $x\_rms_{N,\lambda}$ — Moving RMS at the current sample
- $x^2{}_N$ — Square of the current input data sample
- $x\_rms_{N-1,\lambda}$ — Moving RMS at the previous sample
- λ — Forgetting factor
- $w_{N,\lambda}$ — Weighting factor applied to the current data sample
- $\left(1 - \frac{1}{w_{N,\lambda}}\right) x\_rms_{N-1,\lambda}$ — Effect of the previous data on the RMS

For the first sample, where $N = 1$, the algorithm chooses $w_{N,\lambda}$ = 1. For the next sample, the weighting factor is updated and used to compute the RMS, as per the recursive equation. As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the current RMS than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the previous samples are given an equal weight.

Here is an example of computing the moving RMS using the exponential weighting method. The forgetting factor is 0.9.



References

[1] Bodenham, Dean. “Adaptive Filtering and Change Detection for Streaming Data.” PH.D. Thesis. Imperial College, London, 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

System Objects

dsp.RMS | dsp.MovingAverage | dsp.MovingMaximum | dsp.MovingMinimum | dsp.MovingStandardDeviation | dsp.MovingVariance | dsp.MedianFilter

Blocks

`Median Filter` | `Moving Average` | `Moving Maximum` | `Moving Minimum` | `Moving RMS` | `Moving Standard Deviation` | `Moving Variance` | `RMS`

Topics

"What Are Moving Statistics?"
"Signal Statistics"
"Sliding Window Method and Exponential Weighting Method"
"Energy Detection in the Time Domain"

Introduced in R2016b

reset

System object: dsp.MovingRMS
Package: dsp

Reset internal states of System object

Syntax

`reset(movRMS)`

Description

`reset(movRMS)` resets the states of the input object to their initial values. The initial state values correspond to the initial conditions for the difference equation defining the object. After you call the object using nonzero input data, the filter might have nonzero states. If you continue to pass nonzero input data to the object without first calling `reset`, the object might produce different outputs for an identical input.

Introduced in R2016b

step

System object: dsp.MovingRMS
Package: dsp

Moving RMS of input signal

Syntax

```
y = step(movRMS,x)
```

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(movRMS,x)` computes the moving RMS of the input signal, `x`, using either the sliding window method or exponential weighting method of the input `dsp.MovingRMS` System object, `movRMS`. The input `x` can be a row vector, a column vector, or a matrix. If `x` is a matrix, each column is treated as an independent channel. The moving RMS is computed along each channel. The input `x` can be complex and supports single and double data types.

Introduced in R2016b

dsp.MovingStandardDeviation System object

Package: dsp

Moving standard deviation

Description

The `dsp.MovingStandardDeviation` System object computes the moving standard deviation of the input signal along each channel, independently over time. The object uses either the sliding window method or the exponential weighting method to compute the moving standard deviation. In the sliding window method, a window of specified length is moved over the data, sample by sample, and the object computes the standard deviation over the data in the window. In the exponential weighting method, the object computes the exponentially weighted moving variance, and takes the square root. For more details on these methods, see “Algorithms” on page 2-1276.

The object accepts multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$, and $n > 1$. The object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel. However, the number of channels cannot change. This object supports C and C++ code generation.

To compute the moving standard deviation of the input:

1. Create a `dsp.MovingStandardDeviation` object and set the properties of the object.
2. Call step to compute the moving standard deviation.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`MovStd = dsp.MovingStandardDeviation` returns a moving standard deviation object, `MovStd`, using the default properties.

`MovStd = dsp.MovingStandardDeviation(Len)` sets the `WindowLength` property to `Len`.

`MovStd = dsp.MovingStandardDeviation(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

Example:

```
MovStd = dsp.MovingStandardDeviation('Method','Exponential weighting','ForgettingFactor
```

Properties

`Method` — Moving standard deviation method

`'Sliding window'` (default) | `'Exponential weighting'`

- `'Sliding window'` — A window of length specified by `SpecifyWindowLength` is moved over the input data along each channel. For every sample the window moves by, the object computes the standard deviation over the data in the window.
- `'Exponential weighting'` — The object computes the exponentially weighted moving variance, and takes the square root.

For more details on these methods, see “Algorithms” on page 2-1276.

`SpecifyWindowLength` — Specify window length

true (default) | false

Flag to specify a window length, specified as a scalar boolean.

- `true` — The length of the sliding window is equal to the value you specify in the `WindowLength` property.
- `false` — The length of the sliding window is infinite. In this mode, the standard deviation is computed using the current sample and all the past samples.

This property applies when you set `Method` to `'Sliding window'`.

`WindowLength` — Length of the sliding window

4 (default) | positive scalar integer

Length of the sliding window, specified as a positive scalar integer. This property applies when you set `Method` to `'Sliding window'` and `SpecifyWindowLength` to `true`.

`ForgettingFactor` — Exponential weighting factor

0.9 (default) | positive real scalar in the range (0,1]

Exponential weighting factor, specified as a positive real scalar in the range (0,1].

This property applies when you set `Method` to `'Exponential weighting'`.

This property is tunable. You can change its value even when the object is locked.

Methods

reset	Reset internal states of System object
step	Moving standard deviation of input signal

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Moving Standard Deviation of Noisy Square Wave Signal

Compute the moving standard deviation of a noisy square wave signal with varying amplitude using the `dsp.MovingStandardDeviation` object.

Initialization

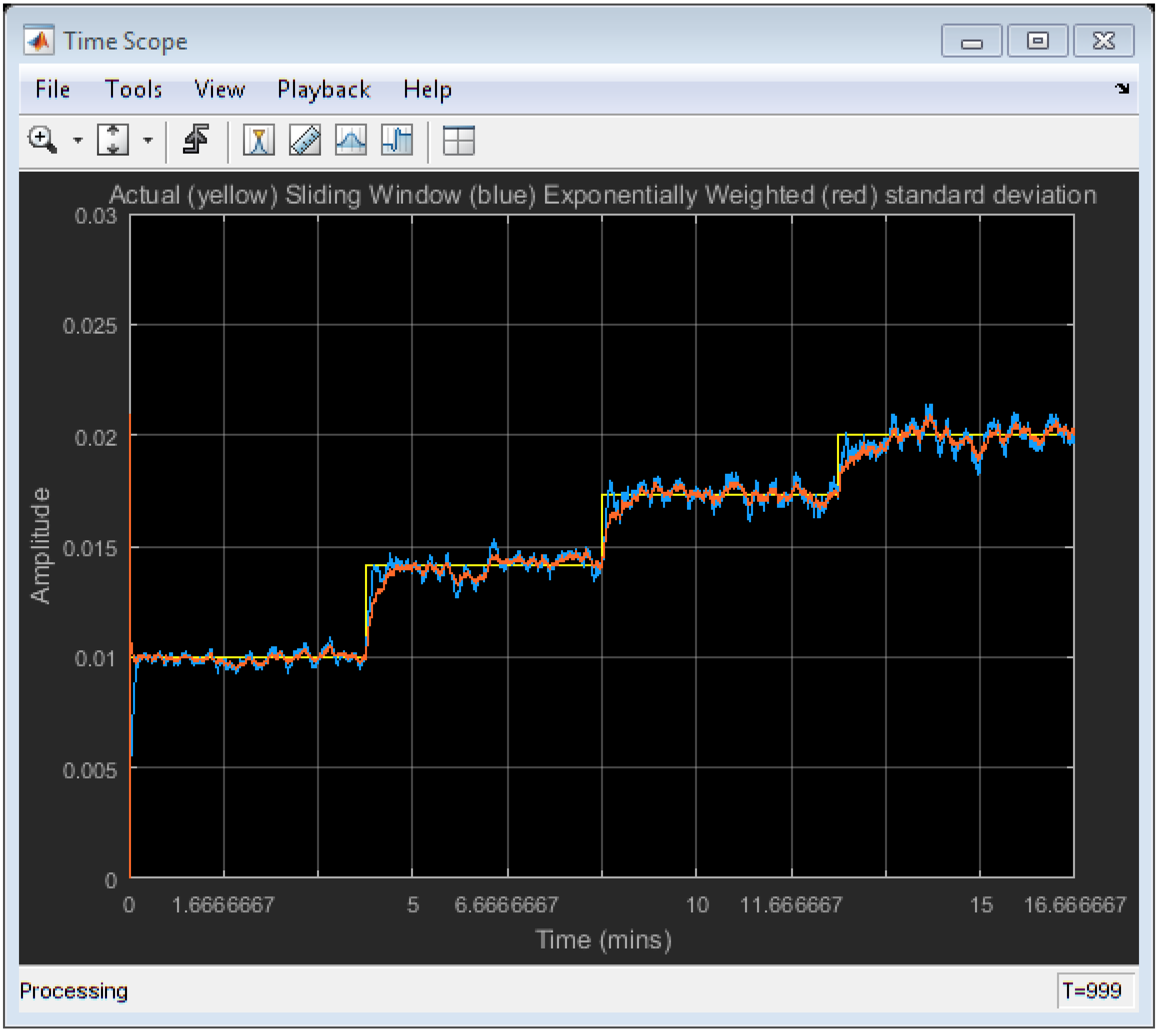
Set up `movstdWindow` and `movstdExp` objects. `movstdWindow` uses the sliding window method with a window length of 800. `movstdExp` uses the exponential weighting method with a forgetting factor of 0.999. Create a time scope for viewing the output.

```
FrameLength = 100;
Fs = 100;
movstdWindow = dsp.MovingStandardDeviation(800);
movstdExp = dsp.MovingStandardDeviation('Method','Exponential weighting',...
    'ForgettingFactor',0.999);
scope  = dsp.TimeScope('SampleRate',Fs,...
    'TimeSpanOverrunAction','Scroll',...
    'TimeSpan',1000,...
    'ShowGrid',true,...
    'BufferLength',1e7,...
    'YLimits',[0 3e-2]);
title = 'Actual (yellow) Sliding Window (blue) Exponentially Weighted (red) standard de
scope.Title = title;
```

Compute the Standard Deviation

Generate a noisy square wave signal. Vary the amplitude of the square wave after a given number of frames. Apply the sliding window method and the exponential weighting method on this signal. The actual standard deviation is `sqrt(np)`. This value is used while adding noise to the data. Compare the actual standard deviation with the computed standard deviation on the time scope.

```
count = 1;
noisepower = 1e-4 * [1 2 3 4];
index = 1;
for index = 1:length(noisepower)
    np = noisepower(index);
    yexp = sqrt(np)*ones(FrameLength,1);
    for i = 1:250
        x = sqrt(np) * randn(FrameLength,1);
        y1 = movstdWindow(x);
        y2 = movstdExp(x);
        scope([yexp,y1,y2]);
    end
end
```



- “What Are Moving Statistics?”
- “Sliding Window Method and Exponential Weighting Method”
- “Signal Statistics”

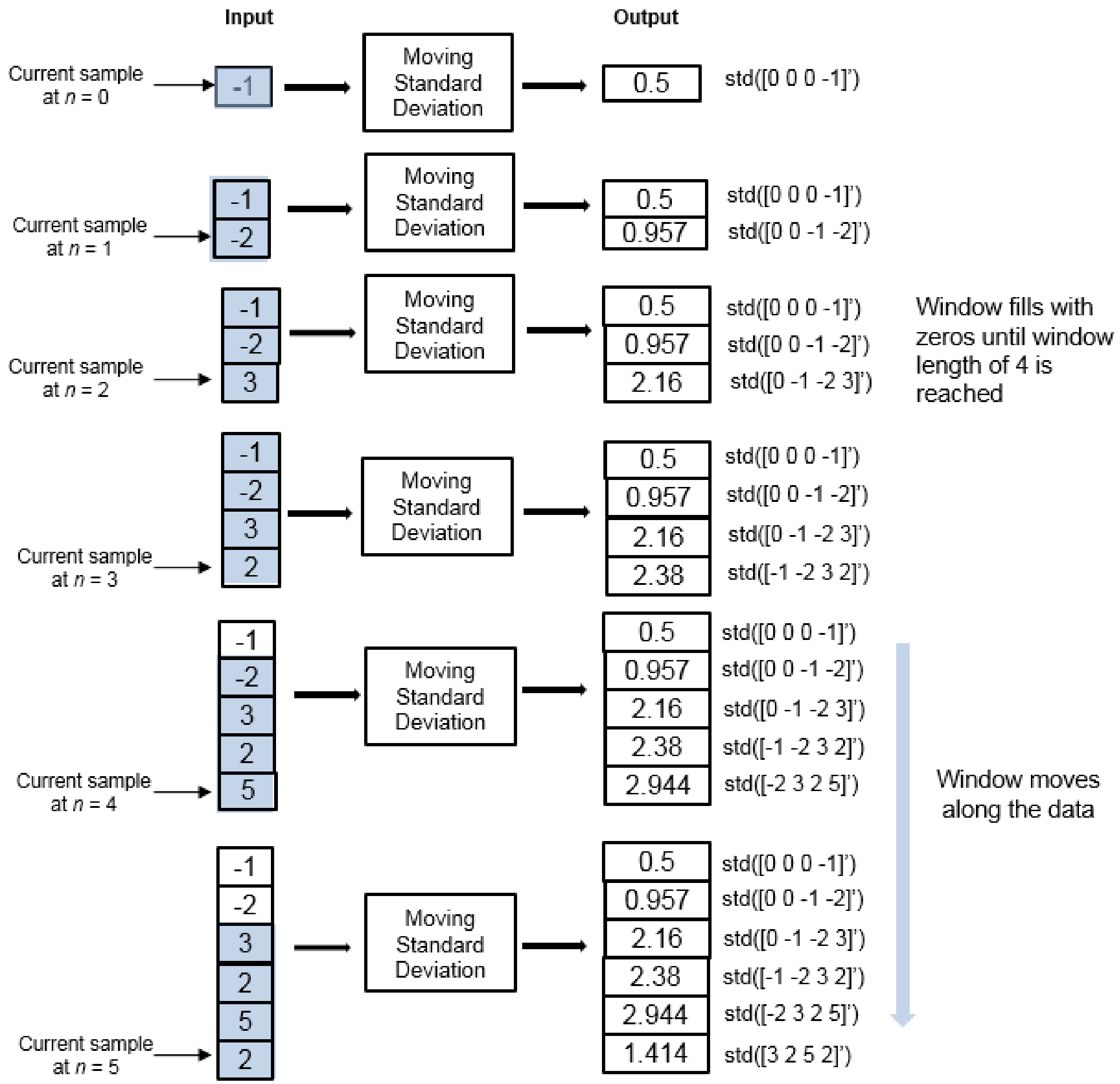
Algorithms

Sliding Window Method

In the sliding window method, the output at the current sample is the standard deviation of the current sample with respect to the data in the window. To compute the first *Len* - 1 outputs, when the window does not have enough data yet, the algorithm fills the window with zeros. As an example, to compute the standard deviation when the second input sample comes in, the algorithm fills the window with *Len* - 2 zeros. *Len* is the length of the window. The data vector, *x*, is then the two data samples followed by *Len* - 2 zeros.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving standard deviation of the current sample with respect to all the previous samples in the channel.

Consider an example of computing the moving standard deviation of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.


Input
Output
Current sample at n = 0
-1
Moving Standard Deviation
0.5
std([0 0 0 -1]')
Current sample at n = 1
-1
-2
0.5
0.957
std([0 0 0 -1]')
std([0 0 -1 -2]')
Current sample at n = 2
3
2.16
std([0 -1 -2 3]')
Window fills with zeros until window length of 4 is reached
Current sample at n = 3
2
2.38
std([-1 -2 3 2]')
Current sample at n = 4
5
2.944
std([-2 3 2 5]')
Window moves along the data
Current sample at n = 5
1.414
std([3 2 5 2]')

Exponential Weighting Method

In the exponential weighting method, the moving standard deviation is computed recursively using these formulas:

$$s_{N,\lambda} = \sqrt{\frac{1}{v_{N,\lambda}} \sum_{k=1}^{N} \lambda^{N-k} \left[x_k - \bar{x}_{N,\lambda} \right]^2},$$

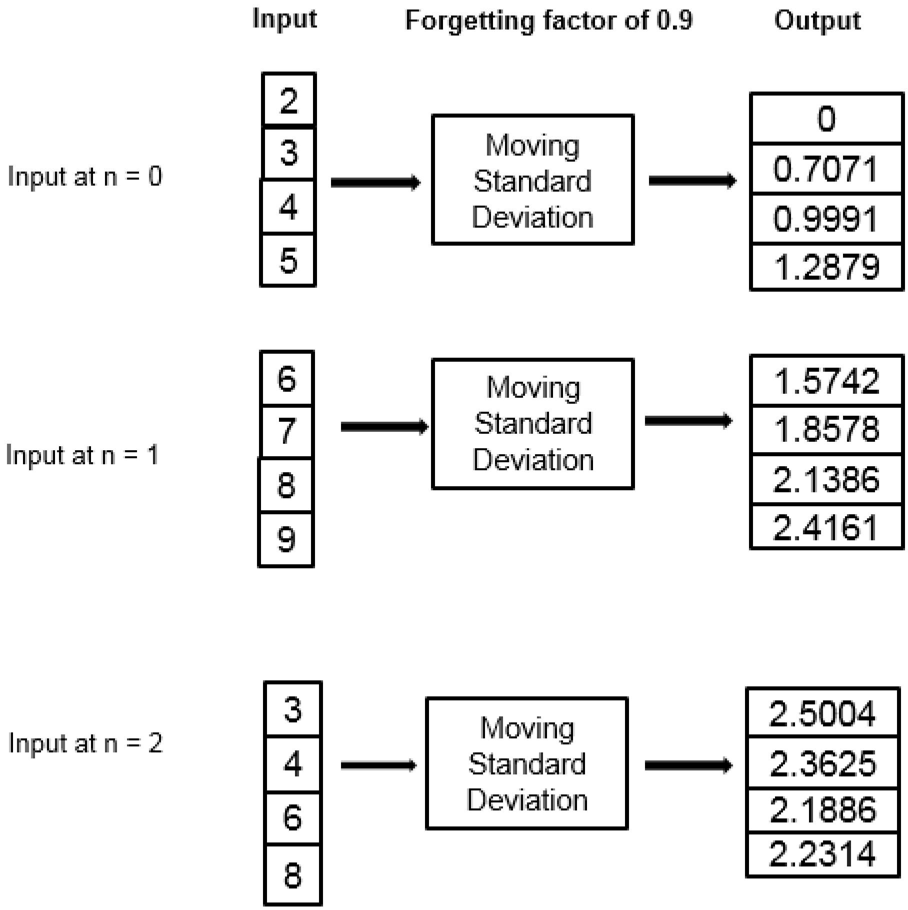
$$v_{N,\lambda} = \frac{2\lambda(1-\lambda^{N-1})}{(1-\lambda)(1+\lambda)}$$

- $s_{N,\lambda}$ — Moving standard deviation of the current data sample with respect to the rest of the data.
- $\left[x_k - \bar{x}_{N,\lambda} \right]^2$ — Difference between each data sample and the average of the data, squared.
- $\sum_{k=1}^{N} \lambda^{N-k} \left[x_k - \bar{x}_{N,\lambda} \right]^2$ — Difference between each data sample and the average of the data, squared and multiplied with the forgetting factor. All the squared terms are added.
- $\frac{1}{v_{N,\lambda}}$ — Weighting factor applied to the sum.
- λ — Forgetting factor.

As the age of the data increases, the magnitude of the weighting factor decreases exponentially and never reaches zero. In other words, the recent data has more influence on the current standard deviation than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All previous samples are given an equal weight.

Consider an example of computing the moving standard deviation using the exponential weighting method. The forgetting factor is 0.9.



References

[1] Bodenham, Dean. “Adaptive Filtering and Change Detection for Streaming Data.” PH.D. Thesis. Imperial College, London, 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

System Objects

dsp.StandardDeviation | dsp.MovingMaximum | dsp.MovingMinimum | dsp.MovingAverage | dsp.MovingRMS | dsp.MovingVariance | dsp.MedianFilter

Blocks

`Median Filter` | `Moving Average` | `Moving Maximum` | `Moving Minimum` | `Moving RMS` | `Moving Standard Deviation` | `Moving Variance` | `Standard Deviation`

Topics

"What Are Moving Statistics?"
"Sliding Window Method and Exponential Weighting Method"
"Signal Statistics"

Introduced in R2016b

reset

System object: dsp.MovingStandardDeviation
Package: dsp

Reset internal states of System object

Syntax

```
reset(movStd)
```

Description

`reset(movStd)` resets the states of the input object to their initial values. The initial state values correspond to the initial conditions for the difference equation defining the object. After you call the object using nonzero input data, the filter might have nonzero states. If you continue to pass nonzero input data to the object without first calling `reset`, the object might produce different outputs for an identical input.

Introduced in R2016b

step

System object: dsp.MovingStandardDeviation
Package: dsp

Moving standard deviation of input signal

Syntax

```
y = step(movStd,x)
```

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(movStd,x)` computes the moving standard deviation of the input signal, `x`, using either the sliding window method or exponential weighting method of the input `dsp.MovingStandardDeviation` System object, `movStd`. The input `x` can be a row vector, a column vector, or a matrix. If `x` is a matrix, each column is treated as an independent channel. The moving standard deviation is computed along each channel. The input `x` can be complex and supports single and double data types.

Introduced in R2016b

dsp.MovingVariance System object

Package: dsp

Moving variance

Description

The `dsp.MovingVariance` System object computes the moving variance of the input signal along each channel, independently over time. The object uses either the sliding window method or the exponential weighting method to compute the moving variance. In the sliding window method, a window of specified length is moved over the data, sample by sample, and the variance is computed over the data in the window. In the exponential weighting method, the object subtracts each sample of the data from the average, squares the difference, and multiplies the squared result with a weighting factor. The object then computes the variance by adding all the weighted data. For more details on these methods, see "Algorithms" on page 2-1268.

The object accepts multichannel inputs, that is, m-by-n size inputs, where $m \geq 1$, and $n > 1$. The object also accepts variable-size inputs. Once the object is locked, you can change the size of each input channel. However, the number of channels cannot change. This object supports C and C++ code generation.

To compute the moving variance of the input:

1. Create a `dsp.MovingVariance` object and set the properties of the object.
2. Call step to compute the moving variance.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`MovVar = dsp.MovingVariance` returns a moving variance object, `MovVar`, using the default properties.

`MovVar = dsp.MovingVariance(Len)` sets the `WindowLength` property to `Len`.

`MovVar = dsp.MovingVariance(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

Example:

```
MovVar = dsp.MovingVariance('Method','Exponential weighting','ForgettingFactor',0.9);
```

Properties

`Method` — Method to compute the variance

`'Sliding window'` (default) | `'Exponential weighting'`

- `'Sliding window'` — A window of length specified by `SpecifyWindowLength` is moved over the input data along each channel. For every sample the window moves by, the object computes the variance over the data in the window.
- `'Exponential weighting'` — The object subtracts each sample of the data from the average, squares the difference, and multiplies the squared result with a weighting factor. The object then computes the variance by adding all the weighted data. The magnitude of the weighting factors decreases exponentially as the age of the data increases, never reaching zero.

For more details on these methods, see “Algorithms” on page 2-1268.

`SpecifyWindowLength` — Specify window length

true (default) | false

Flag to specify a window length, specified as a scalar Boolean.

- `true` — The length of the sliding window is equal to the value you specify in the `WindowLength` property.
- `false` — The length of the sliding window is infinite. In this mode, the variance is computed using the current sample and all past samples.

This property applies when you set `Method` to `'Sliding window'`.

`WindowLength` — Length of the sliding window

4 (default) | positive scalar integer

Length of the sliding window, specified as a positive scalar integer. This property applies when you set `Method` to `'Sliding window'` and `SpecifyWindowLength` to `true`.

`ForgettingFactor` — Exponential weighting factor
0.9 (default) | positive real scalar in the range (0,1]

Exponential weighting factor, specified as a positive real scalar in the range (0,1]. This property applies when you set `Method` to `'Exponential weighting'`.

A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the past samples are given an equal weight.

This property is tunable. You can change its value even when the object is locked.

Methods

reset	Reset internal states of System object
step	Moving Variance of input signal

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Moving Variance of Noisy Square Wave Signal

Compute the moving variance of a noisy square wave signal with varying amplitude using the `dsp.MovingVariance` object.

Initialization

Set up movvarWindow and movvarExp objects. movvarWindow uses the sliding window method with a window length of 800. movvarExp uses the exponentially weighting method with a forgetting factor of 0.999. Create a time scope for viewing the output.

```
FrameLength = 100;
Fs = 100;
movvarWindow = dsp.MovingVariance(800);
movvarExp = dsp.MovingVariance('Method','Exponential weighting',...
    'ForgettingFactor',0.999);
scope  = dsp.TimeScope('SampleRate',Fs,...
    'TimeSpanOverrunAction','Scroll',...
    'TimeSpan',1000,...
    'ShowGrid',true,...
    'BufferLength',1e7,...
    'YLimits',[0 7e-4]);
title = 'Actual (yellow) Sliding Window (blue) Exponentially Weighted (red) variance';
scope.Title = title;
```

Compute the Variance

Generate a noisy square wave signal. Vary the amplitude of the square wave after a given number of frames. Apply the sliding window method and the exponentially weighting method on this signal. The actual variance is *np*. This value is used while adding noise to the data. Compare the actual variance with the computed variances on the time scope.

```
count = 1;
noisepower = 1e-4 * [1 2 3 4];
index = 1;
for index = 1:length(noisepower)
    np = noisepower(index);
    yexp = np*ones(FrameLength,1);
    for i = 1:250
        x = 1 + sqrt(np) * randn(FrameLength,1);
        y1 = movvarWindow(x);
        y2 = movvarExp(x);
        scope([yexp,y1,y2]);
    end
end
```

- “What Are Moving Statistics?”
- “Sliding Window Method and Exponential Weighting Method”
- “Signal Statistics”

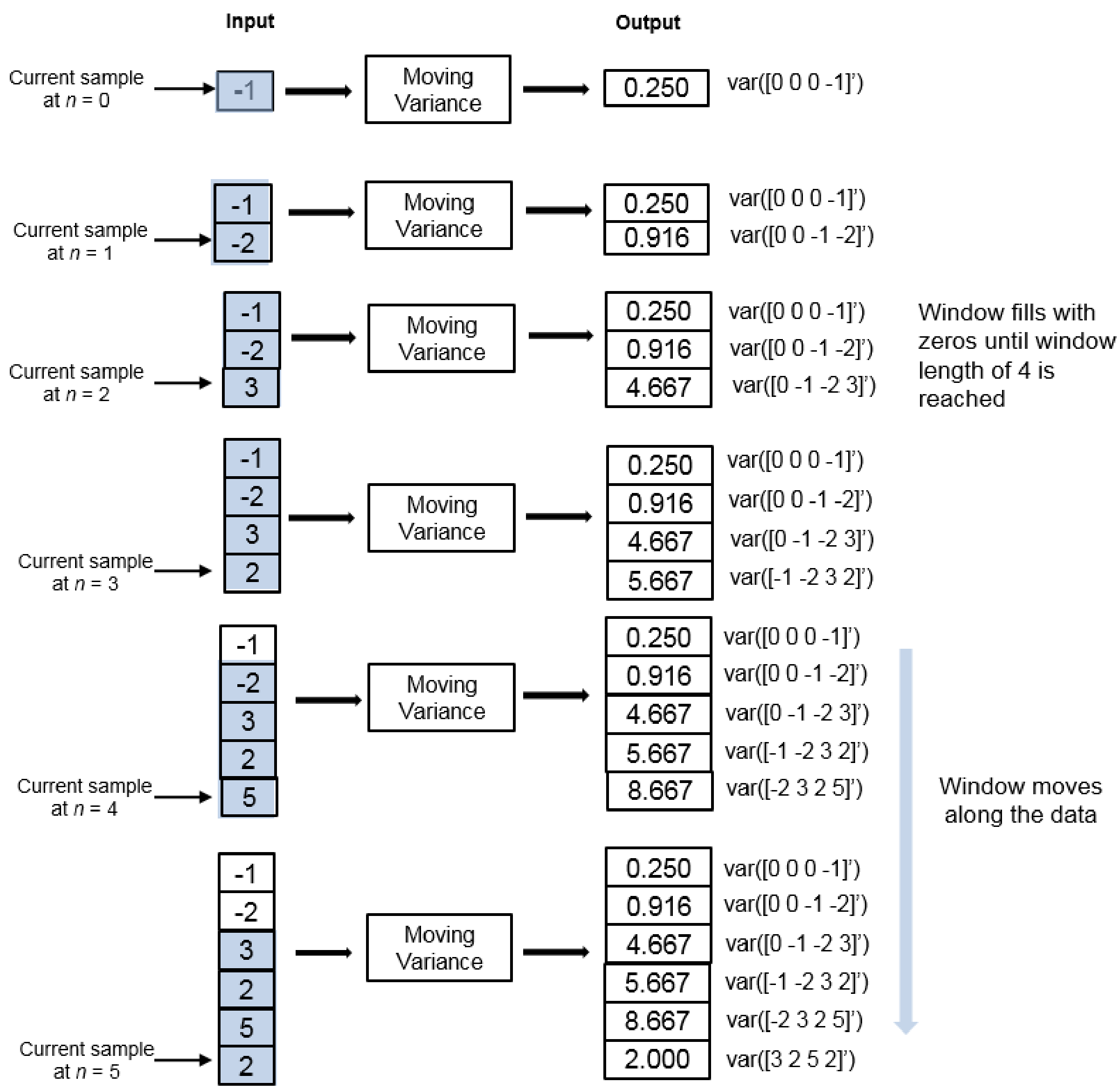
Algorithms

Sliding Window Method

In the sliding window method, the output at the current sample is the variance of the current sample with respect to the data in the window. To compute the first *Len* - 1 outputs, when the window does not have enough data yet, the algorithm fills the window with zeros. As an example, to compute the variance when the second input sample comes in, the algorithm fills the window with *Len* - 2 zeros. *Len* is the length of the window. The data vector, *x*, is then the two data samples followed by *Len* - 2 zeros.

When you do not specify the window length, the algorithm chooses an infinite window length. In this mode, the output is the moving variance of the current sample with respect to all previous samples in the channel.

Consider an example of computing the moving variance of a streaming input data using the sliding window method. The algorithm uses a window length of 4. With each input sample that comes in, the window of length 4 moves along the data.


Input
Output
Current sample at n = 0
-1
Moving Variance
0.250
var([0 0 0 -1]')
Current sample at n = 1
-1
-2
0.250
0.916
var([0 0 0 -1]')
var([0 0 -1 -2]')
Current sample at n = 2
3
4.667
var([0 -1 -2 3]')
Window fills with zeros until window length of 4 is reached
Current sample at n = 3
2
5.667
var([-1 -2 3 2]')
Current sample at n = 4
5
8.667
var([-2 3 2 5]')
Window moves along the data
Current sample at n = 5
2.000
var([3 2 5 2]')

Exponential Weighting Method

In the exponential weighting method, the moving variance is computed recursively using these formulas:

$$s^2_{N,\lambda} = \frac{1}{v_{N,\lambda}} \sum_{k=1}^{N} \lambda^{N-k} \left[x_k - \bar{x}_{N,\lambda} \right]^2,$$

$$v_{N,\lambda} = \frac{2\lambda(1-\lambda^{N-1})}{(1-\lambda)(1+\lambda)}$$

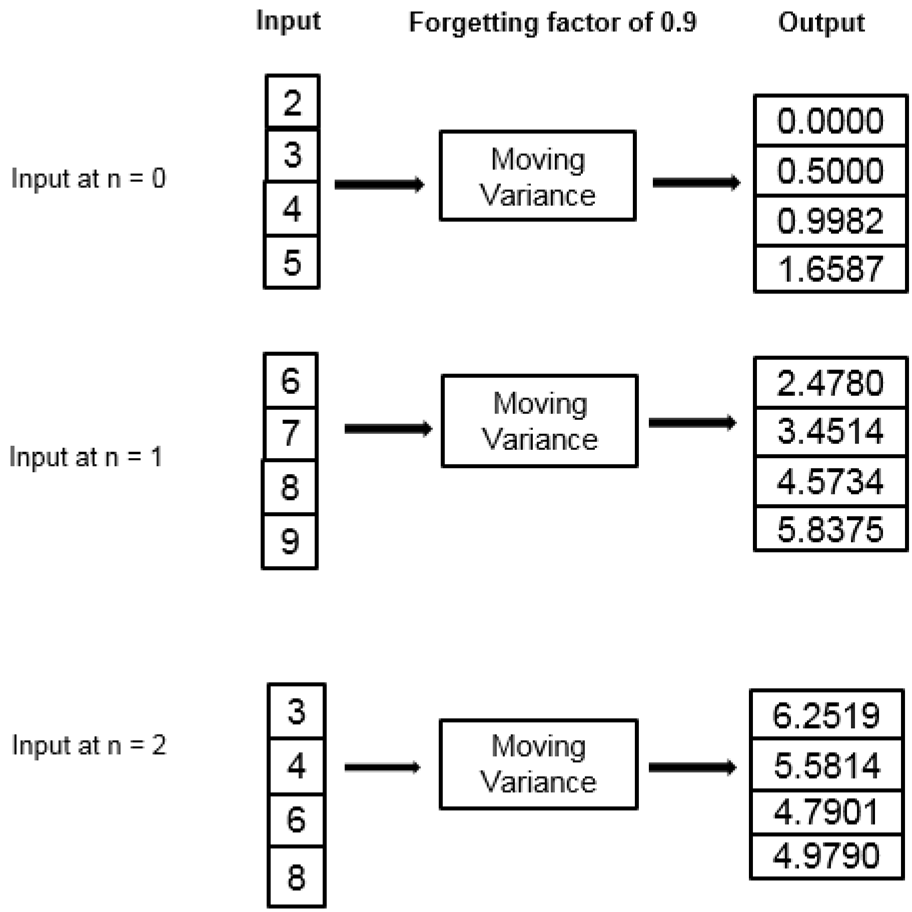
To compute the moving variance, the algorithm implements these equations recursively.

- $s^2_{N,\lambda}$ — Moving variance of the current data sample with respect to the rest of the data in the channel.
- $\bar{x}_{N,\lambda}$ — Moving average at the current sample. For details on computing the moving average, see dsp.MovingAverage.
- $\left[x_k - \bar{x}_{N,\lambda} \right]^2$ — Difference between each data sample and the average of the data, squared.
- $\sum_{k=1}^{N} \lambda^{N-k} \left[x_k - \bar{x}_{N,\lambda} \right]^2$ — Difference between each data sample and the average of the data, squared and multiplied with the forgetting factor. All the squared terms are added.
- $\frac{1}{v_{N,\lambda}}$ — Weighting factor applied to the sum.
- λ — Forgetting factor you can specify through the `ForgettingFactor` property.

As the age of the data increases, the magnitude of the weighting factor decreases exponentially, and never reaches zero. In other words, the recent data has more influence on the current variance, than the older data.

The value of the forgetting factor determines the rate of change of the weighting factors. A forgetting factor of 0.9 gives more weight to the older data than does a forgetting factor of 0.1. A forgetting factor of 1.0 indicates infinite memory. All the past samples are given an equal weight.

Consider an example of computing the moving variance using the exponential weighting method. The forgetting factor is 0.9.



References

[1] Bodenham, Dean. “Adaptive Filtering and Change Detection for Streaming Data.” PH.D. Thesis. Imperial College, London, 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

System Objects

dsp.Variance | dsp.MovingMaximum | dsp.MovingMinimum | dsp.MovingAverage | dsp.MovingRMS | dsp.MovingStandardDeviation | dsp.MedianFilter

Blocks

`Median Filter` | `Moving Average` | `Moving Maximum` | `Moving Minimum` | `Moving RMS` | `Moving Standard Deviation` | `Moving Variance` | `Variance`

Topics

“What Are Moving Statistics?”
“Sliding Window Method and Exponential Weighting Method”
“Signal Statistics”

Introduced in R2016b

reset

System object: dsp.MovingVariance
Package: dsp

Reset internal states of System object

Syntax

```
reset(movVar)
```

Description

`reset(movVar)` resets the states of the input object to their initial values. The initial state values correspond to the initial conditions for the difference equation defining the object. After you call the object using nonzero input data, the filter might have nonzero states. If you continue to pass nonzero input data to the object without first calling `reset`, the object might produce different outputs for an identical input.

Introduced in R2016b

step

System object: dsp.MovingVariance
Package: dsp

Moving Variance of input signal

Syntax

```
y = step(movVar,x)
```

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(movVar,x)` computes the moving variance of the input signal, `x`, using either the sliding window method or exponential weighting method of the input `dsp.MovingVariance` System object, `movVar`. The input `x` can be a row vector, a column vector, or a matrix. If `x` is a matrix, each column is treated as an independent channel. The moving variance is computed along each channel. The input `x` can be complex and supports single and double data types.

Introduced in R2016b

dsp.NCO System object

Package: dsp

Generate real or complex sinusoidal signals

Description

The numerically controlled oscillator, or `NCO`, object generates real or complex sinusoidal signals. The amplitude of the generated signal is always 1.

To generate real or complex sinusoidal signals:

1. Define and set up your NCO System object. See "Construction" on page 4-1363.
2. Call step to generate the signals according to the properties of `dsp.NCO`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`nco = dsp.NCO` returns an NCO System object, `nco`, that generates a multichannel real or complex sinusoidal signal, with independent frequency and phase in each output channel.

`nco = dsp.NCO('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an `NCO` System object, `nco`, with each specified property set to the specified value.

Properties

`PhaseIncrementSource`

Source of phase increment

Indicate how to specify the phase increment: `Property` or `Input port`. The default is `Input port`.

PhaseIncrement

Phase increment

Specify the phase increment as an integer scalar. This property applies only when you set the `PhaseIncrementSource on page 4-1363` property to `Property`. The default is `100`.

PhaseOffsetSource

Source of phase offset

Specify the phase offset as `Property` or `Input port`. The default is `Property`.

PhaseOffset

Phase offset

Specify the phase offset as an integer scalar. This property applies only when you set the `PhaseOffsetSource on page 4-1364` property to `Property`. The default is `0`.

Dither

Enable adding internal dithering to NCO algorithm

Set this property to `true` to add internal dithering to the NCO algorithm. Dithering is added using the PN Sequence Generator from the Communications System Toolbox product. The default is `true`.

NumDitherBits

Number of dither bits

Specify the number of dither bits as a positive integer. This property applies only when you set the `Dither` property to `true`. The default is `4`.

PhaseQuantization

Enable quantization of accumulated phase

Set this property to `true` to enable quantization of the accumulated phase. The default is `true`.

NumQuantizerAccumulatorBits

Number of quantizer accumulator bits

Specify the number of quantizer accumulator bits as an integer scalar greater than 1 and less than the accumulator word length. This property determines the number of entries in the lookup table of sine values. This property applies only when you set the `PhaseQuantization` on page 4-1364 property to `true`. The default is `12`.

PhaseQuantizationErrorOutputPort

Enable output of phase quantization error

Set this property to `true` to output the phase quantization error. This property applies only when you set the `PhaseQuantization` on page 4-1364 property to `true`. The default is `false`.

Waveform

Type of output signal

Specify the type of the output signal: `Sine`, `Cosine`, `Complex exponential` or `Sine and cosine`. The default is `Sine`.

SamplesPerFrame

Number of output samples per frame

Specify the number of samples per frame of the output signal. This property applies only when you set the `PhaseOffsetSource` on page 4-1364 property to `Property`. The default is `1`. When the `PhaseOffsetSource` property is `Input port`, and the `PhaseIncrementSource` on page 4-1363 property is `Property`, the number of rows or frame size of the phase offset input determines the number of samples per frame of the output signal. When you set both the `PhaseOffsetSource` on page 4-1364 and `PhaseIncrementSource` on page 4-1363 properties to `Input port`, the number of rows in the inputs must be 1, and the samples per frame of the output signal is 1.

OutputDataType

Output data type

Specify the output data type as `double`, `single` or `Custom`. The default is `Custom`. When you select `Custom` you must also set the `CustomOutputDataType` property.

You can find the CustomOutputDataType property described under Fixed-Point Properties on this page.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

This constant property has a value Floor.

OverflowAction

Overflow action for fixed-point operations

This constant property has a value Wrap.

AccumulatorDataType

Accumulator word and fraction lengths

This constant property has a value Custom.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as an unscaled numerictype object with a Signedness of Auto. The default is numerictype([],16).

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the OutputDataType on page 4-1365 property to Custom. The default is numerictype([],16,14).

Methods

info	Characteristic information about generated signal

reset	Reset accumulator of NCO object
step	Generate multichannel real or complex sinusoidal signal using NCO (Numerically Controlled Oscillator)

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Design an NCO Source

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject() becomes step(myObject).

Design an NCO source according to given specifications.

```
df = 0.05; % Frequency resolution = 0.05 Hz
minSFDR = 96; % Spurious free dynamic range >= 96 dB
Ts = 1/8000; % Sample period = 1/8000 sec
dphi = pi/2; % Desired phase offset = pi/2;
```

Calculate number of accumulator bits required for the given frequency resolution.

```
Nacc = ceil(log2(1/(df*Ts)));
```

Actual frequency resolution achieved.

```
actdf = 1/(Ts*2^Nacc);
```

Calculate number of quantized accumulator bits required from the SFDR requirement

```
Nqacc = ceil((minSFDR-12)/6);
```

Calculate the phase offset

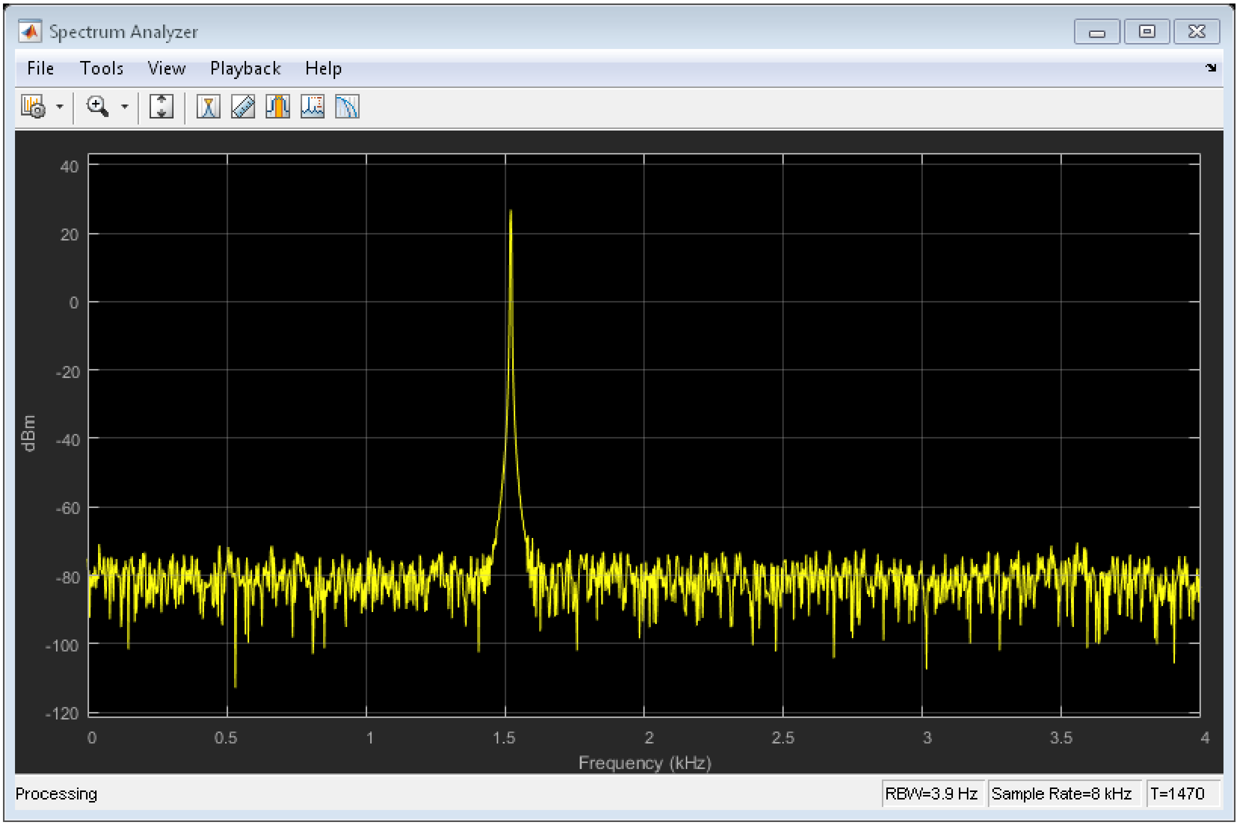
```
phOffset = 2^Nacc*dphi/(2*pi);
```

Design the NCO source.

```
nco = dsp.NCO('PhaseOffset', phOffset,...
    'NumDitherBits', 4, ...
    'NumQuantizerAccumulatorBits', Nqacc,...
    'SamplesPerFrame', 1/Ts, ...
    'CustomAccumulatorDataType', numerictype([],Nacc));
san = dsp.SpectrumAnalyzer('SampleRate', 1/Ts, ...
    'PlotAsTwoSidedSpectrum', false);
```

View the output of the NCO source on a spectrum analyzer. Change the output frequency in the middle of the simulation from 510 Hz to 1520 Hz.

```
tic;
while toc < 10

    if toc < 5
        F0 = 510;
    else
        F0 = 1520;
    end
    % Calculate the phase increment
    phIncr = int32(round(F0*Ts*2^Nacc));
    y = nco(phIncr);
    san(y);
end
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `NCO` block reference page. The object properties correspond to the block properties, except:
There is no object property that corresponds to the **Sample time** block parameter. The objects assumes a sample time of one second.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.SineWave

Introduced in R2012a

info

System object: dsp.NCO
Package: dsp

Characteristic information about generated signal

Syntax

```
S = info(nco)
```

Description

`S = info(nco)` returns a structure containing characteristic information, `S`, about the signal being generated by the `NCO` System object, `nco`. The number of fields of `S`, and their values vary depending on the property value settings of `nco`. For description of the possible fields and their values, see the following table.

Field	Value
`NumPointsLUT`	Number of data points for lookup table. The lookup table is implemented as a quarter-wave sine table.
`SineLUTSize`	Quarter-wave sine lookup table size in bytes.
`TheoreticalSFDR`	Theoretical spurious free dynamic range (SFDR) in dBc. This field applies when you set the `PhaseQuantization` property to `true`.
`FrequencyResolution`	Frequency resolution of the `NCO` in Hz. The sample time of the output signal is assumed to be `1` sec.

reset

System object: dsp.NCO
Package: dsp

Reset accumulator of NCO object

Syntax

```
reset(nco)
```

Description

`reset(nco)` resets the accumulator of the `NCO` System object, `nco`, to zero.

step

System object: dsp.NCO
Package: dsp

Generate multichannel real or complex sinusoidal signal using NCO (Numerically Controlled Oscillator)

Syntax

```
Y = step(nco,INC)
Y = step(nco)
Y = step(nco,OFFSET)
Y = step(nco,INC,OFFSET)
[Y,COSINE] = step(nco,___)
[Y,QERR] = step(nco,___)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`Y = step(nco,INC)` returns a sinusoidal signal, `Y` , generated by the `NCO` with the specified phase increment, `INC`. `INC` must be a built-in integer or a fi object comprising either a scalar or a row vector, where each row element corresponds to a separate channel.

`Y = step(nco)` when the `PhaseIncrementSource` and `PhaseOffsetSource` properties are both `Property`, returns a sinusoidal signal, `Y`.

`Y = step(nco,OFFSET)` returns a sinusoidal signal, `Y`, with phase offset, `OFFSET`, when the `PhaseOffsetSource` property is `Input port`. `OFFSET` must be a built-in integer or a fi object. The number of rows of `OFFSET` determines the number of samples per frame of the output signal. The number of columns of the `OFFSET` determines the number of channels of the output signal.

Y = step(nco,INC,OFFSET) returns a sinusoidal signal, Y, with phase increment, INC, and phase offset, OFFSET, when the PhaseIncrementSource and the PhaseOffsetSource properties are both Input port. INC and OFFSET must both be row vectors of the same length, where the length determines the number of channels in the output signal.

[Y,COSINE] = step(nco,___) returns a sinusoidal signal, Y, and a cosinusoidal signal, COSINE, when the Waveform property is set to Sine and cosine. This syntax can include any of the input arguments in previous syntaxes.

[Y,QERR] = step(nco,___)when the PhaseQuantization and the PhaseQuantizationErrorOutputPort properties are both true, returns a sinusoidal signal, Y, and output quantization error, QERR.

Note: obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

---

dsp.Normalizer System object

Package: dsp

Vector normalization along specified dimension

Description

The `Normalizer` object performs vector normalization along rows, columns, or specified dimension.

To perform vector normalization:

1. Define and set up your normalization object. See "Construction" on page 4-1375.
2. Call step to perform vector normalization according to the properties of `dsp.Normalizer`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`norm = dsp.Normalizer` returns a normalization object, `norm`, that normalizes the input over each column by the squared 2-norm of the column plus a bias term of 1e-10 used to protect against divide-by-zero.

`norm = dsp.Normalizer('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a normalization object, `norm`, with each property set to the specified value.

Properties

`Method`

Type of normalization to perform

Specify the type of normalization to perform as `2-norm` or `Squared 2-norm`. The `2-norm` mode supports floating-point signals only. The `Squared 2-norm` supports both fixed-point and floating-point signals. The default is `Squared 2-norm`.

Bias

Real number added in denominator to avoid division by zero

Specify the real number to add in the denominator to avoid division by zero. The default is 1e-10. This property is tunable.

Dimension

Dimension to operate along

Specify whether to normalize along `Column` , `Row`, or `Custom`. The default is `Column`.

CustomDimension

Numerical dimension to operate along

Specify the one-based value of the dimension over which to normalize. The value of this parameter cannot exceed the number of dimensions in the input signal. This property applies when `Dimension` on page 4-1376 property is `Custom`. The default is 1.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling`, `Convergent`, `Floor` , `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` or `Saturate`. The default is `Wrap`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of `Same as input` or `Custom`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` on page 4-1376 property to `Custom`. The default to `numerictype([],32,32)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Same as product`, `Same as input`, `Custom`. The default is `Same as product`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` on page 4-1377 property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of `Same as accumulator`, `Same as product` , `Same as input`, `Custom`. The default is `Same as product`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` on page 4-1377 property to `Custom`. The default is `numerictype([],32,32)`.

Methods

step	Normalize input along specified dimension

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Normalize a Matrix

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
norm = dsp.Normalizer;
x = magic(3);
y = norm(x);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Normalization` block reference page. The object properties correspond to the block parameters, except:

- **Treat sample-based row input as column** — The block allows you to input a row vector and normalize the row vector as a column vector. The normalization object always normalizes along the value of the `Dimension` on page 4-1376 property.
- The normalization object does not support the **Minimum** and **Maximum** options for data output.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.ArrayVectorMultiplier

Introduced in R2012a

step

System object: dsp.Normalizer
Package: dsp

Normalize input along specified dimension

Syntax

`Y = step(norm,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(norm,X)` returns a normalized output`Y` . The input `X` must be floating-point for the `2-norm` mode, and either fixed-point and floating-point for the `Squared 2-norm` mode.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.NotchPeakFilter System object

Package: dsp

Second-order tunable notching and peaking IIR filter

Description

The `NotchPeakFilter` object filters each channel of the input using IIR filter implementation.

To filter each channel of the input:

1. Define and set up your NotchPeak filter. See "Construction" on page 4-1381.
2. Call step to filter each channel of the input according to the properties of `dsp.NotchPeakFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`npFilter = dsp.NotchPeakFilter` returns a second-order notching and peaking IIR filter which independently filters each channel of the input over time, using a specified center frequency and 3 dB bandwidth. Both of these properties are specified in Hz and are tunable. Both of these values must be scalars between `0` and half the sample rate.

`npFilter = dsp.NotchPeakFilter('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a notch filter with each specified property name set to the specified value. You can specify additional name-value pair arguments in any order as `(Name1,Value1,...,NameN,ValueN)`.

`npFilter = dsp.NotchPeakFilter('specification','Quality factor and center frequency')` specifies the quality factor (Q factor) of the notch or peak filter instead of the 3 dB bandwidth. The Q factor is defined as the center frequency divided by the bandwidth. A higher Q factor corresponds to a narrower notch or peak. The Q factor should be a scalar value greater than 0. The Q factor is tunable.

`npFilter = dsp.NotchPeakFilter('Specification','Coefficients')` specifies the coefficient values that affect bandwidth and center frequency directly, rather than specifying the design parameters in Hz. This removes the trigonometry calculations involved when the properties are tuned. The `CenterFrequencyCoefficients` should be a scalar between `-1` and `1`, with `-1` corresponding to `0` Hz and `1` corresponding to the Nyquist frequency. The `BandwidthCoefficient` should be a scalar between `-1` and `1`, with `-1` corresponding to the largest `3` dB bandwidth and `1` corresponding to the smallest `3` dB bandwidth. Both coefficient values are tunable.

Properties

`Specification`

Filter specification

Set the specification as one of `'Bandwidth and center frequency'` | `'Quality factor and center frequency'` | `'Coefficients'`. The default is `'Bandwidth and center frequency'`.

`Bandwidth`

3 dB bandwidth

Specify the filter's `3` dB bandwidth as a finite positive numeric scalar in Hertz. This property is applicable only if `specification` is `'Bandwidth and center frequency'`. The default is `2205` Hz. This property is tunable.

`CenterFrequency`

Notch or Peak center frequency

Specify the filter's center frequency (for both the notch and the peak) as a finite positive numeric scalar in Hertz. This property is applicable only if `specification` is `'Bandwidth and center frequency'` | `'Quality factor and center frequency'`. The default is `11025` Hz. This property is tunable.

QualityFactor

Quality factor for notch or peak filter

Specify the quality factor (Q factor) for both the notch and the peak filters. The Q factor is defined as the center frequency divided by the bandwidth. This property is applicable only if `specification` is set to `'Quality factor and center frequency'`. The default value is `5`. This property is tunable.

SampleRate

Sample rate of input

Specify the sample rate of the input in Hertz as a finite numeric scalar. The default is `44100` Hz.

BandwidthCoefficient

Bandwidth coefficient

Specify the value that determines the filter's `3` dB bandwidth as a finite numeric scalar in the range `[-1 1]`. Where `-1` corresponds to the maximum `3` dB bandwidth (SampleRate/4), and `1` corresponds to the minimum `3` dB bandwidth (0 Hz, an allpass filter). The default is `0.72654`. This property is only applicable if `specification` is set to `'Coefficients'`. This property is tunable.

CenterFrequencyCoefficient

Center frequency coefficient

Specify the coefficient that determines the filter's center frequency as a finite numeric scalar between `-1` and `1`. Where `-1` corresponds to the minimum center frequency (0 Hz), and 1 corresponds to the maximum center frequency (`SampleRate/2` Hz). This property is only applicable if `specification` is set to `'Coefficients'`. The default is 0 which corresponds to `SampleRate/4` Hz. This property is tunable.

Methods

getBandwidth	Get 3 dB bandwidth

getCenterFrequency	Get center frequency
getOctaveBandwidth	Bandwidth in number of octaves
getQualityFactor	Get quality factor
reset	Reset the internal states of notch or peak filter
step	Process input using the notch or peak filter algorithm
tf	Transfer function

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

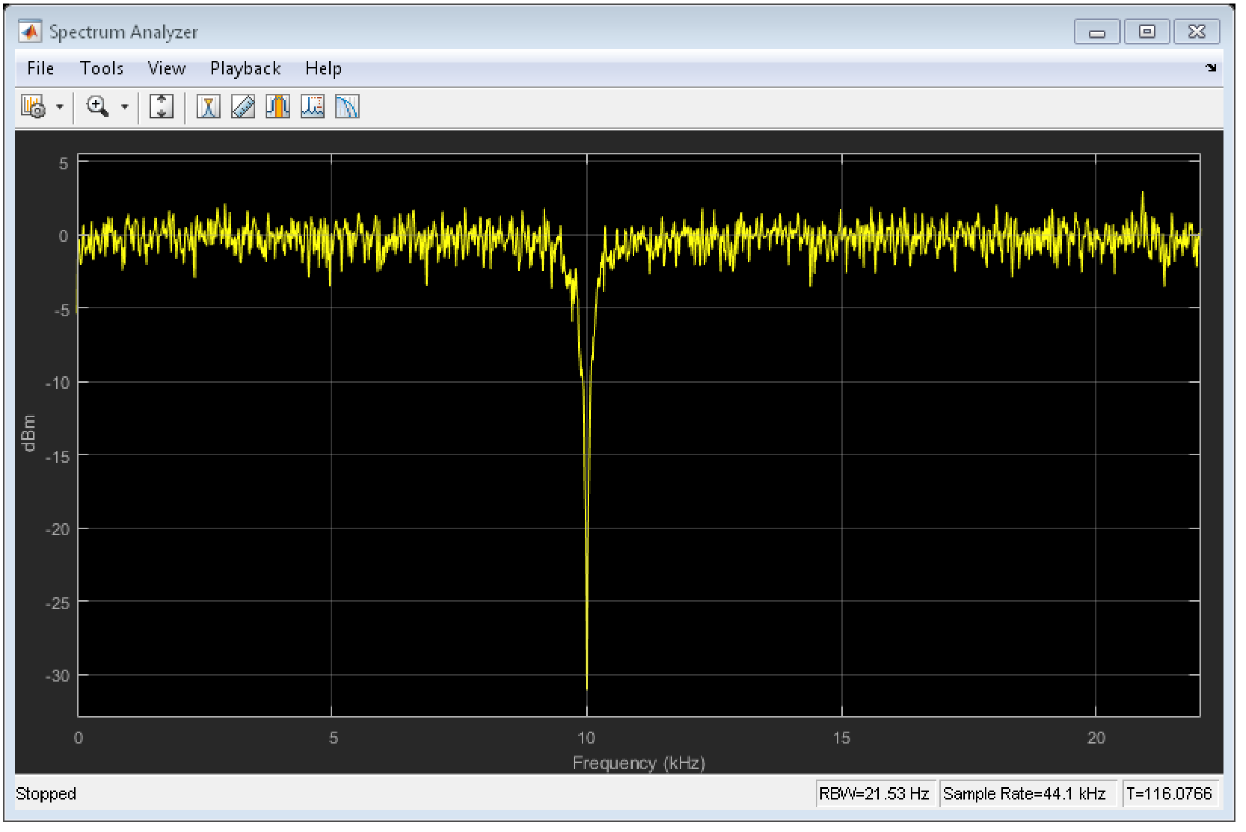
Notch filter

This example shows how to use dsp.NotchPeakFilter as a notch filter with center frequency of 5000 Hz and a 3 dB bandwidth of 500 Hz.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
npFilter = dsp.NotchPeakFilter('CenterFrequency',5000,'Bandwidth',500);
sa =  dsp.SpectrumAnalyzer('SampleRate',44100,...
    'PlotAsTwoSidedSpectrum',false,'SpectralAverages',50);
for i=1:5000
    y = npFilter(randn(1024,1));
    sa(y);
    if (i==2500)
        % Tune center frequency to 10000
        npFilter.CenterFrequency = 10000;
```

```
    end
end
release(npFilter)
release(sa)
```



Algorithms

The design equations for this filter are:

$$H(z) = (1-b)\frac{1-z^{-2}}{1-2b\cos w_0 z^{-1} + (2b-1)z^{-2}}$$

The previous equation is for peak filter, and the following equation is for notch filter.

$$H(z) = b\frac{1 - 2\cos w_0 z^{-1} + z^{-2}}{1 - 2b\cos w_0 z^{-1} + (2b-1)z^{-2}}$$

With

$$b = \frac{1}{1 + \tan(\Delta w / 2)}$$

where $\omega_0 = 2\pi f_0/f_s$ is the center frequency in radians/sample (f_0 is the center frequency in Hz and f_s is the sampling frequency in Hz). $\Delta\omega = 2\pi\Delta f/f_s$ is the 3 dB bandwidth in radians/sample (Δf is the 3 dB bandwidth in Hz). Note that the two filters are complementary:

$$H_{notch}(z) + H_{peak}(z) = 1$$

they can be written as:

$$H_{peak}(z) = \frac{1}{2}[1 - A(z)]$$

$$H_{notch}(z) = \frac{1}{2}[1 + A(z)]$$

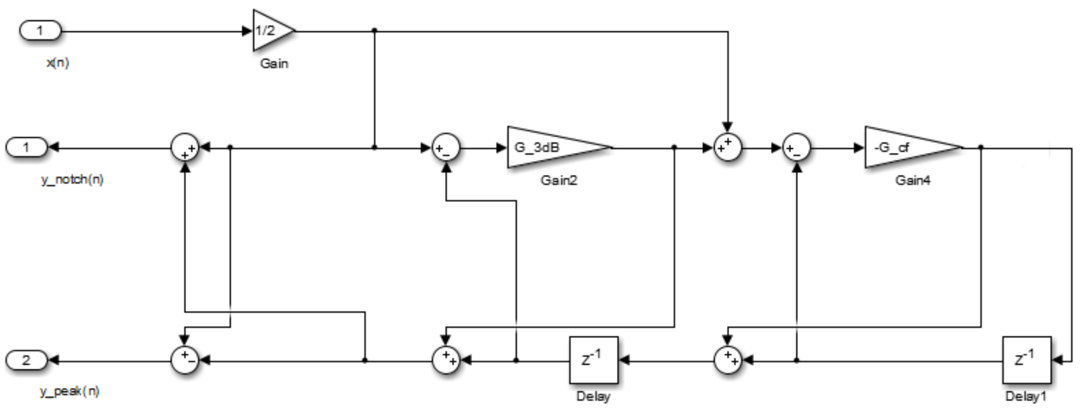
where A(z) is a 2^{nd} order allpass filter.

$$A(z) = \frac{a_2 + a_1 z^{-1} + z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

and

$$a_1 = -2b\cos\omega_0$$
$$a_2 = 2b - 1$$

The filter is implemented as follows:



where

$$G_{3dB} = a_2 = 2b - 1$$

$$G_{cf} = \frac{a_1 - a_1 a_2}{1 - a_2{}^2} = -\cos w_0$$

Notice that G_{cf} depends only on the center frequency, and G_{3dB} depends only on the 3 dB bandwidth.

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

See Also

dsp.BiquadFilter | `iirnotch` | `iirpeak` | `Notch-Peak Filter`

Introduced in R2014a

getBandwidth

System object: dsp.NotchPeakFilter
Package: dsp

Get 3 dB bandwidth

Syntax

```
BW = getBandwidth(npFilter)
```

Description

`BW = getBandwidth(npFilter)` returns the 3 dB bandwidth for the notch or peak filter. If the `Specification` is set to `'Quality factor and center frequency'`, the 3 dB bandwidth is determined from the quality factor value. If the `Specification` property is set to `'Coefficients'`, the 3 dB bandwidth is determined from the `BandwidthCoefficient` value and the sample rate.

getCenterFrequency

System object: dsp.NotchPeakFilter
Package: dsp

Get center frequency

Syntax

```
CF= getCenterFrequency(npFilter)
```

Description

`CF= getCenterFrequency(npFilter)` returns the center frequency of the notch or peak filter. If the `Specification` property is set to `'Coefficients'`, the center frequency is determined from the `CenterFrequencyCoefficient` value and the sample rate.

getOctaveBandwidth

System object: dsp.NotchPeakFilter
Package: dsp

Bandwidth in number of octaves

Syntax

```
N = getOctaveBandwidth(npFilter)
```

Description

`N = getOctaveBandwidth(npFilter)` returns the bandwidth of the notch or peak filter, measured in number of octaves rather than Hz.

getQualityFactor

System object: dsp.NotchPeakFilter
Package: dsp

Get quality factor

Syntax

```
Q = getQualityFactor(npFilter)
```

Description

`Q = getQualityFactor(npFilter)` returns the quality factor (`Q` factor) for both the notch and the peak filters. The `Q` factor is defined as the center frequency divided by the bandwidth.

reset

System object: dsp.NotchPeakFilter
Package: dsp

Reset the internal states of notch or peak filter

Syntax

`reset(npFilter)`

Description

`reset(npFilter)` resets the internal states of System object, `npFilter`, to their initial values. System objects without states are unaffected by reset.

step

System object: dsp.NotchPeakFilter
Package: dsp

Process input using the notch or peak filter algorithm

Syntax

```
Y = step(npFilter,x)
[Yn,Yp] = step(npFilter,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(npFilter,x)` filters each channel of the input signal, `x`, using the notch peak filter, `npFilter`, to produce the notch filter output, `Y`.

`[Yn,Yp] = step(npFilter,x)` filters each channel of the input signal, `x`, using the notch peak filter, `npFilter`, to produce the notch filter output, `Yn`, and peak filter output, `Yp`.

Calling step on an object puts that object into a locked state. When locked, nontunable properties or any input characteristics (size, data type and complexity) cannot change without reinitializing (unlocking and relocking) the object.

tf

System object: dsp.NotchPeakFilter
Package: dsp

Transfer function

Syntax

```
[B,A] = tf(npFilter)
[B,A,B2,A2] = tf(npFilter)
```

Description

`[B,A] = tf(npFilter)` returns the vector of numerator coefficients, `B`, and the vector of denominators, `A`, for the equivalent transfer function corresponding to the notch filter.

In addition to `B` and `A`, `[B,A,B2,A2] = tf(npFilter)` returns the vector of numerator coefficients, `B2`, and the vector of denominator coefficients, `A2`, for the equivalent transfer function corresponding to the peak filter.

Examples

Determine the Transfer Function of Notch Peak Filter

Create a `dsp.NotchPeakFilter` System object™. Obtain the coefficients of the transfer function corresponding to the notch and peak filters.

```
notchpeak = dsp.NotchPeakFilter;
[Bnotch,Anotch,Bpeak,Apeak] = tf(notchpeak);
```

`Bnotch` and `Anotch` are the vector of numerator and denominator coefficients for the equivalent transfer function corresponding to the notch filter. `Bpeak` and `Apeak` are the vector of numerator and denominator coefficients for the equivalent transfer function corresponding to the peak filter.

dsp.ParametricEQFilter System object

Package: dsp

Tunable second-order parametric equalizer filter

Note: The `dsp.ParametricEQFilter` object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `multibandParametricEQ` object from Audio System Toolbox instead.

Description

The `ParametricEQFilter` object is a tunable, second-order parametric equalizer filter.

To apply the filter to each channel of the input:

1. Define and set up your equalizer filter. See "Construction" on page 4-1395.
2. Call step to filter each channel according to the properties of `dsp.ParametricEQFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = dsp.ParametricEQFilter` returns a second-order parametric equalizer filter that independently filters each channel of the input over time, using the default values for `Bandwidth`, `CenterFrequency`, and `PeakGaindB`. The `center frequency` and `bandwidth` are specified in Hz and are tunable. The `peak gain` (dip) is specified in dB and is also tunable. The `bandwidth` is measured at the arithmetic mean between the `peak gain` in absolute power units and one.

`H = dsp.ParametricEQFilter('Specification', 'Quality factor and center frequency')` specifies the quality factor (Q factor) of the filter. The `Q factor` is defined as the center frequency/bandwidth. A higher `Q factor` corresponds to a narrower peak/dip. The Q factor should be a scalar value greater than 0. The Q factor is tunable.

`H = dsp.ParametricEQFilter('Specification', 'Coefficients')` specifies the gain values for the bandwidth and center frequency. This removes the trigonometry calculations involved when the properties are tuned. The `CenterFrequencyCoefficient` should be a scalar between -1 and 1, with -1 corresponding to 0 Hz, and 1 corresponding to the Nyquist frequency. The `BandwidthCoefficient` should be a scalar between -1 and 1, with -1 corresponding to the largest bandwidth, and 1 corresponding to the smallest bandwidth. In this mode, the peak gain is specified in linear units rather than dB.

`H = dsp.ParametricEQFilter('Name', Value, ...)` returns a parametric equalizer filter with each specified property name set to the specified value. You can specify several name-value pair arguments in any order as `('Name1',Value1,...,'NameN',ValueN)`.

Properties

`Specification`

Design parameters or coefficients that specify the filter

Choose one of the following `Specification` values. Use the corresponding tunable properties to specify the filter:

- `Bandwidth and center frequency` — Use `Bandwidth`, `CenterFrequency`, and `PeakGaindB`.
- `Quality factor and center frequency` — Use `QualityFactor`, `CenterFrequency`, and `PeakGaindB`.
- `Coefficients` — Use `BandwidthCoefficient`, `CenterFrequencyCoefficient`, and `PeakGain`.

The default value is `Bandwidth and center frequency`.

Using `Coefficients` specifies gain values for the bandwidth and center frequency. This approach does not require the trigonometric calculations of the other two approaches where design parameters are specified in Hz.

Bandwidth

bandwidth of filter

Specify the bandwidth of the filter as a finite positive numeric scalar that is less than half the sample rate of the input signal, in Hz. This property is applicable if `Specification` is set to `Bandwidth and center frequency`. The default is 2205 Hz. This property is tunable.

BandwidthCoefficient

Coefficient for bandwidth of filter

Specify the value that determines the filter's bandwidth as a finite numeric scalar in the range `[-1 1]`:

- `-1` corresponds to the maximum bandwidth (`SampleRate`/4).
- `1` corresponds to the minimum bandwidth (0 Hz, that is, an allpass filter).

This property is only applicable if `Specification` is set to `Coefficients`. The default is 0.72654. This property is tunable.

CenterFrequency

Center frequency of the filter

Specify the filter's center frequency as a finite positive numeric scalar that is less than half the sample rate of the input signal, in Hz. This property is only applicable if `Specification` is set to `Bandwidth and center frequency` or `Quality factor and center frequency`. The default is `11025` Hz. This property is tunable.

CenterFrequencyCoefficient

Coefficient for center frequency of filter

Specify the value that determines the filter's center frequency as a finite numeric scalar between -1 and 1:

- -1 corresponds to the minimum center frequency (0 Hz).
- 1 corresponds to the maximum center frequency (`SampleRate`/2 Hz).

This property is only applicable if `Specification` is set to `Coefficients`. The default is `0`, which corresponds to `SampleRate`/4 Hz.

This property is tunable.

`PeakGain`

Peak or dip gain of the filter in linear units

Specify the filter's peak or dip gain in linear units. A value greater than one boosts the signal. A value less than one attenuates the signal. The default is 2 (6.0206 dB). This property is tunable.

`PeakGaindB`

Peak or dip gain of the filter in dB

Specify the filter's peak or dip gain in dB. A positive value boosts the signal. A negative value attenuates the signal. The default is `6.0206` dB. This property is tunable.

`QualityFactor`

Quality factor of the parametric EQ filter

Specify the Quality factor (Q factor) of the filter. The Q factor is defined as the center frequency divided by the bandwidth. A higher Q factor corresponds to a narrower peak or dip. This property is only applicable if `Specification` is set to `Quality factor and center frequency`. The default value is 5. This property is tunable.

`SampleRate`

Input sample rate

Specify the sample rate of the input as a finite numeric scalar, in Hz. The default is `44100` Hz.

Methods

getBandwidth	Convert quality factor or bandwidth coefficient to bandwidth in Hz
getCenterFrequency	Convert center frequency coefficient to frequency in Hz

getOctaveBandwidth	Measure bandwidth of parametric equalizer filter in octaves
getPeakGain	Convert peak or notch gain from dB to absolute units
getPeakGaindB	Convert peak or notch gain from absolute units to dB
getQualityFactor	Convert bandwidth to quality factor
reset	Reset states of `ParametricEQFilter` object
step	Filter input with `ParametricEQFilter` object
tf	Compute transfer function

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Tune Equalizer Filter

Create a `ParametricEQFilter` object where the center frequency and bandwidth of the equalizer filter are 5000 Hz and 500 Hz respectively. The sample rate for the filter is the default, 44,100 Hz.

```
h = dsp.ParametricEQFilter('CenterFrequency',5000,...
       'Bandwidth',500);
```

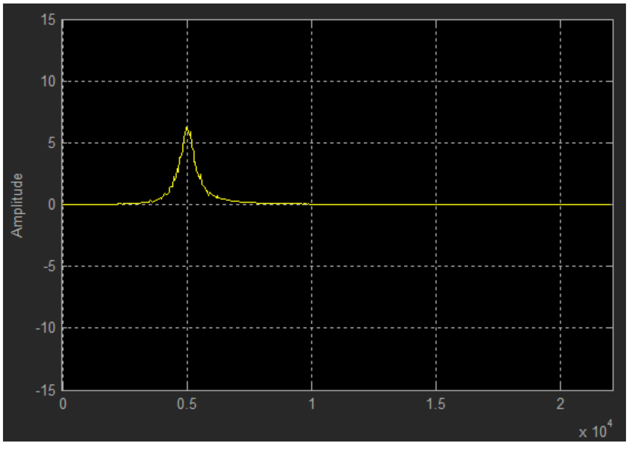
Create objects to estimate and display the transfer function of the filter.

```
htf = dsp.TransferFunctionEstimator('FrequencyRange','onesided',...
         'SpectralAverages',50);
hplot = dsp.ArrayPlot('PlotType','Line','YLimits',[-15 15],...
       'SampleIncrement',44100/1024);
```

Generate a random signal and filter the signal.

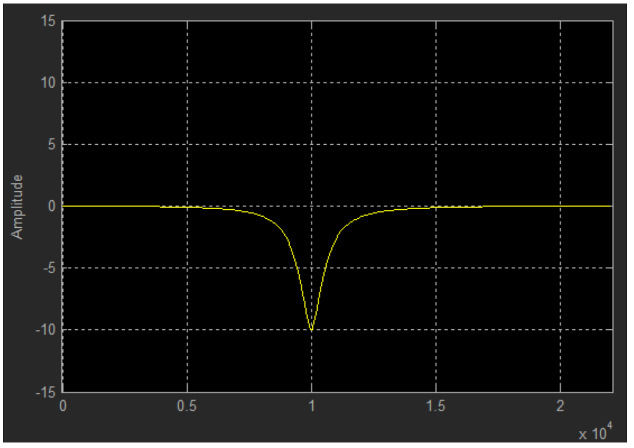
```
for i=1:1000
   x = randn(1024,1);       % Random signal
   y = h(x);           % Filter signal
   H = htf(x,y);       % Estimate transfer function
   magdB = 20*log10(abs(H));% Convert to dB
   hplot(magdB);       % Display transfer function

   if (i==1)                % Pause to display initial transfer function
      pause;
   end
   if (i==500)              % Tune filter
      h.CenterFrequency = 10000;
      h.Bandwidth = 2000;
      h.PeakGaindB = -10;
   end
 end
```

The software displays the initial transfer function estimate.



To continue, press any key.

At `i=500`, the filter is tuned. The center frequency, bandwidth, and peak gain of the filter now have different values. The software displays the new transfer function.

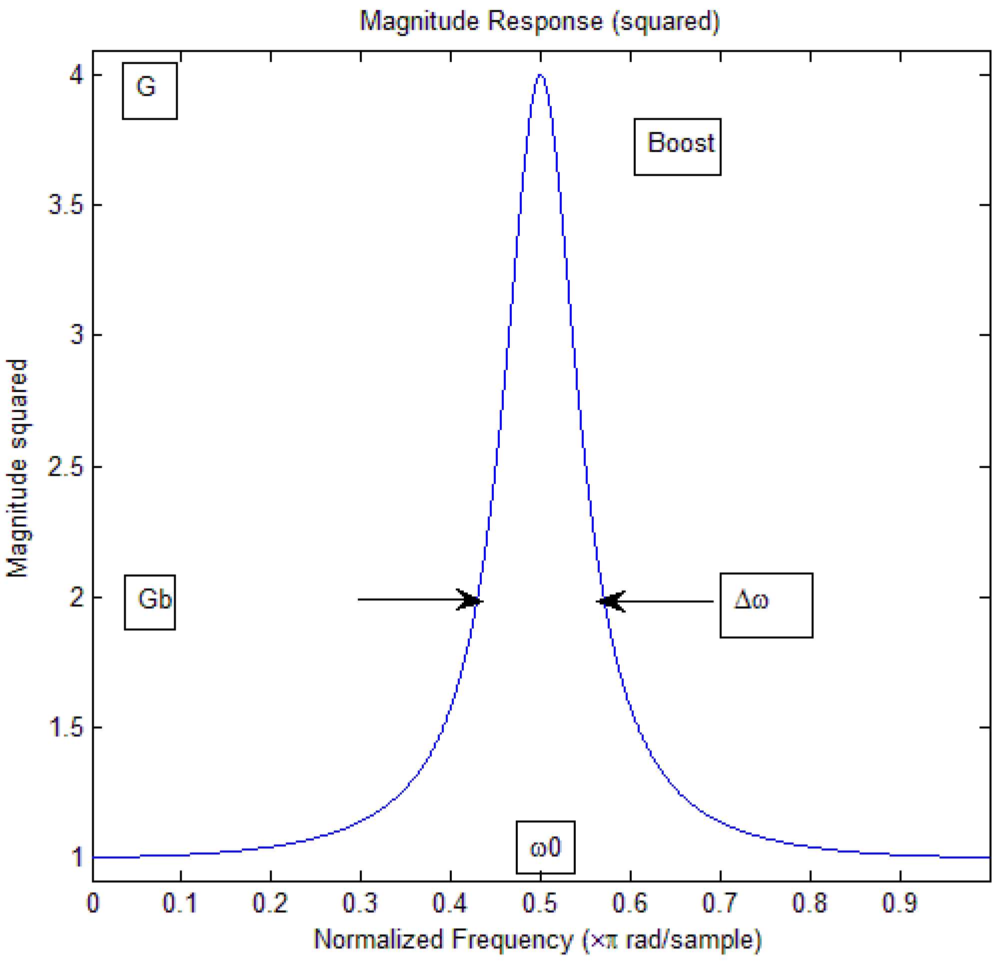


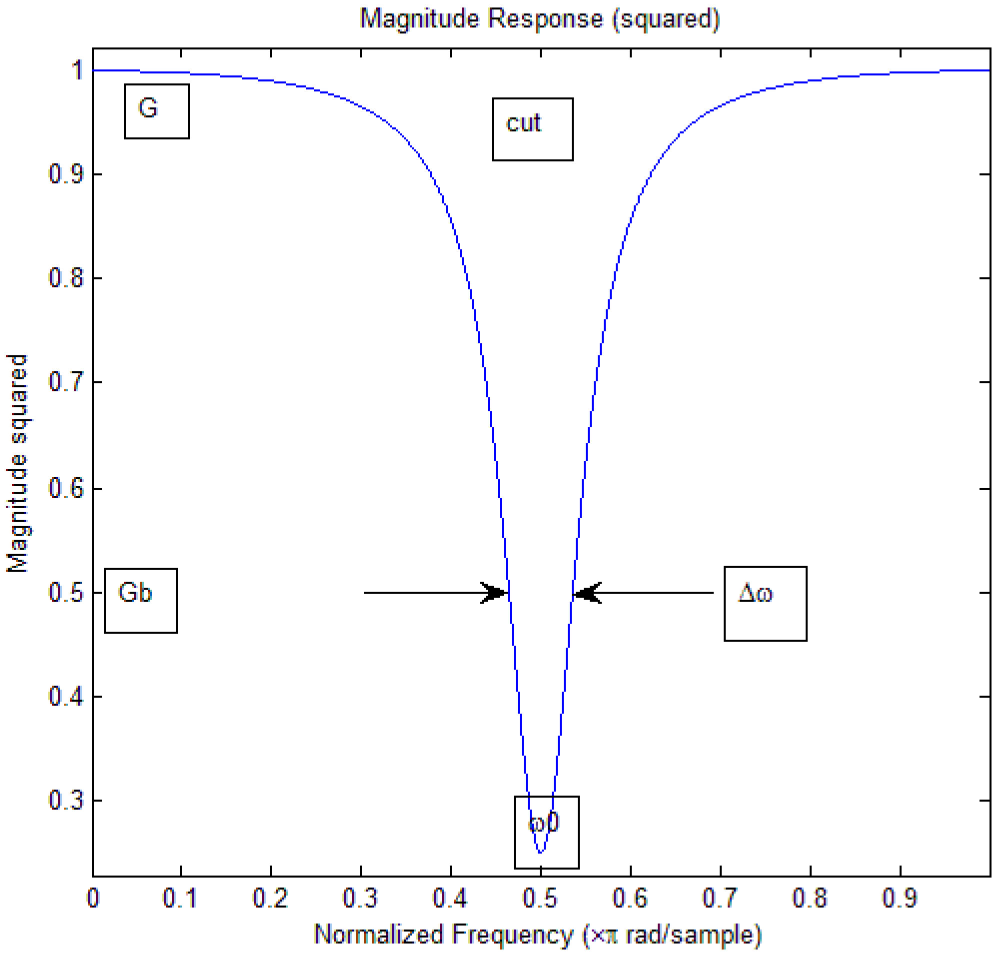
Algorithm

The parametric equalizer is formed by a linear combination of a peak and a notch filter. See the **Algorithm** section of dsp.NotchPeakFilter for details.

$$H(z) = H_{notch}(z) + GH_{peak}(z)$$

Here is a graph of the two cases (boost and cut) of the magnitude squared of the transfer functions:


Magnitude Response (squared)
G
Boost
Gb
Δω
ω0
Magnitude squared
Normalized Frequency (×π rad/sample)
4
3.5
3
2.5
2
1.5
1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9



The transfer function can be written as:

$$H(z) = \frac{(\frac{1+G_\gamma}{1+\gamma}) - 2(\frac{\cos\omega_0}{1+\gamma})z^{-1} + (\frac{1-G_\gamma}{1+\gamma})z^{-2}}{1 - 2(\frac{\cos\omega_0}{1+\gamma})z^{-1} + (\frac{1-\gamma}{1+\gamma})z^{-2}}$$

where

$$\gamma = \tan(\frac{\Delta\omega}{2})$$

and

$$G_B{}^2 = \frac{1+G^2}{2}$$

G is the parametric equalizer gain, and G_B is the bandwidth gain, that is, the gain level at which the bandwidth Δω is measured.

The dsp.NotchPeakFilter that does most of the work is implemented in a decoupled way so that the center frequency can be tuned independently from the bandwidth. Note that the Q factor is defined as center frequency/bandwidth.

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing* Upper Saddle River, NJ: Prentice-Hall, 1996

Related Examples

Parametric Equalizer Design

See Also

`Parametric EQ Filter`	DSP System Toolbox
`iirparameq`	DSP System Toolbox

Introduced in R2014a

getBandwidth

System object: dsp.ParametricEQFilter
Package: dsp

Convert quality factor or bandwidth coefficient to bandwidth in Hz

---

Note: The `dsp.ParametricEQFilter` object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `multibandParametricEQ` object from Audio System Toolbox instead.

Syntax

```
BW = getBandwidth(H)
```

Description

`BW = getBandwidth(H)` returns the bandwidth of the parametric equalizer filter. The bandwidth is measured halfway between 1 and the peak (or notch) gain of the filter. The gain is specified in absolute power units (filter magnitude squared). If the `Specification` property is set to `Quality factor and center frequency`, the bandwidth is determined from the quality factor of the filter. If the `Specification` property is set to `Coefficients`, the bandwidth is determined from the `BandwidthCoefficient` value and the sample rate of the filter.

getCenterFrequency

System object: dsp.ParametricEQFilter
Package: dsp

Convert center frequency coefficient to frequency in Hz

Note: The `dsp.ParametricEQFilter` object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `multibandParametricEQ` object from Audio System Toolbox instead.

Syntax

```
CF = getCenterFrequency(H)
```

Description

`CF = getCenterFrequency(H)` returns the center frequency of the parametric equalizer filter. If the `Specification` property is set to `Coefficients`, the center frequency is determined from the `CenterFrequencyCoefficient` value and the sample rate of the filter.

getOctaveBandwidth

System object: dsp.ParametricEQFilter
Package: dsp

Measure bandwidth of parametric equalizer filter in octaves

Note: The `dsp.ParametricEQFilter` object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `multibandParametricEQ` object from Audio System Toolbox instead.

Syntax

```
N = getOctaveBandwidth(H)
```

Description

`N = getOctaveBandwidth(H)` returns the bandwidth of the parametric equalizer filter in octaves instead of Hz.

getPeakGain

System object: dsp.ParametricEQFilter
Package: dsp

Convert peak or notch gain from dB to absolute units

Note: The `dsp.ParametricEQFilter` object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `multibandParametricEQ` object from Audio System Toolbox instead.

Syntax

```
G = getPeakGain(H)
```

Description

`G = getPeakGain(H)` returns the peak or notch gain of the parametric equalizer filter in absolute units.

getPeakGaindB

System object: dsp.ParametricEQFilter
Package: dsp

Convert peak or notch gain from absolute units to dB

Note: The `dsp.ParametricEQFilter` object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `multibandParametricEQ` object from Audio System Toolbox instead.

Syntax

```
G = getPeakGaindB(H)
```

Description

`G = getPeakGaindB(H)` returns the peak or notch gain of the parametric equalizer filter in dB.

getQualityFactor

System object: dsp.ParametricEQFilter
Package: dsp

Convert bandwidth to quality factor

Note: The `dsp.ParametricEQFilter` object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `multibandParametricEQ` object from Audio System Toolbox instead.

Syntax

```
Q = getQualityFactor(H)
```

Description

`Q = getQualityFactor(H)` returns the quality factor (Q factor) for the parametric equalizer filter. The Q factor is defined as the center frequency divided by the bandwidth.

reset

System object: dsp.ParametricEQFilter
Package: dsp

Reset states of `ParametricEQFilter` object

Note: The `dsp.ParametricEQFilter` object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `multibandParametricEQ` object from Audio System Toolbox instead.

Syntax

```
reset(H)
```

Description

`reset(H)` resets the filter states of the `ParametricEQFilter` object,`H`, to the specified initial conditions. After the `step` method applies the `ParametricEQFilter` object to nonzero input data, the states can change. Invoking the `step` method again without first invoking the `reset` method can produce different outputs for an identical input.

step

System object: dsp.ParametricEQFilter
Package: dsp

Filter input with `ParametricEQFilter` object

Note: The `dsp.ParametricEQFilter` object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `multibandParametricEQ` object from Audio System Toolbox instead.

Syntax

```
Y = step(H, X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H, X)` filters the real or complex input signal `X` using the specified filter to produce the equalized filter output `Y`. The filter processes each channel of the input signal (each column of `X`) independently over time.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change

nontunable properties or inputs, you must first call the `release` method to unlock the object.

tf

System object: dsp.ParametricEQFilter
Package: dsp

Compute transfer function

Note: The `dsp.ParametricEQFilter` object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `multibandParametricEQ` object from Audio System Toolbox instead.

Syntax

`[B,A] = tf(H)`

Description

`[B,A] = tf(H)` returns the vector of numerator coefficients `B` and the vector of denominator coefficients `A` for the equivalent transfer function of the parametric equalizer filter.

dsp.PeakFinder System object

Package: dsp

Determine extrema (maxima or minima) in input signal

Description

The `PeakFinder` object determines the extrema (maxima or minima) in the input signal.

To compute the extrema in the input signal:

1. Define and set up your peak finder. See "Construction" on page 4-1415.
2. Call step to compute the extrema according to the properties of `dsp.PeakFinder`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`pkFind = dsp.PeakFinder` returns a peak finder System object, `pkFind`, that compares the current signal value to the previous and next values to determine if the current value is an extremum.

`pkFind = dsp.PeakFinder('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a peak finder System object, `pkFind`, with each specified property set to the specified value.

Properties

`PeakType`

Looking for maxima, minima, or both

Specify whether the object is looking for maxima, minima, or both. You can set this property to`Maxima`, `Minima`, or `Maxima and Minima`. The default is `Maxima and Minima`.

PeakIndicesOutputPort

Enable output of extrema indices

Set this property to `true` to output the extrema indices. The default is `false`.

PeakValuesOutputPort

Enable output of extrema values

Set this property to `true` to output the extrema values. The default is `false`.

MaximumPeakCount

Number of extrema to look for in each input signal

The object stops searching the input signal for extrema after the maximum number of extrema has been found. The value of this property must be an integer greater than or equal to one. The default is `10`.

IgnoreSmallPeaks

Enable ignoring peaks below threshold

Set this property to `true` if you want to eliminate the detection of peaks whose amplitudes are within a specified threshold of neighboring values. The default is `false`.

PeakThreshold

Threshold below which peaks are ignored

Specify the noise threshold value. This property defines the current input value as a maximum if:

- The current input value minus the previous input value is greater than the threshold, and
- The current input value minus next input value is greater than the threshold.

This property applies when you set the `IgnoreSmallPeaks on page 4-1416` property to `true`. The default is `0`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

The rounding method is a constant property set to `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

Methods

step	Number of extrema in input signal

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Determine the local Maxima and Minima

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Determine whether each value of an input signal is local maximum or minimum.

```
pkFind = dsp.PeakFinder;
pkFind.PeakIndicesOutputPort = true;
```

```
pkFind.PeakValuesOutputPort = true;
```

```
x1 = [9 6 10 3 4 5 0 12]';
```

Find the peaks of each input [prev;cur;next]: {[9;6;10],[6;10;3],...}

```
[cnt1, idx1, val1, pol1] = pkFind(x1)
```

```
cnt1 =

  uint32

   5


idx1 =

  10×1 uint32 column vector

   1
   2
   3
   5
   6
   0
   0
   0
   0
   0


val1 =

     6
    10
     3
     5
     0
     0
     0
     0
     0
     0
```

```
pol1 =

  10×1 logical array

   0
   1
   0
   1
   0
   0
   0
   0
   0
   0
```

Peak Value For a Fixed-Point Signal

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Determine peak values for a fixed-point input signal.

```
pkFind = dsp.PeakFinder('PeakType', 'Maxima', ...
    'PeakValuesOutputPort', true, ...
    'MaximumPeakCount', 2, ...
    'IgnoreSmallPeaks', true, ...
    'PeakThreshold', 0.25, ...
    'OverflowAction', 'Saturate');
x2 = fi([-1;0.5;0],true,16,15);
[cnt2, val2] = pkFind(x2)
```

```
cnt2 =

  uint32

   1


val2 =

    0.5000
```

```
          0

      DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
    FractionLength: 15
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Peak Finder` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.Minimum | dsp.Maximum

Introduced in R2012a

step

System object: dsp.PeakFinder
Package: dsp

Number of extrema in input signal

Syntax

```
YCNT = step(pkFind,X)
[YCNT,IDX] = step(pkFind,X)
[..., VAL] = step(pkFind,X)
[..., POL] = step(pkFind,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`YCNT = step(pkFind,X)` returns the number of extrema (minima, maxima, or both) `YCNT` in input signal `X`.

`[YCNT,IDX] = step(pkFind,X)` when the `PeakIndicesOutputPort` property is `true`, returns the number of extrema `YCNT` and peak indices `IDX` in input signal `X`.

`[..., VAL] = step(pkFind,X)` also returns the peak values `VAL` in input signal `X` when the `PeakValuesOutputPort` property is `true`.

`[..., POL] = step(pkFind,X)` also returns the extrema polarity `POL` in input signal `X` when the `PeakType` property is `Maxima and Minima` and the `PeakIndicesOutputPort` property is `true`. The polarity is `1` for maxima and `0` for minima.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.PeakToPeak System object

Package: dsp

Peak-to-peak value

Description

The `PeakToPeak` object computes the peak-to-peak value of an input.

To obtain the peak-to-peak value:

1. Define and set up your peak-to-peak calculation. See "Construction" on page 4-1423.
2. Call step to compute the peak-to-peak value for an input vector according to the properties of `dsp.PeakToPeak`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ptp = dsp.PeakToPeak` creates a peak-to-peak System object, `ptp`, that computes the difference between the maximum and minimum value in an input or a sequence of inputs.

`ptp = dsp.PeakToPeak('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an `PeakToPeak` System object, `ptp`, with each specified property set to the specified value.

Properties

CustomDimension

Dimension to operate along. Specify the dimension as a positive integer along which the peak-to-peak difference is computed. The value of this property cannot exceed the number of dimensions in the input signal. This property applies when the `Dimension` property is `'Custom'`.

Default: `1`

Dimension

Dimension to operate along. Specify the dimension along which to calculate the peak-to-peak ratio as one of `'All'`, `'Row'`, `'Column'`, or `'Custom'`. This property applies when the `RunningPeakToPeak` property is `false`. If you set this property to `'Custom'`, specify the dimension using the `CustomDimension` property.

Default: `'Column'`

ResetCondition

Reset condition for running peak-to-peak mode. Specify the event to reset the running peak-to-peak as one of `'Rising edge'`, `'Falling edge'`, `'Either edge'`, or `'Non-zero'`. This property applies when the `ResetInputPort` property is `true`.

ResetInputPort

Enables resetting in running peak-to-peak mode. Set this property to `true` to enable resetting the running peak-to-peak. When the property is set to `true`, a reset input must be specified to the call of step to reset the running peak-to-peak difference. This property applies when the `RunningPeakToPeak` property is `true`.

Default: `false`

RunningPeakToPeak

Calculation over successive calls to step. Set this property to `true` to enable the calculation of the peak-to-peak difference over successive calls to the `step`.

Default: `false`

Methods

reset	Reset the running peak-to-peak object
step	Compute peak-to-peak value

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Peak-to-Peak Value of Vector

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Determine the peak-to-peak value for a vector input.

```
in = (1:10)';
ptp = dsp.PeakToPeak;
y = ptp(in)
```

```
y =

     9
```

Peak-to-Peak Value of Matrix Input

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Determine the peak-to-peak value of a matrix input.

```
in = magic(4);
ptp = dsp.PeakToPeak;
ptp.Dimension = 'All';
y = ptp(in)
```

```
y =

    15
```

References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

dsp.Maximum | dsp.PeakToRMS

Introduced in R2012a

reset

System object: dsp.PeakToPeak
Package: dsp

Reset the running peak-to-peak object

Syntax

`reset(ptp)`

Description

`reset(ptp)` resets the running peak-to-peak value for the object `ptp`.

step

System object: dsp.PeakToPeak
Package: dsp

Compute peak-to-peak value

Syntax

```
Y = step(ptp,X)
Y = step(ptp,X,R)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(ptp,X)` computes the peak-to-peak value, `Y`, of the floating-point input vector `X`. When the `RunningPeakToPeak` property is `true`, `Y` corresponds to the peak-to-peak value of the input elements over successive calls to the `step` method.

`Y = step(ptp,X,R)` computes the peak-to-peak value of the input elements over successive calls to the `step` method. The object optionally resets its state based on the reset input signal, `R`, and the value of the `ResetCondition` property. This is possible when you set both the `RunningPeakToPeak` and the `ResetInputPort` properties to `true`.

dsp.PeakToRMS System object

Package: dsp

Peak-to-root-mean-square value of vector

Description

The `PeakToRMS` object calculates the peak-to-root-mean-square ratio of a vector.

To compute the peak-to-root-mean-square ratio:

1 Define and set up your peak-to-root-mean-square calculation. See "Construction" on page 4-1429.
2 Call step to calculate the peak-to-root-mean-square value for an input vector according to the properties of `dsp.PeakToRMS`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ptr = dsp.PeakToRMS` creates a peak-to-root-mean-square System object, `ptr`, that returns the ratio of the maximum magnitude (peak) to the root-mean-square (RMS) value in an input or a sequence of inputs.

`ptr = dsp.PeakToRMS('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an `PeakToRMS` System object, `ptr`, with each specified property set to the specified value.

Properties

`CustomDimension`

Numerical dimension to operate along. Specify the dimension as a positive integer along which the peak-to-RMS ratio is computed. The value of this property cannot exceed the number of dimensions in the input signal. This property applies when `Dimension` property is `'Custom'`.

Default: `1`

`DecibelScaledOutput`

Report output in decibels (dB). Set this property to `true` to enable output in dB. Set this property to `false` to report output as a ratio.

Default: `true`

`Dimension`

Dimension to operate along. Specify the dimension along which to calculate the peak-to-RMS ratio as one of `'All'`, `'Row'`, `'Column'`, or `'Custom'`. This property applies when the `RunningPeakToRMS` property is `false`. If you set this property to `'Custom'`, specify the dimension using the `CustomDimension` property.

Default: `'Column'`

`ResetCondition`

Reset condition for running peak-to-RMS mode. Specify the event to reset the running peak-to-RMS as one of `'Rising edge'`, `'Falling edge'`, `'Either edge'`, or `'Non-zero'`. This property applies when the `ResetInputPort` property is `true`.

`ResetInputPort`

Enables resetting in running peak-to-RMS mode. Set this property to `true` to enable resetting. When the property is set to `true`, a reset input must be specified in the call to step to reset the running peak-to-RMS ratio. This property applies when the `RunningPeakToRMS` property is `true`.

Default: `false`

RunningPeakToRMS

Calculation over successive calls to step. Set this property to `true` to enable the calculation of the peak-to-RMS ratio over successive calls to the `step`.

Default: `false`

Methods

reset	Reset the running peak-to-root-mean-square object
step	Compute peak-to-RMS ratio

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Peak-to-RMS Ratio of Vector Input

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Determine the peak-to-RMS ratio of a vector input.

```
in = (1:10)';
ptr = dsp.PeakToRMS;
y = ptr(in)
```

```
y =
```

```
    1.6116
```

Peak-to-RMS Ratio of Matrix Input

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Determine the peak-to-RMS ratio of a matrix input.

```
in = magic(4);
ptr = dsp.PeakToRMS;
ptr.Dimension = 'All';
y = ptr(in)
```

```
y =

    1.6547
```

Definitions

Peak-magnitude-to-RMS Level

The peak-magnitude-to-RMS level is

$$\frac{||X||_\infty}{\sqrt{\frac{1}{N}\sum_{n=1}^{N}|X_n|^2}}$$

where the l-infinity norm and RMS values are computed along the specified dimension.

References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

dsp.Maximum | dsp.PeakToPeak

Introduced in R2012a

reset

System object: dsp.PeakToRMS
Package: dsp

Reset the running peak-to-root-mean-square object

Syntax

`reset(ptr)`

Description

`reset(ptr)` resets the running peak-to-root-mean-square value for the object `ptr`.

step

System object: dsp.PeakToRMS
Package: dsp

Compute peak-to-RMS ratio

Syntax

```
Y = step(ptr,X)
Y = step(ptr,X,R)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(ptr,X)` computes the peak-to-RMS ratio, `Y`, of the floating-point input vector `X`. When the `RunningPeakToRMS` property is `true`, `Y` corresponds to the peak-to-RMS ratio of the input elements over successive calls to the `step` method.

`Y = step(ptr,X,R)` computes the peak-to-RMS ratio of the input elements over successive calls to the `step` method. The object optionally resets its state based on the reset input signal, `R`, and the value of the `ResetCondition` property. This is possible when you set both the `RunningPeakToRMS` and the `ResetInputPort` properties to `true`.

dsp.PhaseExtractor System object

Package: dsp

Extract the unwrapped phase of a complex input

Description

The `PhaseExtractor` object extracts the unwrapped phase of a complex input.

To extract the unwrapped phase of a signal input:

1. Define and set up your `dsp.PhaseExtractor` System object. See "Construction" on page 4-1436.
2. Call step with your input signal to extract the signal phase according to the properties of this System object. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`phase = dsp.PhaseExtractor` returns a System object, `phase`, that extracts the unwrapped phase of an input signal.

`phase = dsp.PhaseExtractor('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a System object, `phase`, with each specified property set to the specified value.

Properties

`TreatFramesIndependently`

Specify if you want to reset the unwrapped phase at the end of a frame.

If you set this property to:

- `true`: The object treats each frame of data independently. It resets the initial cumulative unwrapped phase to zero each time a new input frame is received.
- `false`: The object ignores boundaries between frames when returning the unwrapped phase.

The default value is `false`.

Methods

reset	Reset internal states of object
step	Extract phase of signal input

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Plot Unwrapped Phase Of a Sine Wave

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create a `dsp.SineWave` System object™. Specify that the object generates an exponential output with a complex exponent.

```
sine = dsp.SineWave('Frequency',10,...
```

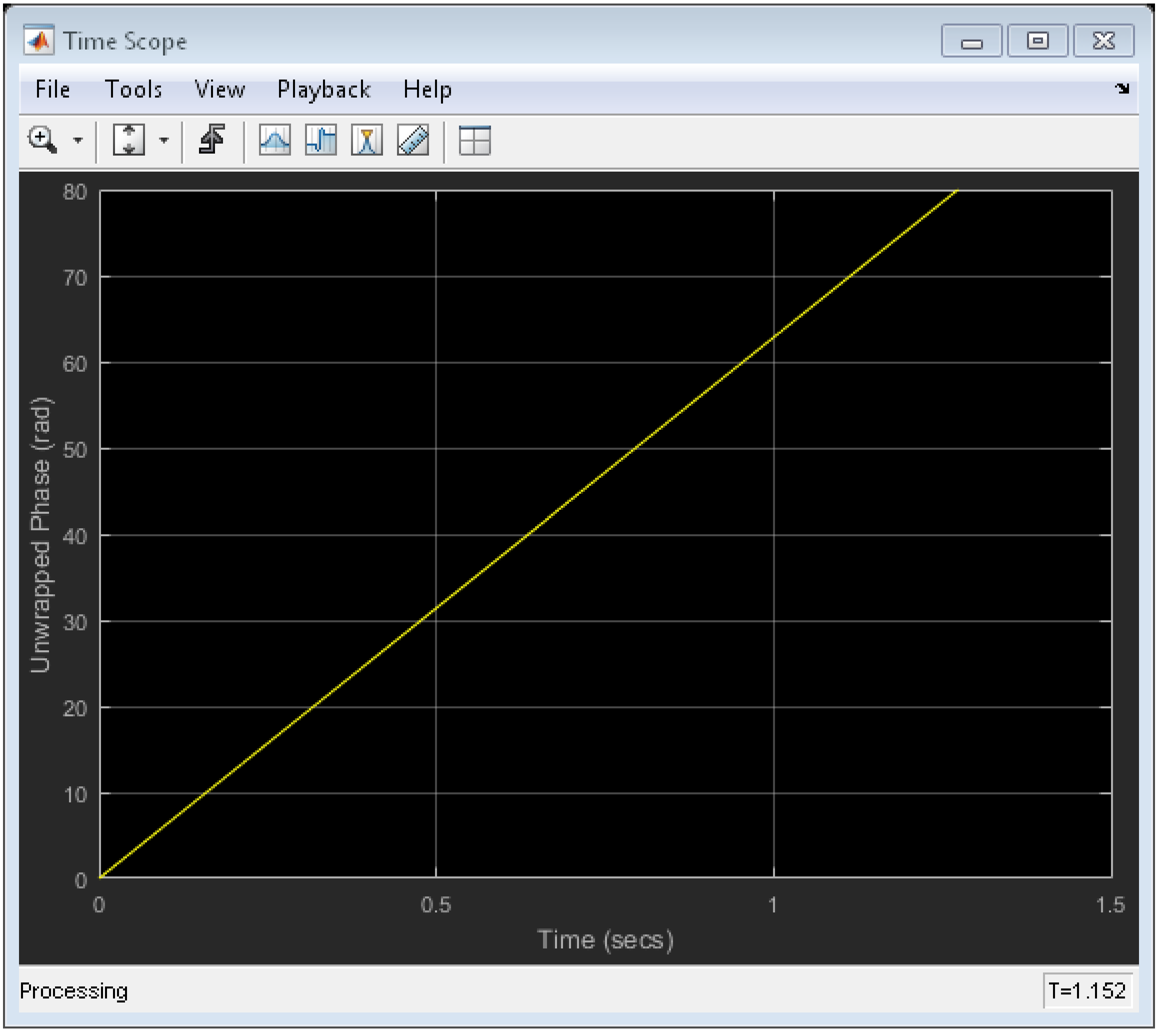
```
    'ComplexOutput',true,'SamplesPerFrame',128);
```

Create a `dsp.PhaseExtractor` System object™. Specify that the object ignores frame boundaries when returning the unwrapped phase.

```
phase = dsp.PhaseExtractor('TreatFramesIndependently',false);
```

Extract the unwrapped phase of a sine wave. Plot the phase versus time using a `dsp.TimeScope` System object™.

```
timeplot = dsp.TimeScope('PlotType','Line','SampleRate',1000,...
    'TimeSpan',1.5,'YLimits',[0 80],...
    'ShowGrid',true,...
    'YLabel','Unwrapped Phase (rad)');
for ii = 1:10
    sineOutput = sine();
    phaseOutput = phase(sineOutput);
    timeplot(phaseOutput)
end
```



Plot Phase Response of Third-Order IIR Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create a `dsp.TransferFunctionEstimator` System object™.

```
tfe = dsp.TransferFunctionEstimator('FrequencyRange','centered');
```

Create a `dsp.PhaseExtractor` System object™. Specify that the object must treat each frame of data independently.

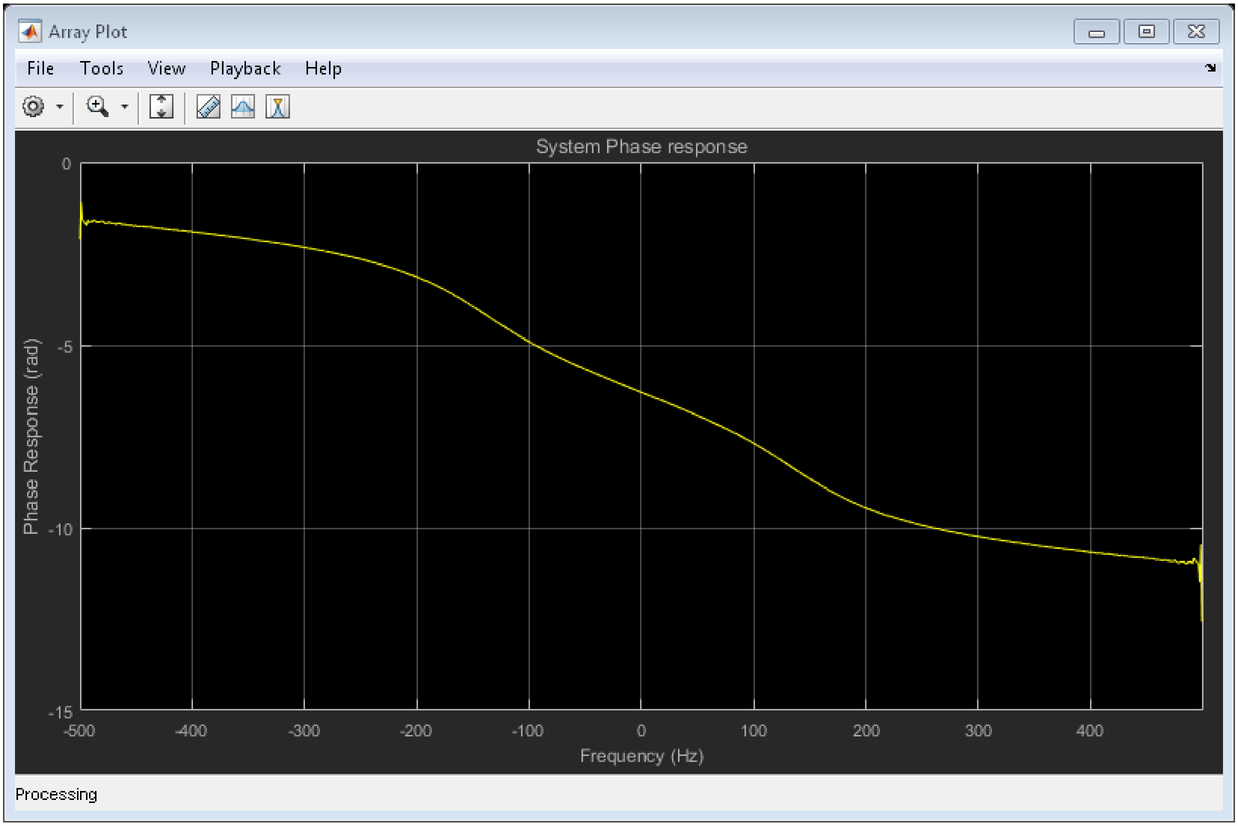
```
phase = dsp.PhaseExtractor('TreatFramesIndependently',true);
```

Create a `dsp.IIRFilter` System object™. Compute the transfer function of a third-order IIR filter. Use the `butter` function to generate coefficients for the filter.

```
[b,a] = butter(3,.3);
iir = dsp.IIRFilter('Numerator',b,'Denominator',a);
```

Extract the phase response of the transfer function. Plot using a `dsp.ArrayPlot` System object™.

```
sampleRate = 1e3;
phaseplot = dsp.ArrayPlot('PlotType','Line','XOffset',-sampleRate/2,...
    'YLimits',[-15 0],...
    'YLabel','Phase Response (rad)',...
    'XLabel','Frequency (Hz)',...
    'Title','System Phase response');
for ii = 1:100
    % Generate input
    input = 0.05*randn(1000,1);
    % Pass through IIR filter
    filterOutput = iir(input);
    % Estimate transfer function
    transferFunction = tfe(input,filterOutput);
    % Plot transfer function phase
    phaseOutput = phase(transferFunction);
    phaseplot(phaseOutput);
end
```



Algorithm

Consider an input frame of length N:

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix}$$

The `step` method acts on this frame and produces this output:

$$\begin{pmatrix} \Phi_1 \\ \Phi_2 \\ \vdots \\ \Phi_N \end{pmatrix}$$

where:

$$\Phi_i = \Phi_{i-1} + \text{angle}(x_{i-1}^* x_i)$$

Here, i runs from 1 to N. The `angle` function returns the phase angle in radians.

If the input signal consists of multiple frames:

- If you set `TreatFramesIndependently` to `true`, the `step` method treats each frame independently. Therefore, in each frame, the `step` method calculates the phase using the preceding formula where:
  - Φ_0 is 0.
  - x_0 is 1.
- If you set `TreatFramesIndependently` to `false`, the `step` method ignores boundaries between frames. Therefore, in each frame, the `step` method calculates the phase using the preceding formula where:
  - Φ_0 is the last unwrapped phase from the previous frame.
  - x_0 is the last sample from the previous frame.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

Blocks

`Phase Extractor`

Introduced in R2014b

reset

System object: dsp.PhaseExtractor
Package: dsp

Reset internal states of object

Syntax

`reset(phase)`

Description

`reset(phase)` resets the values of internal variables Φ_0 to 0 and x_0 to 1. For more information on how the internal variables are used, see the algorithm used by the dsp.PhaseExtractor object.

step

System object: dsp.PhaseExtractor
Package: dsp

Extract phase of signal input

Syntax

`Y = step(phase,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(phase,X)` returns a vector containing the unwrapped phase of the input signal `X` in radians. The method treats each column of `X` as an independent channel. The data types of input signals must be either single or double-precision floating point.

If the property `TreatFramesIndependently` of the `dsp.PhaseExtractor` object `phase` is set to:

- `true`, the object treats each frame of data independently. It resets the initial cumulative unwrapped phase to zero each time a new input frame is received.
- `false`, the object ignores boundaries between frames when returning the unwrapped phase.

.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as

dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.PhaseUnwrapper System object

Package: dsp

Unwrap signal phase

Description

The `PhaseUnwrapper` object unwraps the signal phase input specified in radians.

To unwrap the signal phase input:

1. Define and set up your System object. See "Construction" on page 4-1447.
2. Call step to unwrap the signal phase according to the properties of `dsp.PhaseUnwrapper`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`phUnwrap = dsp.PhaseUnwrapper` returns a System object, `phUnwrap`. This System object adds or subtracts appropriate multiples of 2π to each input element to remove phase discontinuities (unwrap).

`phUnwrap = dsp.PhaseUnwrapper('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an unwrapped System object, `phUnwrap`, with each specified property set to the specified value.

Properties

`InterFrameUnwrap`

Enable unwrapping of phase discontinuities between successive frames

Set this property to `false` to unwrap phase discontinuities only within the frame. Set this property to `true` to also unwrap phase discontinuities between successive frames. The default is `true`.

`Tolerance`

Jump size as true phase discontinuity

Specify the jump size that the phase unwrapper recognizes as a true phase discontinuity. The default is set to π (rather than a smaller value) to avoid altering legitimate signal features. To increase the phase wrapper sensitivity, set the `Tolerance` property to a value slightly less than π.

Methods

reset	Reset internal states of Phase Unwrapper System object
step	Unwrap input phase signal input

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

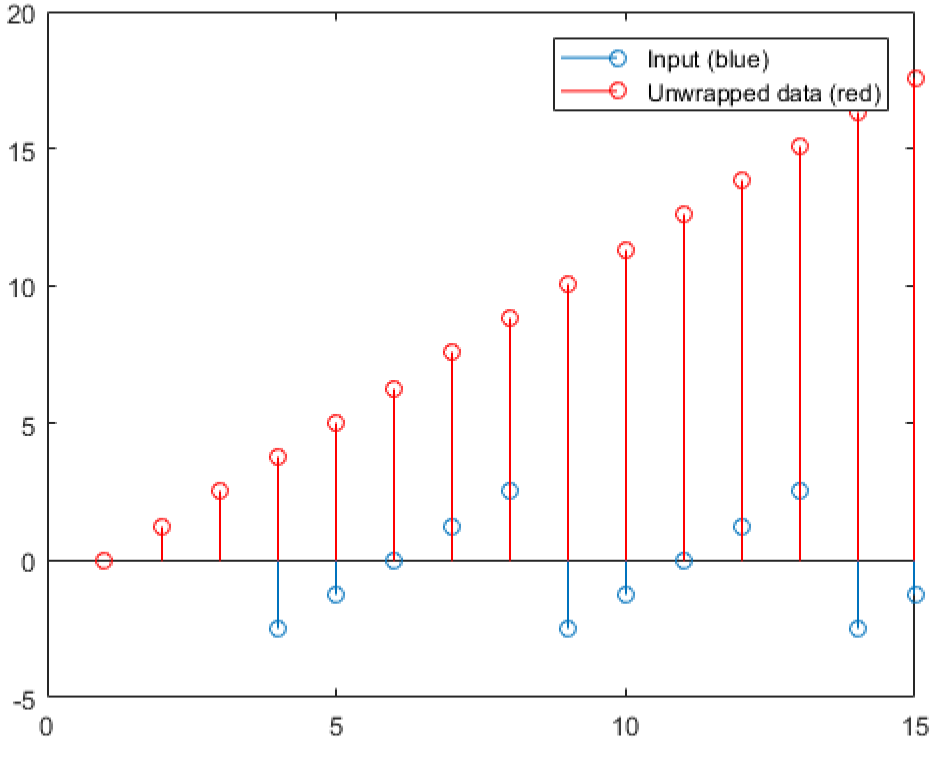
Unwrap input phase data

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
phUnwrap = dsp.PhaseUnwrapper;
p = [0 2/5 4/5 -4/5 -2/5 0 2/5 4/5 -4/5 -2/5 0 2/5 ...
    4/5 -4/5, -2/5]*pi;
```

```
y = phUnwrap(p');
figure,stem(p); hold
stem(y, 'r');
legend('Input (blue)','Unwrapped data (red)');
hold off;
```

```
Current plot held
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Unwrap` block reference page. The object properties correspond to the Simulink block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`unwrap`

Introduced in R2012a

reset

System object: dsp.PhaseUnwrapper
Package: dsp

Reset internal states of Phase Unwrapper System object

Syntax

`reset(phUnwrap)`

Description

`reset(phUnwrap)` resets the 2π multiplier `k` to `0` for the phase unwrapper object `phUnwrap`. See "Frame-Based Processing" on page 2-1807 for details.

step

System object: dsp.PhaseUnwrapper
Package: dsp

Unwrap input phase signal input

Syntax

```
Y = step(phUnwrap,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(phUnwrap,X)` unwraps the input signal `X` in radians for the root mean square (RMS) object `phUnwrap`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.PulseMetrics System object

Package: dsp

Pulse metrics of bilevel waveforms

Description

The `PulseMetrics` object computes rise times, fall times, pulse widths, and cycle metrics including pulse period, pulse separation, and duty cycle for bilevel waveforms.

To obtain pulse metrics for a bilevel waveform:

1. Define and set up your pulse metrics. See "Construction" on page 4-1453.
2. Call step to compute the pulse metrics according to the properties of `dsp.PulseMetrics`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`pm = dsp.PulseMetrics` creates a pulse metrics System object, `pm`. The object computes the rise time, fall time, and width of a pulse. `dsp.PulseMetrics` also computes cycle metrics such as pulse separations, periods, and duty cycles. Because a pulse contains two transitions, the object contains a superset of the capability defined in dsp.TransitionMetrics.

`pm = dsp.PulseMetrics('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a `PulseMetrics` System object, `pm`, with each specified property set to the specified value.

Properties

CycleOutputPort

Enable cycle metrics. If CycleOutputPort is true, cycle metrics are reported for each pulse period.

Default: false

MaximumRecordLength

Maximum samples to preserve between calls to step. This property requires a positive integer that specifies the maximum number of samples to save between calls to the step method. When the number of samples to be saved exceeds this length, the oldest excess samples are discarded. This property applies when the RunningMetrics property is true and is tunable.

Default: 1000

PercentReferenceLevels

Lower-, middle-, and upper-percent reference levels. This property contains a three-element numeric row vector that contains the lower-, middle-, and upper-percent reference levels. These reference levels are used as an offset between the lower and upper states of the waveform when computing the duration of each transition.

Default: [10 50 90]

PercentStateLevelTolerance

Tolerance of the state level (in percent). This property requires a scalar that specifies the maximum deviation from either the low or high state before it is considered to be outside that state. The tolerance is expressed as a percentage of the waveform amplitude.

Default: 2

Polarity

Polarity of pulse to extract. This property specifies the type of pulse to extract by the polarity of the leading transition. Valid values for this property are 'positive' or 'negative'.

Default: `'positive'`

PostshootOutputPort

Enable posttransition aberration metrics. If this property is set to `true`, overshoot and undershoot metrics are reported for a region defined immediately after each transition. The posttransition aberration region is defined as the waveform interval that begins at the end of each transition and whose duration is the value of `PostshootSeekFactor` times the computed transition duration. If a complete subsequent transition is detected before the interval is over, the region is truncated at the start of the subsequent transition. The metrics are computed for each transition that has a complete posttransition aberration region.

Default: `false`

PostshootSeekFactor

Corresponds to the duration of time to search for the overshoot and undershoot metrics immediately following each transition. The duration is expressed as a factor of the duration of the transition. This property is enabled only when the `PostshootOutputPort` property is set to `true` and is tunable.

Default: 3

PreshootOutputPort

Enable pretransition aberration metrics. If `PreshootOutputPort` is set to `true`, overshoot and undershoot metrics are reported for a region defined immediately before each transition. The pretransition aberration region is defined as the waveform interval that ends at the start of each transition and whose duration is `PreshootSeekFactor` times the computed transition duration.

Default: `'false'`

PreshootSeekFactor

Corresponds to the duration of time to search for the overshoot and undershoot metrics immediately preceding each transition. The duration is expressed as a factor of the duration of the transition. This property is enabled only when the `PreshootOutputPort` property is set to `true` and is tunable.

Default: 3

RunningMetrics

Enable metrics over all calls to step. If RunningMetrics is set to false, metrics are computed for each call to step independently. If RunningMetrics is set to true, metrics are computed across subsequent calls to step. If there are not enough samples to compute metrics associated with the last transition, posttransition aberration region, or settling seek duration in the current record, the object will defer reporting all transition, aberration, and settling metrics associated with the last transition until a subsequent call to step is made with enough data to compute all enabled metrics for that transition.

Default: false

SampleRate

Sampling rate of uniformly-sampled signal. Specify the sample rate in hertz as a positive scalar. This property is used to construct the internal time values that correspond to the input sample values. Time values start with zero. This property applies when the TimeInputPort property is set to false.

Default: 1

SettlingOutputPort

Enable settling metrics. If the SettlingOutputPort property is set to true, settling metrics are reported for each transition. The region used to compute the settling metrics starts at the midcrossing and lasts until the SettlingSeekDuration has elapsed. If an intervening transition occurs, or the signal has not settled within the PercentStateLevelTolerance of the final level, NaN is returned for each metric. If there are not enough samples after the last transition to complete the SettlingSeekDuration, no metrics are reported for the last transition. The metrics are reported for the transition the next time step is called if the RunningMetrics property is set to true.

Default: false

SettlingSeekDuration

Duration of time over which to search for settling. This property value is a scalar that specifies the amount of time to inspect from the mid-reference level crossing (in seconds). If the transition has not yet settled, or a subsequent complete transition is detected within this duration, the PulseMetrics object will report NaN for all settling metrics.

This property is tunable and applies only when you set the `SettlingOutputPort` property to `true`.

Default: `0.02`

StateLevels

Low- and high-state levels. This property is a 2–element numeric row vector that contains the low- and high-state levels respectively. These state levels correspond to the nominal logic low and high levels of the pulse waveform. This property is tunable.

Default: `[0 2.3]`

StateLevelsSource

Auto or manual state-level computation. If `StateLevelsSource` is set to `'Auto'`, the first record sent to `step` is sent to dsp.StateLevels with the default settings to determine the state levels of the incoming waveform. If this property is set to `'Property'`, the object uses the values the user specifies in the `StateLevels` property.

Default: `'Property'`

TimeInputPort

Add input to specify sample instants. Set `TimeInputPort` to `true` to enable an additional real input column vector to `step` to specify the sample instants that correspond to the sample values. If this property is `false`, the sample instants are built internally. The sample instants start at zero and increment by the reciprocal of the `SampleRate` property for subsequent samples. The sample instants continue to increment if the `RunningMetrics` property is set to `true` and no intervening calls to the `reset` or `release` methods are encountered.

Default: `false`

TransitionOutputPort

Enable transition metrics. If the `TransitionOutputPort` property is set to `true`, transition metrics are reported for the initial and final transitions of each pulse.

Default: `false`

Methods

plot	Plots signal and metrics computed in last call to `step`
reset	Reset the running pulse metrics object
step	Pulse metrics of bilevel waveforms

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Width, Period, and Duty Cycle

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Determine the width, period, and duty cycle of a 5 V pulse sampled at 4 MHz.

```
load('pulseex.mat', 'x', 't');
```

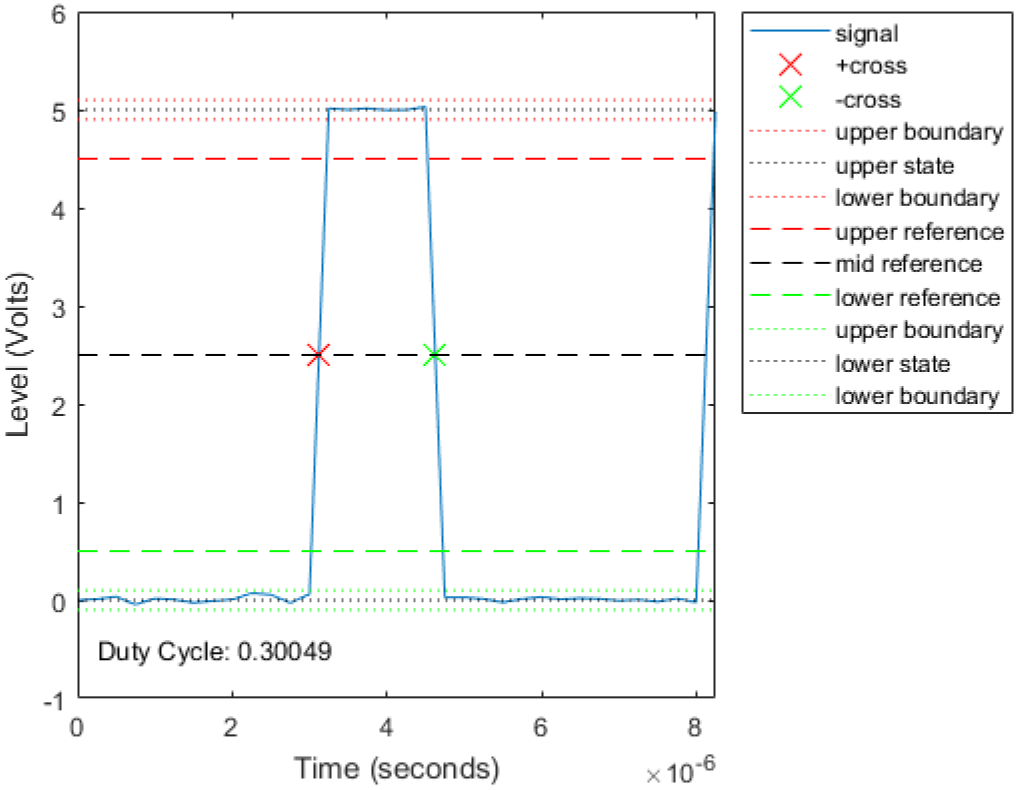
Construct the `dsp.PulseMetrics` object. Set the `TimeInputPort` property to `true` to specify the sampling instants as an input. Set the `CycleOutputPort` property to `true` to obtain metrics for each pulse. Because the input is a 5V pulse, set the `StateLevels` property to `[0 5]`.

```
pm = dsp.PulseMetrics('TimeInputPort',true, ...
                      'CycleOutputPort', true, ...
                      'StateLevels',[0 5])
```

```
pm =

  dsp.PulseMetrics with properties:

                    Polarity: 'Positive'
           StateLevelsSource: 'Property'
                 StateLevels: [0 5]
   PercentStateLevelTolerance: 2
      PercentReferenceLevels: [10 50 90]
              RunningMetrics: false
               TimeInputPort: true
             CycleOutputPort: true
        TransitionOutputPort: false
          PreshootOutputPort: false
         PostshootOutputPort: false
          SettlingOutputPort: false
```

Call `step` to compute the cycle metrics and plot the result.

```
[pulse, cycle] = pm(x,t);
plot(pm)
text(t(2),-0.5,['Duty Cycle: ',num2str(cycle.DutyCycle)]);
```



Slew Rates for 2.3 V Digital Clock

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Find the slew rates of the leading and trailing edges of a 2.3 V digital clock sampled at 4 MHz.

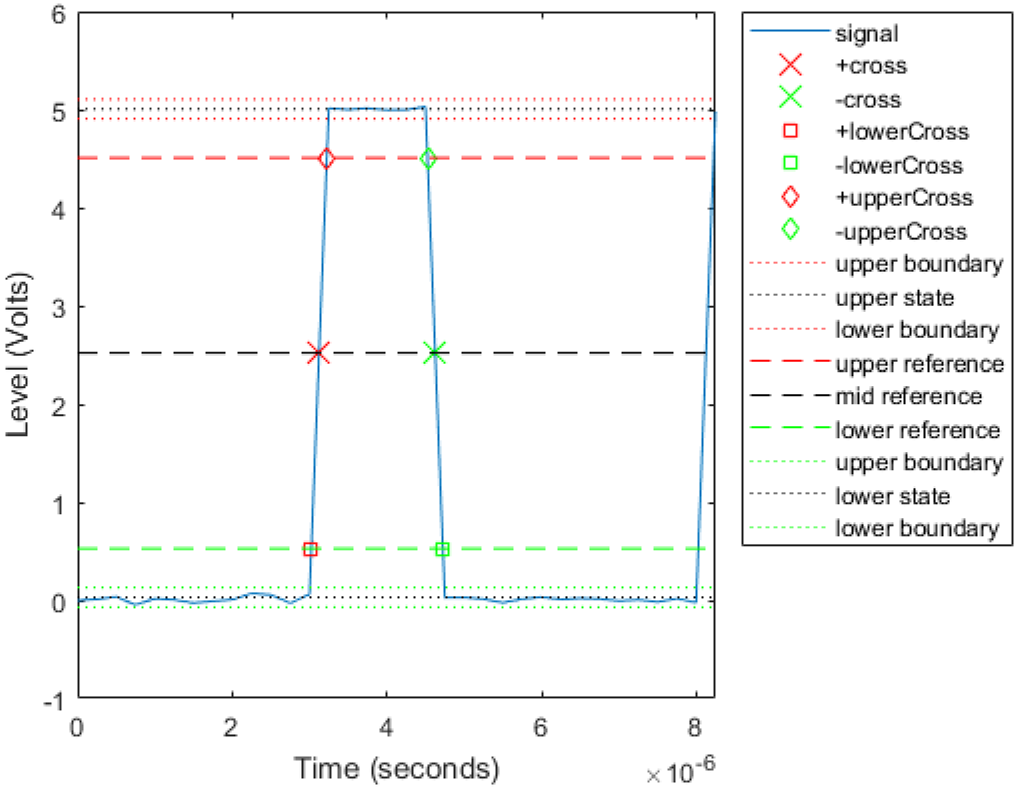
```
load('pulseex.mat', 'x', 't');
```

Construct the `dsp.PulseMetrics` object. Set the `TransitionOutputPort` property to `true` to report transition metrics for the initial and final transitions. Set the `StateLevelsSource` property to `'Auto'` to estimate the state levels from the data.

```
pm = dsp.PulseMetrics('SampleRate',4e6, ...
                      'TransitionOutputPort', true, ...
                      'StateLevelsSource','Auto');
```

Compute the pulse and transition metrics and plot the result.

```
[pulse, transition] = pm(x);
plot(pm);
```



References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

See Also

See Also

dsp.StateLevels | dsp.TransitionMetrics

Introduced in R2012a

plot

System object: dsp.PulseMetrics
Package: dsp

Plots signal and metrics computed in last call to `step`

Syntax

`plot(pm)`

Description

`plot(pm)` plots the signal and metrics resulting from the last call of the step method.

By default `plot` displays

- the low and high state levels and the state-level boundaries defined by the `PercentStateLevelTolerance` property.
- the lower-, middle-, and upper-reference levels.
- the locations of the mid-reference level crossings of the positive (+) and negative (-) transitions of each detected pulse.

When the `TransitionOutputPort` property is set to `true`, the locations of the upper and lower crossings are also plotted. When the `PreshootOutputPort` or `PostShootOutputPort` properties are set to `true`, the corresponding overshoots and undershoots are plotted as inverted or noninverted triangles. When the `SettlingOutputPort` property is set to `true`, the locations where the signal enters and remains within the lower and upper state boundaries over the specified seek duration are plotted.

reset

System object: dsp.PulseMetrics
Package: dsp

Reset the running pulse metrics object

Syntax

`reset(pm)`

Description

`reset(pm)` clears information from previous calls to step.

step

System object: dsp.PulseMetrics
Package: dsp

Pulse metrics of bilevel waveforms

Syntax

```
PULSE = step(pm,X)
[PULSE,CYCLE] = step(pm,X)
[PULSE, TRANSITION] = step(pm,X)
[PULSE,PRESHOOT] = step(pm,X)
[PULSE,POSTSHOOT] = step(pm,X)
[PULSE,SETTLING] = step(pm,X)
[...] = step(pm,X,T)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`PULSE = step(pm,X)` returns a structure array, `PULSE`, whose fields contain real-valued column vectors. The number of rows of each field corresponds to the number of complete pulses found in the real-valued column vector input, `X`. Each pulse starts with a transition of the polarity specified by the `Polarity` property and ends with a transition of the opposite polarity.

`PULSE` fields:

- `PositiveCross` — Instants where the positive-going transitions cross the mid-reference level of each pulse
- `NegativeCross` — Instants where the negative-going transitions cross the mid-reference level of each pulse

- `Width` — Absolute difference between `PositiveCross` and `NegativeCross` of each pulse
- `RiseTime` — Duration between the linearly-interpolated instants when the positive-going (rising) transition of each pulse crosses the lower- and upper-reference levels
- `FallTime` — Duration between the linearly-interpolated instants when the negative-going (falling) transition of each pulse crosses the upper- and lower-reference levels

`[PULSE,CYCLE] = step(pm,X)` returns a structure array, `CYCLE`, whose fields contain real-valued column vectors when you set the `CycleOutputPort` property to `true`. The number of rows of each field corresponds to the number of complete pulse periods found in the real-valued column vector input, `X`. You need at least three consecutive alternating polarity transitions that start and end with the same polarity as the value of the `Polarity` property if you want to compute cycle metrics. If the last transition found in the input `X` does not match the polarity of the `Polarity` property, the pulse separation, period, frequency, and duty cycle are not reported for the last pulse. If the `RunningMetrics` property is set to `true` when this occurs, all pulse, cycle, transition, preshoot, postshoot, and settling metrics associated with the last pulse are deferred until a subsequent call to `step` detects the next transition.

`CYCLE` fields:

- `Period` — Duration between the first transition of the current pulse and the first transition of the next pulse
- `Frequency` — Reciprocal of the period
- `Separation` — Durations between the mid-reference level crossings of the second transition of each pulse and the first transition of the next pulse
- `Width` — Durations between the mid-reference level crossings of the first and second transitions of each pulse. This is equivalent to the width parameter of the `PULSE` structure.
- `DutyCycle` — Ratio of the width to the period for each pulse

`[PULSE, TRANSITION] = step(pm,X)` returns a structure array, `TRANSITION`, when you set the `TransitionOutputPort` property to `true`. The fields of `TRANSITION` contain real-valued matrices with two columns which correspond to the metrics of the first and second transitions. The number of rows corresponds to the number of pulses found in the input waveform.

`TRANSITION` fields:

- `Duration` — Amount of time between the interpolated instants where the transition crosses the lower- and upper-reference levels
- `SlewRate` — Ratio of absolute difference between the upper and lower reference levels to the transition duration
- `MiddleCross` — Linearly-interpolated instant in time where the transition first crosses the mid-reference level
- `LowerCross` — Linearly-interpolated instant where the signal crosses the lower-reference level
- `UpperCross` — Linearly-interpolated instant where the signal crosses the upper-reference level

`[PULSE,PRESHOOT] = step(pm,X)` returns a structure array, `PRESHOOT`, when you set the `PreshootOutputPort` property to `true`. The fields of `PRESHOOT` contain real-valued 2-column matrices whose row length corresponds to the number of transitions found in the input waveform. The field names are identical to those of the `POSTSHOOT` structure array.

`[PULSE,POSTSHOOT] = step(pm,X)` returns a structure, `POSTSHOOT`, when you set the `PostshootOutputPort` property to `true`. The fields of `POSTSHOOT` contain real-valued 2-column matrices whose row length corresponds to the number of transitions found in the input waveform.

`PRESHOOT` and `POSTSHOOT` fields:

- `Overshoot` — Overshoot of the region of interest expressed as a percentage of the waveform amplitude
- `Undershoot` — Undershoot of the region of interest expressed as a percentage of the waveform amplitude
- `OvershootLevel` — Level of the overshoot
- `UndershootLevel` — Level of the undershoot
- `OvershootInstant` — Instant that corresponds to the overshoot
- `UndershootInstant` — Instant that corresponds to the undershoot

`[PULSE,SETTLING] = step(pm,X)` returns a structure array, `SETTLING`, when you set the `SettlingOutputPort` property to `true`. The fields of `SETTLING` correspond to the settling metrics for each transition. Each field is a column vector whose elements correspond to the individual settling durations, levels, and instants.

SETTLING fields:

- Duration — Amount of time from when the signal crosses the mid-reference level to the time where the signal enters and remains within the specified PercentStateLevelTolerance of the waveform amplitude over the specified settling seek duration
- Instant — Instant in time where the signal enters and remains within the specified tolerance
- Level — Level of the waveform where it enters and remains within the specified tolerance

The above operations can be used simultaneously, provided the System object properties are set appropriately. One example of providing all possible inputs and returning all possible outputs is :

[PULSE,CYCLE,TRANSITION,PRESHOOT,POSTSHOOT,SETTLING] = step(pm,X) which returns the PULSE, CYCLE, TRANSITION, PRESHOOT, POSTSHOOT, and SETTLING structures when the CycleOutputPort, PreshootOutputPort, PostshootPort, and SettlingOutputPort properties are true. You may enable or disable any combination of output ports. However, the output arguments are defined in the order shown here.

[...] = step(pm,X,T) performs the above metrics with respect to a sampled signal, whose sample values, X, and sample instants, T, are real-valued column vectors of the same length. The additional input T applies only when you set the TimeInputPort property to true.

dsp.RCToAutocorrelation System object

Package: dsp

Convert reflection coefficients to autocorrelation coefficients

Description

The `RCToAutocorrelation` object converts reflection coefficients to autocorrelation coefficients.

To convert reflection coefficients to autocorrelation coefficients:

1. Define and set up your System object. See "Construction" on page 4-1469.
2. Call step to convert reflection coefficients according to the properties of `dsp.RCToAutocorrelation`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`rc2ac = dsp.RCToAutocorrelation` returns an `RCToAutocorrelation` System object, `rc2ac`. This object converts reflection coefficients to autocorrelation coefficients, assuming an error power of `1`.

`rc2ac = dsp.RCToAutocorrelation('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an object, `rc2ac`, that converts reflection coefficients into autocorrelation coefficients, with each specified property set to the specified value.

Properties

PredictionErrorInputPort

Enable prediction error power input

Choose how to select the prediction error power. When you set this property to `true`, you must specify the prediction error power as a second input to the `step` method. When you set this property to `false`, the object assumes a prediction error power of `1`. The default is `false`.

Methods

step	Convert columns of reflection coefficients to autocorrelation coefficients

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Convert Reflection Coefficients to Autocorrelation Coefficients

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
k = [-0.8091 0.2525 -0.5044 0.4295 -0.2804 0.0711].';
rc2ac = dsp.RCToAutocorrelation;
```

```
ac = rc2ac(k);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `LPC/RC to Autocorrelation` block reference page. The object properties correspond to the block parameters, except:

- There is no object property that corresponds to the **Type of conversion** block parameter.
- **PredictionErrorInputPort** is a pop-up menu choice on the block. Setting the `PredictionErrorInputPort` object property to `false` corresponds to selecting `Assume P = 1` in the pop-up menu. Setting `PredictionErrorInputPort` to `true` corresponds to selecting `Via input port` from the pop-up menu.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.LPCToRC | dsp.LPCToLSF | dsp.LPCToCepstral | dsp.LPCToAutocorrelation

Introduced in R2012a

step

System object: dsp.RCToAutocorrelation
Package: dsp

Convert columns of reflection coefficients to autocorrelation coefficients

Syntax

```
AC = step(rc2ac,K)
AC = step(rc2ac,K,P)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`AC = step(rc2ac,K)` converts the columns of the reflection coefficients, `K`, to autocorrelation coefficients, `AC`.

`AC = step(rc2ac,K,P)` when you set the `PredictionErrorInputPort` property to `true`, converts the columns of the reflection coefficients, `K` , to autocorrelation coefficients, `AC`, using `P` as the prediction error power. `P` must be a row vector with same number of columns as in `K`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.RCToLPC System object

Package: dsp

Convert reflection coefficients to linear prediction coefficients

Description

The `RCToLPC` object converts reflection coefficients to linear prediction coefficients.

To convert reflection coefficients to LPC:

1. Define and set up your RC to LPC System object. See “Construction” on page 4-1473.
2. Call step to convert RC to LPC according to the properties of `dsp.RCToLPC`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`rc2lpc = dsp.RCToLPC` returns an RC to LPC System object, `rc2lpc`, that converts reflection coefficients (RC) to linear prediction coefficients (LPC).

`rc2lpc = dsp.RCToLPC('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an RC to LPC conversion object, `rc2lpc`, with each specified property set to the specified value.

Properties

`PredictionErrorOutputPort`

Enable normalized prediction error power output

Set this property to `true` to return the normalized error power as a vector with one element per input channel. Each element varies between `0` and `1`. The default is `true`.

`ExceptionOutputPort`

Produce output with stability status of filter represented by LPC coefficients

Set this property to `true` to return the stability of the filter. The output is a vector of a length equal to the number of channels. A logical value of `1` indicates a stable filter. A logical value of `0` indicates an unstable filter. The default is `false`.

Methods

step	Convert columns of reflection coefficients to linear prediction coefficients

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Convert RC to LPC Coefficients

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Convert reflection coefficients to linear prediction coefficients.

```
levinson = dsp.LevinsonSolver;
ac = dsp.Autocorrelator;
ac.MaximumLagSource = 'Property';
```

```
ac.MaximumLag = 10; % Compute autocorrelation
% lags between [0:10]
rc2lpc = dsp.RCToLPC;
x = (1:100)';
a = ac(x);
k = levinson(a); % Compute reflection coefficients
[A, P] = rc2lpc(k);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `LPC to/from RC` block reference page. The object properties correspond to the block parameters, except:
There is no object property that corresponds to the **Type of conversion** block parameter. The object always converts LPC to RC.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.LPCToRC | dsp.LPCToAutocorrelation

Introduced in R2012a

step

System object: dsp.RCToLPC
Package: dsp

Convert columns of reflection coefficients to linear prediction coefficients

Syntax

```
[A,P] = step(rc2lpc,K)
A = step(rc2lpc,K)
[..., S] = step(rc2lpc,K)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[A,P] = step(rc2lpc,K)` converts the columns of the reflection coefficients, `K`, to linear prediction coefficients, `A`, and outputs the normalized prediction error power, `P`.

`A = step(rc2lpc,K)` when the `PredictionErrorOutputPort` property is `false`, converts the columns of the reflection coefficients, `K`, to linear prediction coefficients, `A`.

`[..., S] = step(rc2lpc,K)` also outputs the `LPC` filter stability, `S`, when the `ExceptionOutputPort` property is `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change

nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.RLSFilter System object

Package: dsp

Compute output, error and coefficients using Recursive Least Squares (RLS) algorithm

Description

The `RLSFilter` object filters each channel of the input using RLS filter implementations.

To filter each channel of the input:

1. Define and set up your RLS filter. See “Construction” on page 4-1478.
2. Call step to filter each channel of the input according to the properties of `dsp.RLSFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`rlsFilt = dsp.RLSFilter` returns an adaptive RLS filter System object, `rlsFilt`. This System object computes the filtered output, filter error and the filter weights for a given input and desired signal using the RLS algorithm.

`rlsFilt = dsp.RLSFilter('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns an RLS filter System object, `rlsFilt`, with each specified property set to the specified value.

`rlsFilt = dsp.RLSFilter(LEN, '`*`PropertyName`*`', `*`PropertyValue`*`, ...)` returns an RLS filter System object, `rlsFilt`. This System object has the Length property set to `LEN`, and other specified properties set to the specified values.

Properties

`Method`

Method to calculate the filter coefficients

You can specify the method used to calculate filter coefficients as one of | `Conventional RLS` [1] [2] | `Householder RLS` [3] [4] | `Sliding-window RLS` [5][1][2] | `Householder sliding-window RLS` [4] | `QR decomposition` [1] [2]. The default value is `Conventional RLS`. This property is nontunable.

`Length`

Length of filter coefficients vector

Specify the length of the RLS filter coefficients vector as a scalar positive integer value. The default value is `32`. This property is nontunable.

`SlidingWindowBlockLength`

Width of the sliding window

Specify the width of the sliding window as a scalar positive integer value greater than or equal to the Length property value. This property is applicable only when the Method property is set to `Sliding-window RLS` or `Householder sliding-window RLS`. The default value is `48`. This property is nontunable.

`ForgettingFactor`

RLS forgetting factor

Specify the RLS forgetting factor as a scalar positive numeric value less than or equal to 1. Setting this property value to 1 denotes infinite memory, while adapting to find the new filter. The default value is `1`. This property is tunable.

`InitialCoefficients`

Initial coefficients of the filter

Specify the initial values of the FIR adaptive filter coefficients as a scalar or a vector of length equal to the `Length` property value. The default value is `0`. This property is tunable.

InitialInverseCovariance

Initial inverse covariance

Specify the initial values of the inverse covariance matrix of the input signal. This property must be either a scalar or a square matrix, with each dimension equal to the `Length` property value. If you set a scalar value, the `InverseCovariance` property is initialized to a diagonal matrix with diagonal elements equal to that scalar value. This property applies only when the `Method` property is set to `Conventional RLS` or `Sliding-window RLS`. The default value is `1000`. This property is tunable.

InitialSquareRootInverseCovariance

Initial square root inverse covariance

Specify the initial values of the square root inverse covariance matrix of the input signal. This property must be either a scalar or a square matrix with each dimension equal to the `Length` property value. If you set a scalar value, the `SquareRootInverseCovariance` property is initialized to a diagonal matrix with diagonal elements equal to that scalar value. This property applies only when the `Method` property is set to `Householder RLS` or `Householder sliding-window RLS`. The default value is `sqrt(1000)`. This property is tunable.

InitialSquareRootCovariance

Initial square root covariance

Specify the initial values of the square root covariance matrix of the input signal. This property must be either a scalar or a square matrix with each dimension equal to the `Length` property value. If you set a scalar value, the `SquareRootCovariance` property is initialized to a diagonal matrix with diagonal elements equal to the scalar value. This property applies only when the `Method` property is set to `QR-decomposition RLS`. The default value is `sqrt(1/1000)`. This property is tunable.

LockCoefficients

Lock coefficient updates

Specify whether the filter coefficient values should be locked. When you set this property to `true`, the filter coefficients are not updated and their values remain the same. The default value is `false` (filter coefficients continuously updated). This property is tunable.

Methods

msesim	Mean-square error for RLS filter
reset	Reset the internal states of a System object
step	Process inputs using RLS filter

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

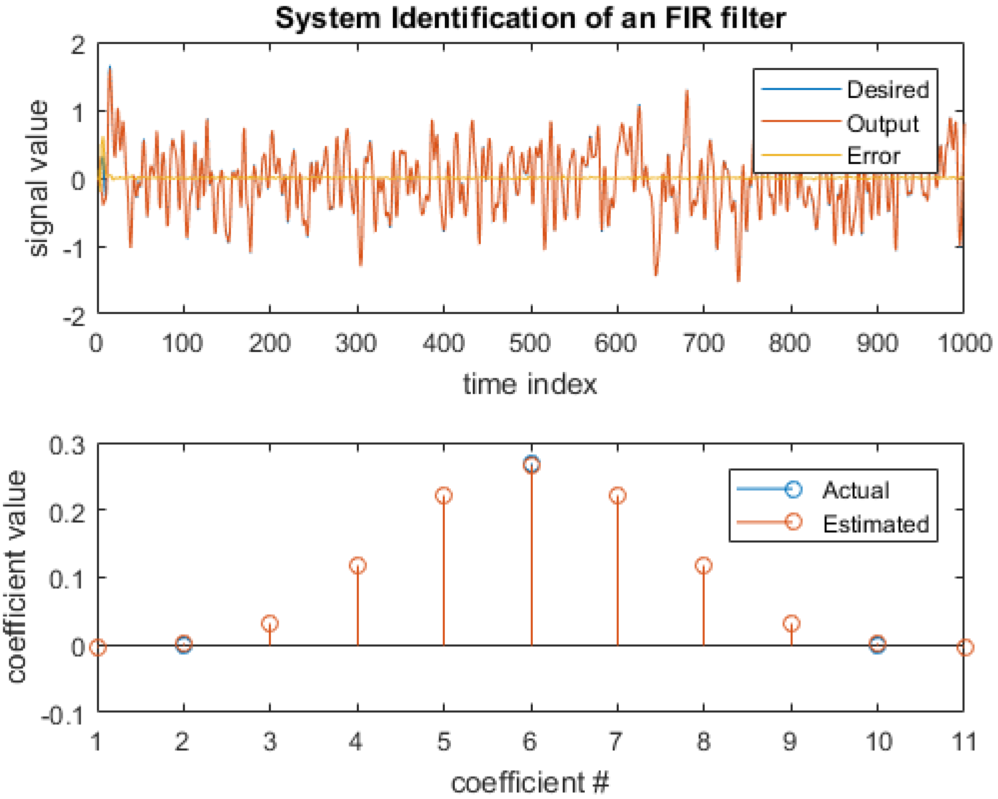
Examples

System Identification of an FIR Filter

Use the RLS filter System object™ to determine the signal value.

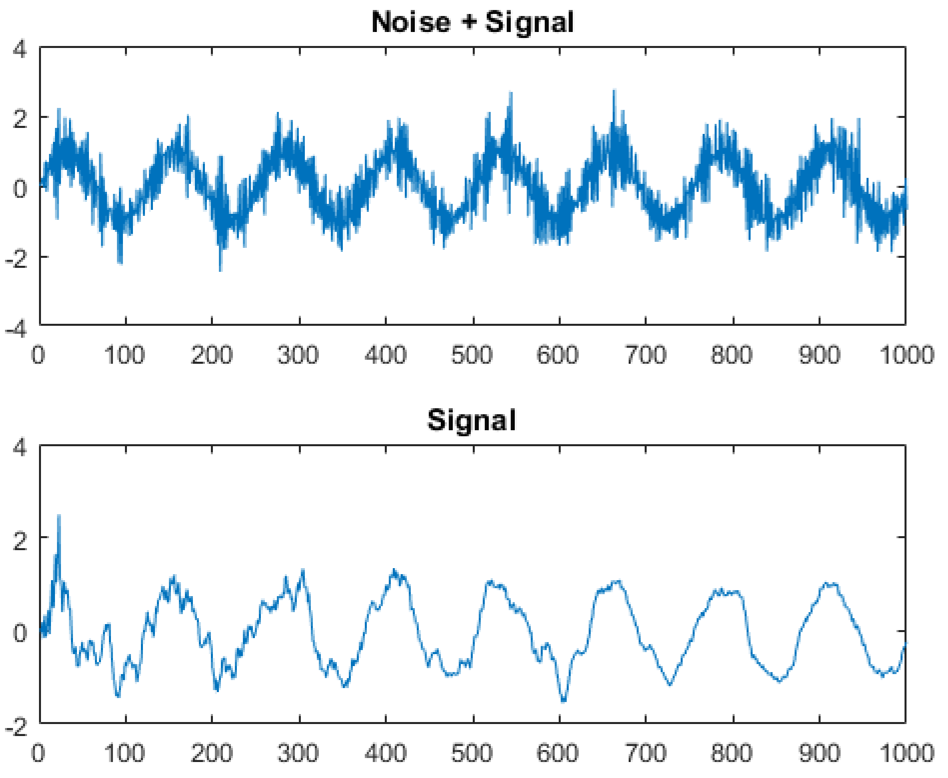
Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
rls1 = dsp.RLSFilter(11, 'ForgettingFactor', 0.98);
filt = dsp.FIRFilter('Numerator',fir1(10, .25)); % Unknown System
x = randn(1000,1);                               % input signal
d = filt(x) + 0.01*randn(1000,1); % desired signal
[y,e] = rls1(x, d);
w = rls1.Coefficients;
subplot(2,1,1), plot(1:1000, [d,y,e]);
title('System Identification of an FIR filter');
legend('Desired', 'Output', 'Error');
xlabel('time index'); ylabel('signal value');
subplot(2,1,2); stem([filt.Numerator; w].');
legend('Actual','Estimated');
xlabel('coefficient #'); ylabel('coefficient value');
```



Noise Cancellation

```
rls2 = dsp.RLSFilter('Length', 11, 'Method', 'Householder RLS');
filt2 = dsp.FIRFilter('Numerator',fir1(10, [.5, .75]));
x = randn(1000,1);                              % Noise
d = filt2(x) + sin(0:.05:49.95)';      % Noise + Signal
[y, err] = rls2(x, d);
subplot(2,1,1), plot(d), title('Noise + Signal');
subplot(2,1,2), plot(err), title('Signal');
```



Algorithms

The `dsp.RLSFilter` System object, when `Conventional RLS` is selected, recursively computes the least squares estimate (RLS) of the FIR filter weights. The System object estimates the filter weights or coefficients, needed to convert the input signal into the desired signal. The input signal can be a scalar or a column vector. The desired signal must have the same data type, complexity, and dimensions as the input signal. The corresponding RLS filter is expressed in matrix form as $\mathbf{P}(n)$:

$$\mathbf{k}(n) = \frac{\lambda^{-1}\mathbf{P}(n-1)\mathbf{u}(n)}{1+\lambda^{-1}\mathbf{u}^{H}(n)\mathbf{P}(n-1)\mathbf{u}(n)}$$

$$y(n) = \mathbf{w}^{T}(n-1)\mathbf{u}(n)$$

$$e(n) = d(n) - y(n)$$

$$\mathbf{w}(n) = \mathbf{w}(n-1) + \mathbf{k}(n)e(n)$$

$$\mathbf{P}(n) = \lambda^{-1}\mathbf{P}(n-1) - \lambda^{-1}\mathbf{k}(n)\mathbf{u}^{H}(n)\mathbf{P}(n-1)$$

where λ^{-1} denotes the reciprocal of the exponential weighting factor. The variables are as follows:

Variable	Description
n	The current time index
$\mathbf{u}(n)$	The vector of buffered input samples at step n
$\mathbf{P}(n)$	The inverse correlation matrix at step n
$\mathbf{k}(n)$	The gain vector at step n
$\mathbf{w}(n)$	The vector of filter tap estimates at step n
$y(\mathrm{n})$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
λ	The forgetting factor

u, **w**, and **k** are all column vectors.

References

[1] *M Hayes, Statistical Digital Signal Processing and Modeling, New York: Wiley, 1996*

[2] *S. Haykin, Adaptive Filter Theory,* 4th Edition, *Upper Saddle River, NJ: Prentice Hall, 2002*

[3] A.A. Rontogiannis and S. Theodoridis, "Inverse factorization adaptive least-squares algorithms," Signal Processing, vol. 52, no. 1, pp. 35-47, July 1996.

[4] S.C. Douglas, "Numerically-robust $O(N^2)$ RLS algorithms using least-squares prewhitening," Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, Istanbul, Turkey, vol. I, pp. 412-415, June 2000.

[5] *A. H. Sayed, Fundamentals of Adaptive Filtering, Hoboken, NJ: John Wiley & Sons, 2003*

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.FIRFilter | dsp.AffineProjectionFilter | dsp.LMSFilter | “Adaptive Noise Cancellation Using RLS Adaptive Filtering”

Introduced in R2013a

msesim

System object: dsp.RLSFilter
Package: dsp

Mean-square error for RLS filter

Syntax

```
MSE = msesim(sysObj,X,D)
[MSE,MEANW,W,TRACEK] = msesim(sysObj,X,D)
[...] = msesim(sysObj,X,D,M)
```

Description

`MSE = msesim(sysObj,X,D)` returns a sequence of mean-square errors. This column vector contains estimates of the mean-square error of the adaptive filter at each time instant. The length of `MSE` is equal to `SIZE(X,1)`. The columns of the matrix `X` contain individual input signal sequences, and the columns of the matrix `D` contain corresponding desired response signal sequences.

`[MSE,MEANW,W,TRACEK] = msesim(sysObj,X,D)` calculates three parameters corresponding to the simulated behavior of the adaptive filter defined by `sysObj`. `MEANW` is a sequence of coefficient vector means. The columns of this matrix contain estimates of the mean values of the adaptive filter coefficients at each time instant. The dimensions of `MEANW` are `(SIZE(X,1))` by `(sysObj.length)`. `W` is an estimate of the final values of the adaptive filter coefficients for the algorithm corresponding to `sysObj`. `TRACEK` is a sequence of total coefficient error powers. This column vector contains estimates of the total coefficient error power of the adaptive filter at each time instant. The length of `TRACEK` is equal to `SIZE(X,1)`.

`[...] = msesim(sysObj,X,D,M)` specifies an optional decimation factor for computing `MSE`, `MEANW`, and `TRACEK`. If `M` > `1`, every M^{th} predicted value of each of these sequences is saved. If omitted, the value of `M` is the default, which is 1.

reset

System object: dsp.RLSFilter
Package: dsp

Reset the internal states of a System object

Syntax

`reset(rlsFilt)`

Description

`reset(rlsFilt)` resets the internal states of the System object, `rlsFilt`, to their initial values.

For many System objects, this method is nonoperational. Objects that have internal states describe in their **help** what the reset method does for that object. The `reset` method is always nonoperational for unlocked System objects, as states may not be allocated when the object is not locked.

step

System object: dsp.RLSFilter
Package: dsp

Process inputs using RLS filter

Syntax

```
y = step(rlsFilt,x,d)
[y,e] = step(rlsFilt,x,d)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(rlsFilt,x,d)` recursively adapts the reference input, `x`, to match the desired signal, `d`, using the System object, `rlsFilt`. The desired signal,`d` is the signal desired plus some undesired noise. `[y,e] = step(rlsFilt,x,d)` shows the output of the RLS filter along with the error, `e`, between the reference input and the desired signal. The filters adapts its coefficients until the error, `e` is minimized. You can access these coefficients by accessing the 'Coefficients' property of the object. You must access this property after calling the `step` method on the object. For example, you can access the optimized coefficients of `rlsFilt` filter object through `rlsFilt.Coefficients`, after calling the `step` method on this object.

dsp.RMS System object

Package: dsp

Root mean square of vector elements

Description

The RMS object computes the root mean square (RMS) value.

To compute the RMS value of your input:

1. Define and set up your RMS calculation. See “Construction” on page 4-1489.
2. Call step to compute the RMS value for an input according to the properties of `dsp.RMS`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Note: The `Running` mode in the `dsp.RMS` System object will be removed in a future release. To compute the running RMS in MATLAB, use the dsp.MovingRMS System object instead.

Construction

`rms = dsp.RMS` returns a System object, `rms`, that computes the root mean square (RMS) of an input or a sequence of inputs over the specified `Dimension`.

`rms = dsp.RMS('PropertyName',PropertyValue,...)` returns an RMS System object, `rms`, with each specified property set to the specified value.

Properties

`RunningRMS`

Enable calculating RMS over time

Set this property to `true` to enable calculating the RMS over successive calls to the `step` method.

Default: `false`

`ResetInputPort`

Enable resetting in running RMS mode

Set this property to `true` to enable resetting the running RMS. When the property is set to `true`, you must specify a reset input to the `step` method to reset the running RMS. This property applies when you set the `RunningRMS` on page 4- property to `true`.

Default: `false`

`ResetCondition`

Reset condition for running RMS mode

Specify the event to reset the running RMS as one of `Rising edge`, `Falling edge`, `Either edge`, or `Non-zero`. `Non-zero` resets the running RMS each time a nonzero sample is acquired. See "Rising and Falling Edges" on page 4-1493 for definitions of rising and falling edges. This property applies when you set the `ResetInputPort` on page 4- property to `true`.

Default: `Non-zero`

`Dimension`

Dimension to compute RMS value along

Specify the dimension along which to calculate the RMS as one of `All`, `Row`, `Column`, or `Custom`. This property applies only when you set the `RunningRMS` on page 4- property to `false`. Specifying the `Dimension` property as `All` computes the RMS value over the entire input.

Default: `Column`

`CustomDimension`

Numerical dimension to operate along

Specify the dimension (one-based scalar integer value) of the input signal, along which the RMS is computed. The cannot exceed the number of dimensions in the input signal. This property applies when you set the `Dimension` on page 4- property to `Custom`.

Default: `1`

Methods

reset	Reset the running root mean square object
step	Root mean square of input

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

RMS Value of Vector Input

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Compute the RMS value of a vector consisting of the integers 1 to 10.

```
x = 1:10;
```

```
rms = dsp.RMS('Dimension','row');
rmsval = rms(x)


rmsval =

    6.2048
```

RMS Value of Matrix Input

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Compute the RMS value of a matrix with the `Dimension` property set to `'All'`.

```
in2 = magic(4);
rms2d = dsp.RMS;
rms2d.Dimension = 'All';
y_rms2 = rms2d(in2)


y_rms2 =

    9.6695
```

The output is equivalent to reshaping the 4-by-4 matrix into a 16-by-1, or 1-by-16 vector and computing the RMS value for the vector.

Definitions

Root-Mean-Square Level

The root-mean-square level of a vector, X, is

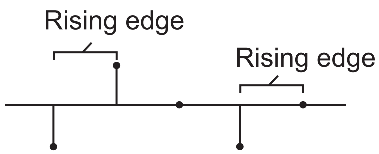
$$X_{\mathrm{RMS}} = \sqrt{\frac{1}{N}\sum_{n=1}^{N}|X_n|^2},$$

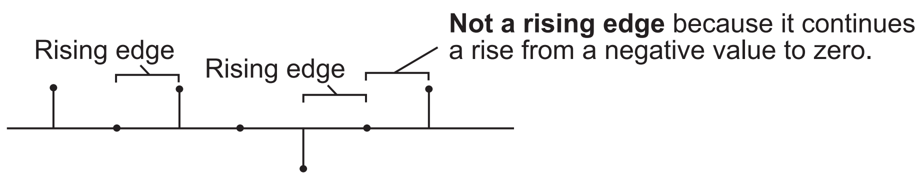
with the summation performed along the specified dimension.

Rising and Falling Edges

A rising edge:

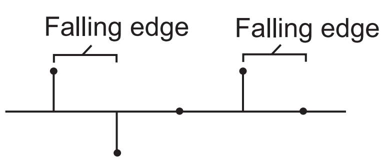
- Rises from a negative value to a positive value or zero.
- Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.

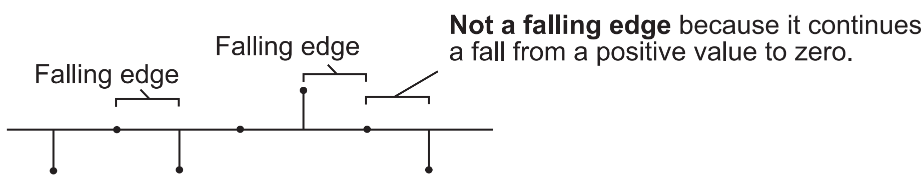




A falling edge:

- Falls from a positive value to a negative value or zero.
- Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.





Algorithms

This object implements the algorithm, inputs, and outputs described on the RMS block reference page. The object properties correspond to the Simulink block parameters, except:

- **Treat sample-based row input as a column** block parameter is not supported by the dsp.RMS object.
- **Reset Port** block parameter corresponds to both the ResetCondition and the ResetInputPort object properties.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

System Objects

dsp.MovingRMS | dsp.MovingStandardDeviation | dsp.StandardDeviation | dsp.MovingVariance | dsp.Variance | dsp.Minimum

Blocks

Moving RMS | Moving Standard Deviation | Moving Variance | RMS | Standard Deviation | Variance

Introduced in R2012a

reset

System object: dsp.RMS
Package: dsp

Reset the running root mean square object

Syntax

```
reset(rms)
```

Description

`reset(rms)` resets the running root mean square (RMS) for the object `rms`.

step

System object: dsp.RMS
Package: dsp

Root mean square of input

Syntax

```
Y = step(rms,X)
Y = step(rms,X,R)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(rms,X)` computes the root mean square (RMS) output, `Y` , of input vector `X`. When the `RunningRMS` property is true, `Y` corresponds to the `RMS` of the input elements over successive calls to the `step` method.

`Y = step(rms,X,R)` resets the running RMS state based on the value of `R`, the reset signal, and the `ResetCondition` property. This computation is possible when you set both the `RunningRMS` and the `ResetInputPort` properties to `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.SampleRateConverter System object

Package: dsp

Multistage sample rate converter

Description

The `SampleRateConverter` System object converts the sample rate of an incoming signal.

To convert the sample rate of a signal:

1. Define and set up your sample rate converter. See “Construction” on page 4-1497.
2. Call step to convert the sample rate according to the properties of `dsp.SampleRateConverter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`src = dsp.SampleRateConverter` creates a multistage FIR sample rate converter System object, `src`, that converts the sample rate of each channel of an input signal.

`src = dsp.SampleRateConverter(Name,Value)` returns a multistage FIR sample rate converter System object, `src`, with properties and options specified by one or more `Name,Value` pair arguments.

Properties

`Bandwidth` — Two-sided bandwidth of interest

40 kHz (default) | positive scalar

Specify the two-sided bandwidth of interest (after rate conversion) as a positive scalar expressed in hertz. This property is the two-sided bandwidth of the information-carrying portion of the signal that you wish to retain. The default is 40 kHz.

`InputSampleRate` — Sample rate of input signal

192 kHz (default) | positive scalar

Specify the sample rate of the input signal as a positive scalar expressed in hertz. The input sample rate must be greater than the bandwidth of interest. The default is 192 kHz.

`OutputRateTolerance` — Maximum allowed tolerance for output sample rate

0 (default) | positive scalar

Specify the maximum allowed tolerance for the sample rate of the output signal as a positive scalar between 0 and 1. The default is 0.

The output rate tolerance allows for a simpler design in many cases. The actual output sample rate varies but is within the specified range. For example, if `OutputRateTolerance` is specified as 0.01, then the actual output sample rate is in the range given by `OutputSampleRate` ± 1%.

`OutputSampleRate` — Sample rate of output signal

44.1 kHz (default) | positive scalar

Specify the sample rate of the output signal as a positive scalar expressed in hertz. The output sample rate must be greater than the bandwidth of interest. The default is 44.1 kHz.

`StopbandAttenuation` — Minimum dB attenuation for aliased components

80 dB (default) | positive scalar

Specify the stopband attenuation as a positive scalar expressed in decibels. This property is the minimum amount by which any aliasing involved in the process is attenuated. The default is 80 dB.

Methods

cost	Compute implementation cost

freqz	Frequency response
getFilters	Obtain single-stage filters
getActualOutputRate	Get actual output rate
getRateChangeFactors	Overall interpolation and decimation factors
info	Display information about sample rate converter
reset	Reset internal states of multistage sample rate converter
step	Convert sample rate of signal
visualizeFilterStages	Visualize filter stages

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Convert Sample Rate of Audio Signal

Convert the sample rate of an audio signal from 44.1 kHz (CD quality) to 96 kHz (DVD quality).

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
fs1 = 44.1e3;
fs2 = 96e3;

SRC = dsp.SampleRateConverter('Bandwidth',40e3,...
    'InputSampleRate',fs1,'OutputSampleRate',fs2);
```

```
[L,M] = getRateChangeFactors(SRC);
FrameSize = 10*M;

AR = dsp.AudioFileReader('guitar10min.ogg', ...
    'SamplesPerFrame',FrameSize);
AW = dsp.AudioFileWriter('guitar10min_96k.wav', ...
    'SampleRate',fs2);
```

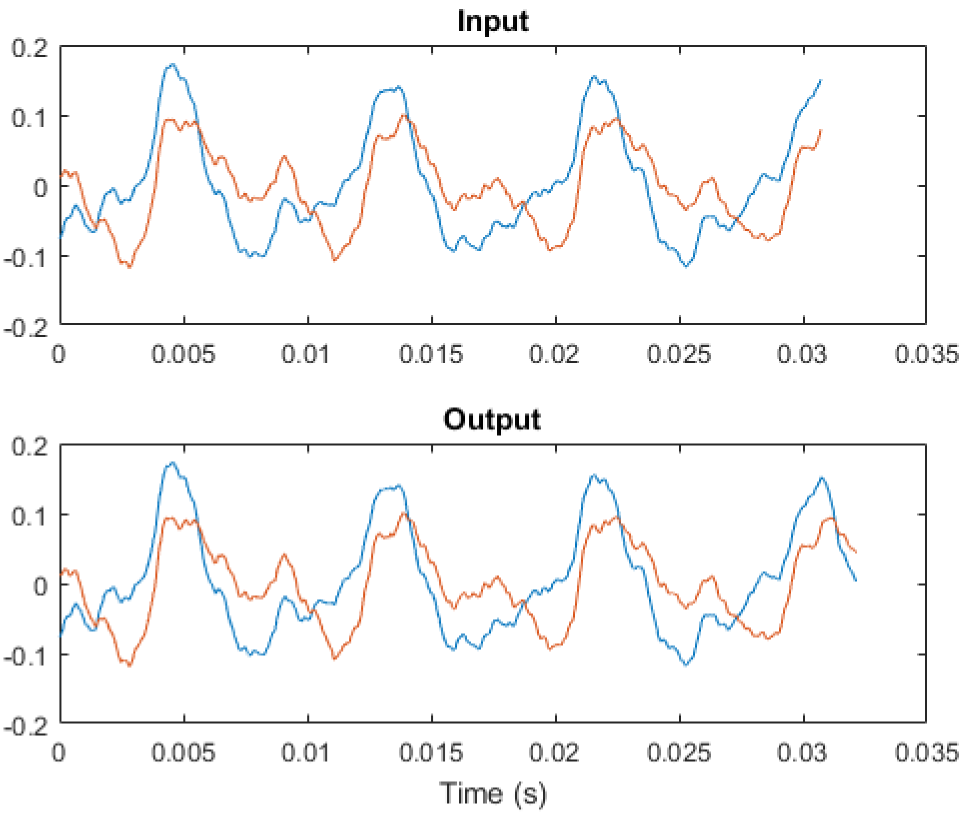
Run the system for 15 s. Release all objects.

```
tic
while toc < 15
    x = AR();
    y = SRC(x);
    AW(y);
end

release(AR);
release(AW);
release(SRC);
```

Plot the input and output signals. Use a different set of axes for each signal. Shift the output to compensate for the delay introduced by the filter.

```
t1 = 0:1/fs1:1/30-1/fs1;
t2 = 0:1/fs2:1/30-1/fs2;

delay = 114;

el1 = 1:length(t1)-delay;

el2 = 1:length(t2);
el2(1:delay) = [];

subplot(2,1,1)
plot(t1(1:length(el1)),x(el1,1))
hold on
plot(t1(1:length(el1)),x(el1,2))
title('Input')

subplot(2,1,2)
plot(t2(1:length(el2)),y(el2,1))
hold on
plot(t2(1:length(el2)),y(el2,2))
```

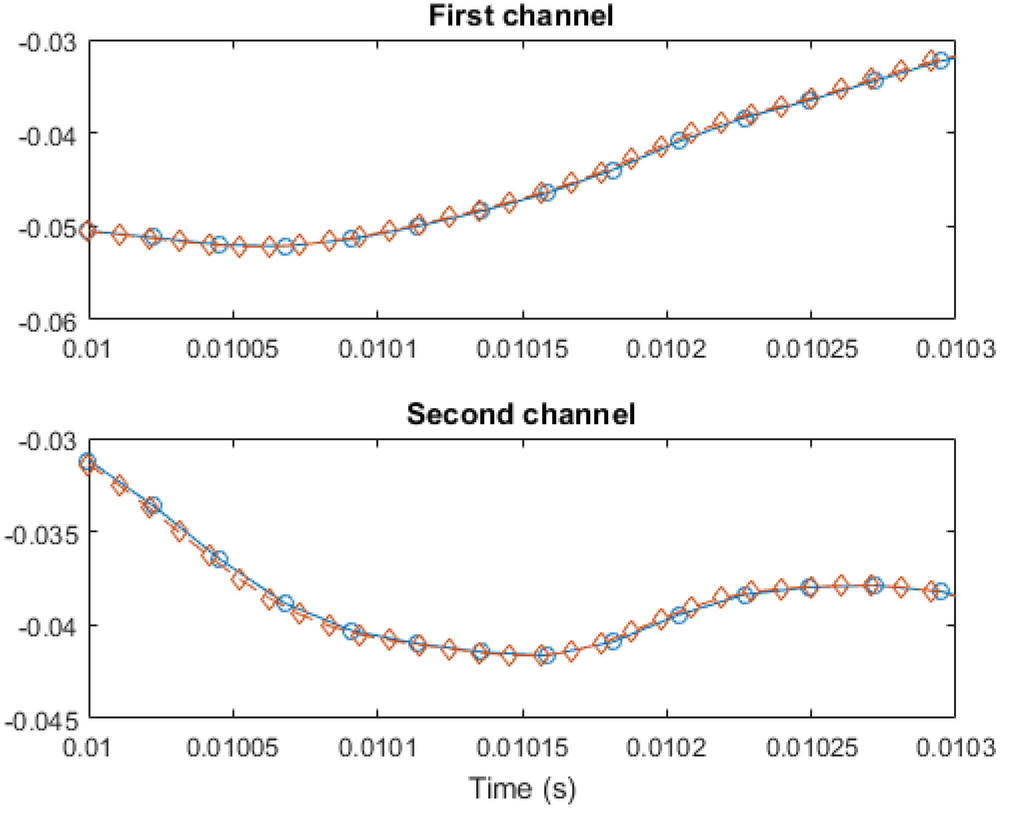
```
xlabel('Time (s)')
title('Output')
```



Zoom in to see the difference in sample rates. Use a different set of axes for each channel.

```
figure

subplot(2,1,1)
plot(t1(1:length(el1)),x(el1,1),'o-')
hold on
plot(t2(1:length(el2)),y(el2,1),'d--')
xlim([0.01 0.0103])
title('First channel')
```

```
subplot(2,1,2)
plot(t1(1:length(el1)),x(el1,2),'o-')
hold on
plot(t2(1:length(el2)),y(el2,2),'d--')
xlim([0.01 0.0103])
xlabel('Time (s)')
title('Second channel')
```



Tolerance Cost in Sample Rate Conversion

A signal output from an A/D converter is sampled at 98.304 MHz. The signal has a bandwidth of 20 MHz. Reduce the sample rate of the signal to 22 MHz, which is the bandwidth of 802.11 channels. Make the conversion exactly and then redo it with an output rate tolerance of 1%.

```
SRC1 = dsp.SampleRateConverter('Bandwidth',20e6, ...
    'InputSampleRate',98.304e6,'OutputSampleRate',22e6, ...
    'OutputRateTolerance',0);
SRC2 = dsp.SampleRateConverter('Bandwidth',20e6, ...
    'InputSampleRate',98.304e6,'OutputSampleRate',22e6, ...
    'OutputRateTolerance',0.01);
```

Use the `cost` method to determine the cost of each sample rate conversion. The zero-tolerance process requires more than 500 times as many coefficients as the 1% process.

```
c1 = cost(SRC1)
c2 = cost(SRC2)
```

```
c1 = 

  struct with fields:

                  NumCoefficients: 84779
                        NumStates: 133
    MultiplicationsPerInputSample: 27.0422
          AdditionsPerInputSample: 26.0684


c2 = 

  struct with fields:

                  NumCoefficients: 150
                        NumStates: 127
    MultiplicationsPerInputSample: 22.6667
          AdditionsPerInputSample: 22.1111
```

Find the integer upsampling and downsampling factors used in each conversion.

```
[L1,M1] = getRateChangeFactors(SRC1)
[L2,M2] = getRateChangeFactors(SRC2)
```

```
L1 =

        1375
```

```
M1 =

        6144


L2 =

     2


M2 =

     9
```

Compute the actual sample rate of the output signal when the sample rate conversion has a tolerance of 1%.

```
getActualOutputRate(SRC2)
```

```
ans =

   2.1845e+07
```

Algorithms

- The general multistage sample rate converter performs a multistage decimation, a single-stage sample rate conversion, and a multistage interpolation, in that order. Actual designs include at most two of those steps.
- The procedure determines automatically the optimal number of decimation or interpolation stages. In special cases, the decimation or the interpolation can be performed in a single stage.
- The algorithm always attempts to start by reducing the sample rate. This decreases the amount of computation required. The decimation step is designed so that no intermediate sample rate goes below the bandwidth of interest. This ensures that no information is filtered out.
- Each individual stage uses halfband or Nyquist filters to minimize the number of nonzero coefficients.

- Transition-band aliasing is allowed because it decreases the implementation cost. The signal within the bandwidth of interest is kept alias free up to the value specified by the `StopbandAttenuation` property.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

Introduced in R2014b

cost

System object: dsp.SampleRateConverter
Package: dsp

Compute implementation cost

Syntax

```
c = cost(src)
```

Description

`c = cost(src)` returns a structure, `c`, whose fields contain information about the computational cost of implementing a multistage sample rate converter, `src`.

Input Arguments

`src` — Multistage sample rate converter

`SampleRateConverter` System object

Multistage sample rate converter, specified as a `SampleRateConverter` System object.

Output Arguments

`c` — Output structure

structure

Output structure with information about the computational cost of `src`:

- The number of coefficients,
- The number of states,
- The number of multiplications per unit sample, and
- The number of additions per unit sample.

Examples

Computational Cost of a Sample Rate Converter

Create `src`, a multistage sample rate converter with default values. `src` combines three filter stages to convert from 192 kHz to 44.1 kHz. Determine its computational cost: the number of coefficients, the number of states, the number of multiplications per unit sample, and the number of additions per unit sample.

```
src = dsp.SampleRateConverter;
cst = cost(src)
```

```
cst = 

  struct with fields:

                 NumCoefficients: 8631
                       NumStates: 138
    MultiplicationsPerInputSample: 27.6672
          AdditionsPerInputSample: 26.6875
```

Repeat the computation allowing a tolerance of 10% in the output sample rate.

```
src.OutputRateTolerance = 0.1;
ctl = cost(src)
```

```
ctl = 

  struct with fields:

                 NumCoefficients: 44
                       NumStates: 80
    MultiplicationsPerInputSample: 14.2500
          AdditionsPerInputSample: 13.5000
```

freqz

System object: dsp.SampleRateConverter
Package: dsp

Frequency response

Syntax

```
[h,f] = freqz(src,n,range)
[h,f] = freqz(src,f)
```

Description

`[h,f] = freqz(src,n,range)` returns the complex frequency response, `h`, of the multistage sample rate converter, `src`, evaluated at the `n` frequencies returned in `f`. `range` is the frequency range over which the response is computed. The sample rate is taken to be the largest of `InputSampleRate` and `OutputSampleRate`.

`[h,f] = freqz(src,f)` returns the complex frequency response evaluated at the frequency points specified in the vector `f`. `f` is assumed to be expressed in hertz.

Input Arguments

`src` — Multistage sample rate converter

`SampleRateConverter` System object

Multistage sample rate converter, specified as a `SampleRateConverter` System object.

`n` — Number of evaluation points

512 (default) | positive integer

Number of frequencies for response evaluation, specified as a positive integer scalar. If `n` is not specified, it defaults to 512.

`range` — Range of frequencies

`'half'` (default) | `'whole'`

Range considered when computing the frequency response, specified as either `'half'` (from 0 to π) or `'whole'` (from 0 to 2π). If `range` is not specified, it defaults to `'half'`.

Output Arguments

`h` — Complex frequency response

vector

Complex frequency response, returned as a vector.

`f` — Frequencies

vector

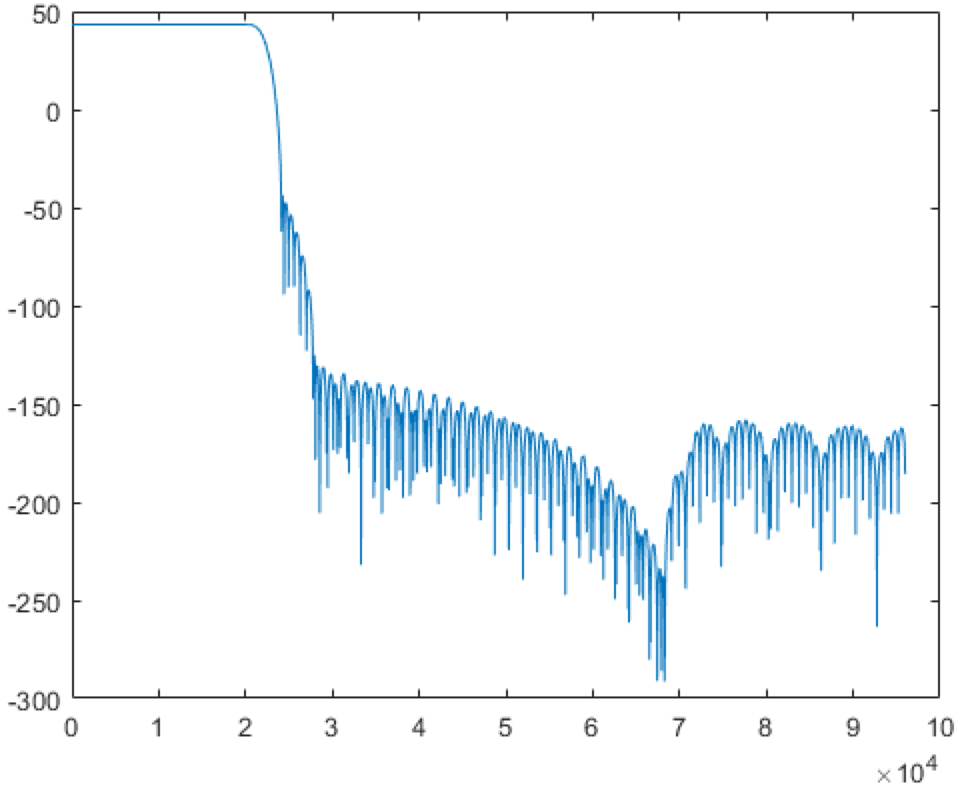
Frequencies at which the response is evaluated, returned as a vector.

Examples

Frequency Response of Default Converter

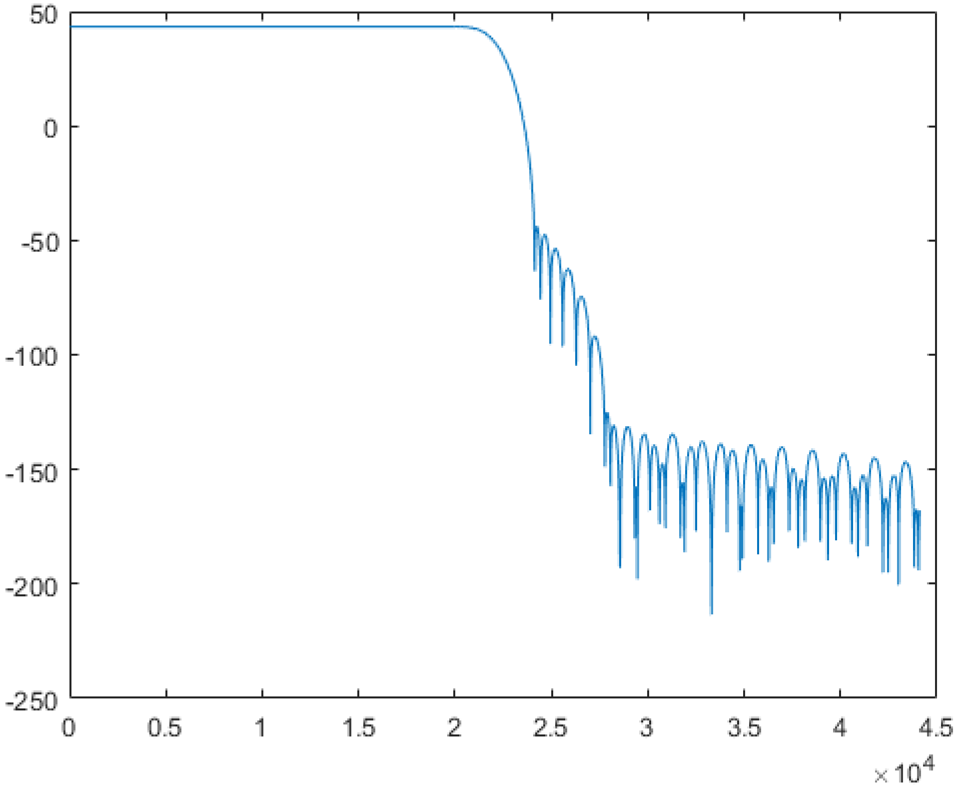
Create a multistage sample rate converter with default properties, corresponding to the combined three filter stages used to convert from 192 kHz to 44.1 kHz. Compute and display the frequency response.

```
src = dsp.SampleRateConverter;
[H,f] = freqz(src);
plot(f,20*log10(abs(H)))
```



Compute and display the frequency response over the range between 20 Hz and 44.1 kHz.

```
f = 20:10:44.1e3;
[H,f] = freqz(src,f);
plot(f,20*log10(abs(H)))
```



50
0
-50
-100
-150
-200
-250
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
× 10^4

getFilters

System object: dsp.SampleRateConverter
Package: dsp

Obtain single-stage filters

Syntax

```
c = getFilters(src)
```

Description

`c = getFilters(src)` returns the multirate filters cascaded together in `src` to perform the overall sample rate conversion. The result is a `FilterCascade` structure, `c`. Each field of `c` holds the filter used at a particular stage and gives access to its coefficients and rate-change factors.

Input Arguments

`src` — Multistage sample rate converter

`SampleRateConverter` System object

Multistage sample rate converter, specified as a `SampleRateConverter` System object.

Output Arguments

`c` — Single-stage filters

`FilterCascade` structure

Single-stage filters, returned as a `FilterCascade` structure.

Examples

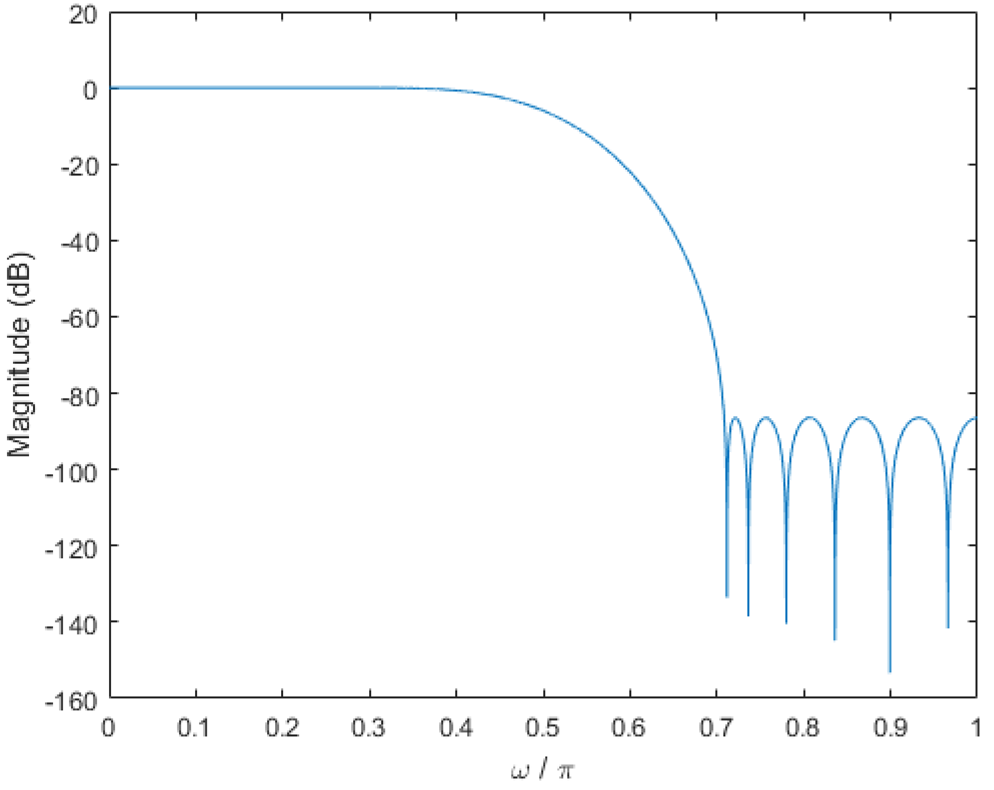
Single-Stage Filters

Create `src`, a multistage sample rate converter with default properties. `src` converts between 192 kHz and 44.1 kHz. Find the individual filters that are cascaded together to perform the conversion.

```
src = dsp.SampleRateConverter;
c = getFilters(src);
```

Visualize the frequency response of the decimator used in the first stage of the process.

```
m = c.Stage1;

[h,w] = freqz(m);
plot(w/pi,20*log10(abs(h)))
xlabel('\omega / \pi')
ylabel('Magnitude (dB)')
```



20
0
-20
-40
-60
-80
-100
-120
-140
-160
Magnitude (dB)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
ω / π

getActualOutputRate

System object: dsp.SampleRateConverter
Package: dsp

Get actual output rate

Syntax

```
fsout = getActualOutputRate(src)
```

Description

`fsout = getActualOutputRate(src)` returns the actual output sample rate yielded by a `SampleRateConverter` System object. The computation takes into account the `OutputRateTolerance` parameter.

Input Arguments

`src` — Multistage sample rate converter

`SampleRateConverter` System object

Multistage sample rate converter, specified as a `SampleRateConverter` System object.

Output Arguments

`fsout` — Actual output sample rate

scalar

Actual output sample rate, returned as a scalar expressed in hertz.

Examples

Output Sample Rate with Given Tolerance

Get the actual output sample rate for conversion between 192 kHz and 44.1 kHz when given a tolerance of 1%.

```
src = dsp.SampleRateConverter;
src.OutputRateTolerance = 0.01;
FsOut = getActualOutputRate(src)
```

```
FsOut =

   4.4308e+04
```

getRateChangeFactors

System object: dsp.SampleRateConverter
Package: dsp

Overall interpolation and decimation factors

Syntax

`[L,M] = getRateChangeFactors(src)`

Description

`[L,M] = getRateChangeFactors(src)` returns the overall interpolation factor, `L`, and the overall decimation factor, `M`, corresponding to the multistage sample rate converter, `src`. The overall decimation factor affects the allowable frame size of the input signal, which must be an integer multiple of `M`.

Input Arguments

`src` — Multistage sample rate converter

`SampleRateConverter` System object

Multistage sample rate converter, specified as a `SampleRateConverter` System object.

Output Arguments

`L` — Overall interpolation factor

scalar

Overall interpolation factor, returned as a scalar.

`M` — Overall decimation factor

scalar

Overall decimation factor, returned as a scalar.

Examples

Default Resampling Factors

Create `src`, a multistage sample rate converter with default properties. `src` combines three filter stages to convert from 192 kHz to 44.1 kHz. Determine its overall interpolation and decimation factors.

```
src = dsp.SampleRateConverter;
[L,M] = getRateChangeFactors(src)
```

```
L =

   147


M =

   640
```

info

System object: dsp.SampleRateConverter
Package: dsp

Display information about sample rate converter

Syntax

`info(src)`

Description

`info(src)` displays information about the multistage `SampleRateConverter` System object, `src`.

Input Arguments

`src` — Multistage sample rate converter

`SampleRateConverter` System object

Multistage sample rate converter, specified as a `SampleRateConverter` System object.

Examples

Default Multistage Sample Rate Converter

Create a multistage sample rate converter with default properties, corresponding to the combined three filter stages used to convert from 192 kHz to 44.1 kHz. Display information about the design.

```
src = dsp.SampleRateConverter
info(src)
```

```
src =
```

```
  dsp.SampleRateConverter with properties:

        InputSampleRate: 192000
       OutputSampleRate: 44100
    OutputRateTolerance: 0
              Bandwidth: 40000
    StopbandAttenuation: 80


ans =

    'Overall Interpolation Factor    : 147
     Overall Decimation Factor       : 640
     Number of Filters               : 3
     Multiplications per Input Sample: 27.667188
     Number of Coefficients          : 8631
     Filters:
        Filter 1:
        dsp.FIRDecimator     - Decimation Factor   : 2
        Filter 2:
        dsp.FIRDecimator     - Decimation Factor   : 2
        Filter 3:
        dsp.FIRRateConverter - Interpolation Factor: 147
                             - Decimation Factor   : 160
     '
```

reset

System object: dsp.SampleRateConverter
Package: dsp

Reset internal states of multistage sample rate converter

Syntax

`reset(src)`

Description

`reset(src)` resets the internal states of a multistage `SampleRateConverter` System object, `src`, to their initial values.

Input Arguments

`src` — Multistage sample rate converter
`SampleRateConverter` System object

Multistage sample rate converter, specified as a `SampleRateConverter` System object.

Examples

Reset a Sample Rate Converter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create a multistage sample rate converter with default properties, corresponding to the combined three filter stages used to convert from 192 kHz to 44.1 kHz. Determine its overall decimation and interpolation factors.

```
src = dsp.SampleRateConverter;
```

```
[L,M] = getRateChangeFactors(src);
```

Create a two-channel random signal. Specify a number of samples equal to the decimation factor. Apply the `step` method twice on the signal.

```
x = randn(M,2);

y1 = src(x);
y2 = src(x);

no = all(y2==y1)
```

```
no =

  1×2 logical array

   0   0
```

The output is different because the internal states of `src` have changed. Use `reset` to reset the converter and call the converter again. Verify that the output is unchanged.

```
reset(src)

y3 = src(x);

yes = all(y3==y1)
```

```
yes =

  1×2 logical array

   1   1
```

step

System object: dsp.SampleRateConverter
Package: dsp

Convert sample rate of signal

Syntax

`y = step(src,x)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(src,x)` returns a signal, `y`, corresponding to the input signal, `x`, with sample rate converted by `src`.

Input Arguments

`src` — Multistage sample rate converter

`SampleRateConverter` System object

Multistage sample rate converter, specified as a `SampleRateConverter` System object.

`x` — Input signal

vector | matrix

Input signal, specified as a vector or matrix. The row length of `x` must be a multiple of the overall decimation factor. Each column of `x` is treated as a separate channel.

Output Arguments

y — Resampled signal
vector | matrix

Resampled signal, returned as a vector or matrix.

Examples

Convert the Sample Rate of a Sinusoid

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create a multistage sample rate converter with default properties. The converter converts from 192 kHz to 44.1 kHz in three stages.

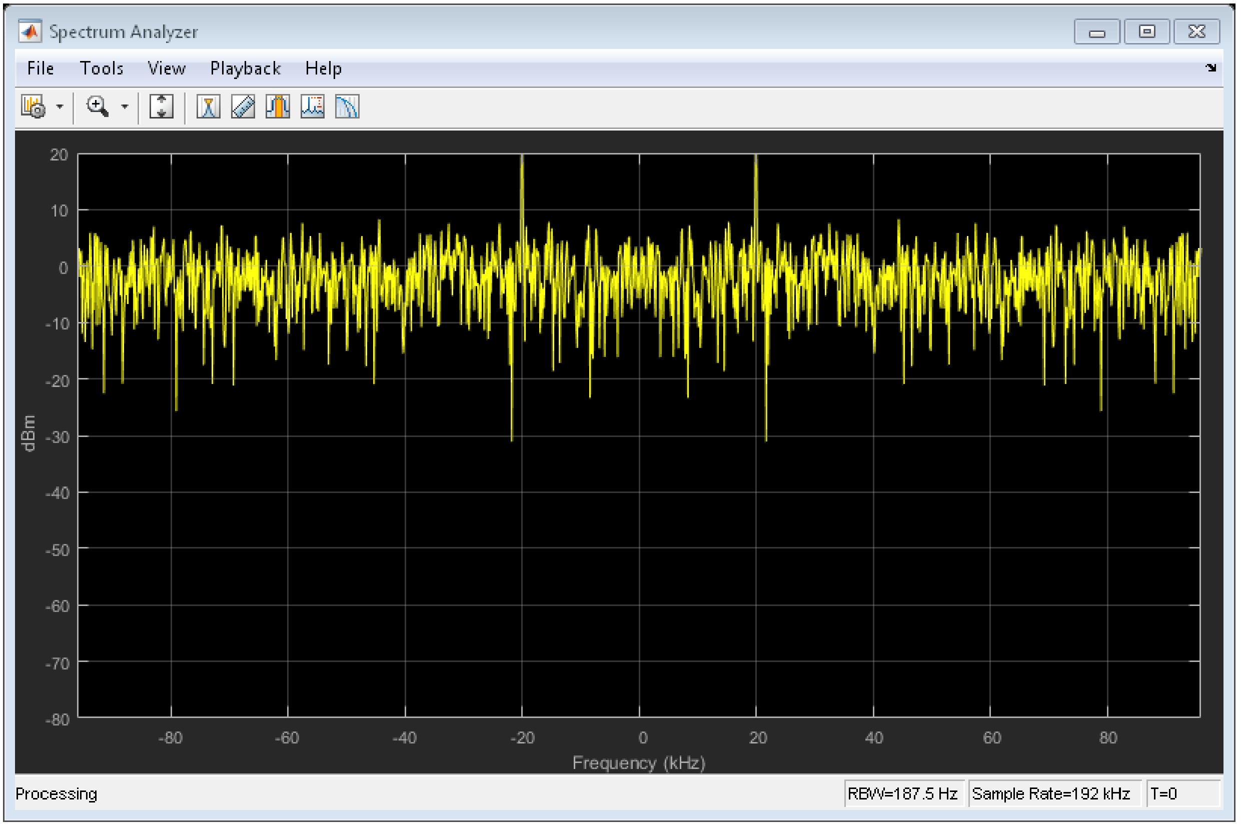
```
src = dsp.SampleRateConverter;
```

Use `src` to convert the sample rate of a noisy sinusoid. The sinusoid has a frequency of 20 kHz and is sampled for 0.1 s.

```
f = 20e3;

FsIn = src.InputSampleRate;
FsOut = src.OutputSampleRate;

t1 = (0:1/FsIn:0.1-1/FsIn)';

sIn = sin(2*pi*f*t1) + randn(size(t1));
```

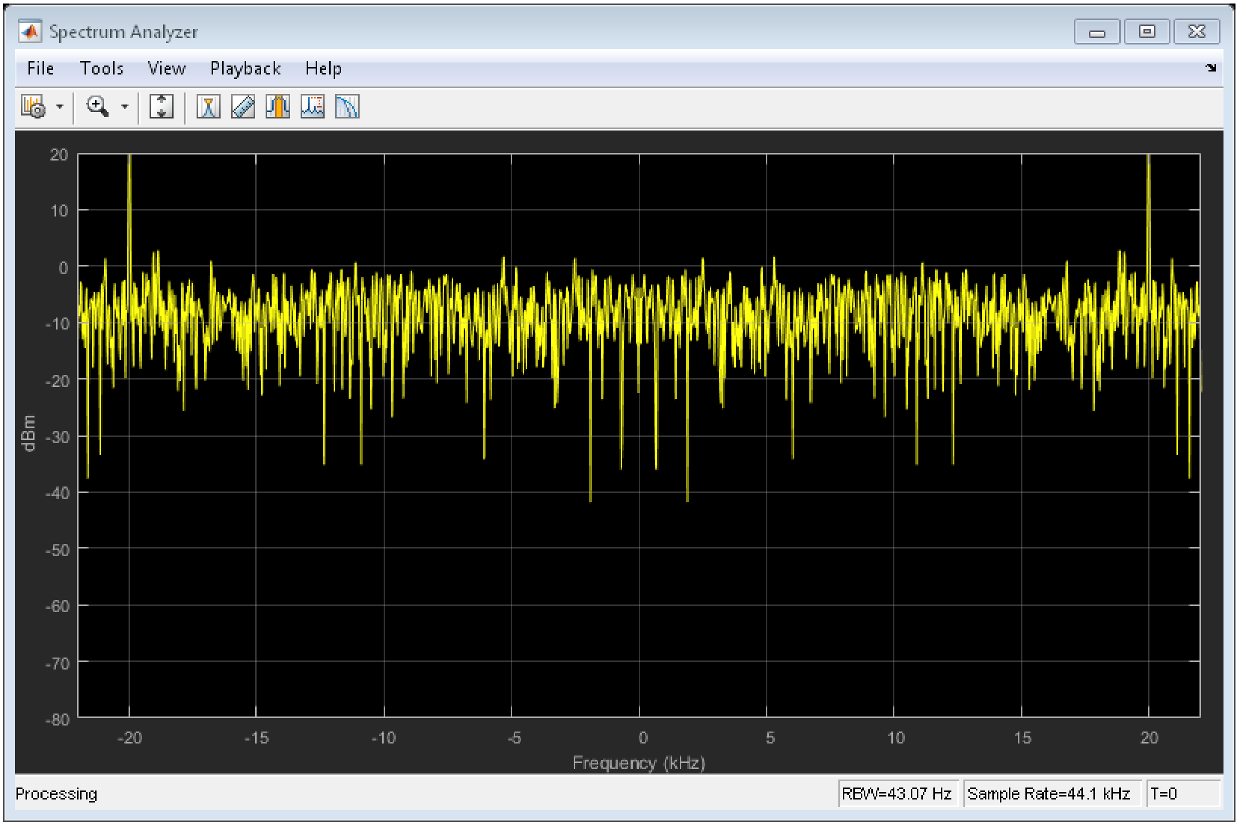
Estimate the power spectral density of the input.

```
hsa = dsp.SpectrumAnalyzer('SampleRate',FsIn);
hsa(sIn)
```



Convert the sample rate of the signal. Estimate the power spectral density of the output.

```
sOut = src(sIn);

hsb = dsp.SpectrumAnalyzer('SampleRate',FsOut);
hsb(sOut)
```


Spectrum Analyzer
File
Tools
View
Playback
Help
20
10
0
-10
-20
-30
-40
-50
-60
-70
-80
dBm
-20
-15
-10
-5
0
5
10
15
20
Frequency (kHz)
Processing
RBW=43.07 Hz
Sample Rate=44.1 kHz
T=0

visualizeFilterStages

System object: dsp.SampleRateConverter
Package: dsp

Visualize filter stages

Syntax

```
visualizeFilterStages(src)
```

Description

`visualizeFilterStages(src)` shows the response of each individual filter stage of `src` using `FVTool`.

Input Arguments

`src` — Multistage sample rate converter

`SampleRateConverter` System object

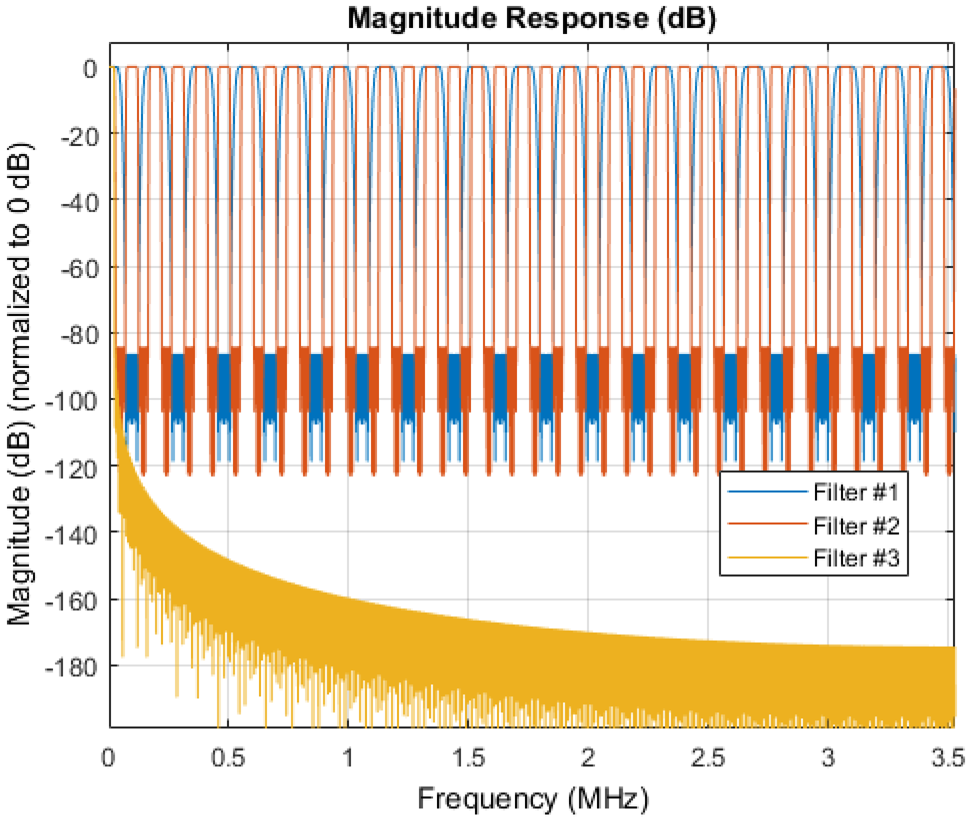
Multistage sample rate converter, specified as a `SampleRateConverter` System object.

Examples

Sample Rate Converter Stages

Create a multistage sample rate converter with default properties, corresponding to the combined three filter stages used to convert from 192 kHz to 44.1 kHz. Visualize the stages.

```
src = dsp.SampleRateConverter;
visualizeFilterStages(src)
```



Magnitude Response (dB)
Magnitude (dB) (normalized to 0 dB)
0
-20
-40
-60
-80
-100
-120
-140
-160
-180
0
0.5
1
1.5
2
2.5
3
3.5
Frequency (MHz)
Filter #1
Filter #2
Filter #3

dsp.ScalarQuantizerDecoder System object

Package: dsp

Convert each index value into quantized output value

Description

The `ScalarQuantizerDecoder` object converts each index value into a quantized output value. The specified codebook defines the set of all possible quantized output values or codewords. Input index values less than 0 are set to 0 and index values greater $N-1$ are set to $N-1$. N is the length of the codebook vector.

To convert an index value into a quantized output value:

1. Define and set up your scalar quantizer decoder. See "Construction" on page 4-1529.
2. Call step to convert the index value according to the properties of `dsp.ScalarQuantizerDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`sqdec = dsp.ScalarQuantizerDecoder` returns a scalar quantizer decoder System object, `sqdec`, that transforms zero-based input index values into quantized output values.

`sqdec = dsp.ScalarQuantizerDecoder('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a scalar quantizer decoder object, `sqdec`, with each specified property set to the specified value.

Properties

CodebookSource

How to specify codebook values

Specify how to determine the codebook values as `Property` or `Input port`. The default is `Property`.

Codebook

Codebook

Specify the codebook as a vector of quantized output values that correspond to each index value. The default is `1:10`. This property is tunable.

OutputDataType

Data type of codebook and quantized output

Specify the data type of the codebook and quantized output values as `Same as input`, `double`, `single` or `Custom`. The default is `double`.

Fixed-Point Properties

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a signed or unsigned `numerictype` object. This property applies only when you set the `OutputDataType` on page 4-1530 property to `Custom`. The default is `numerictype(true,16)`.

Methods

step	Decode using scalar quantization

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object

Common to All System Objects	
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Quantize a Signal

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Given a codebook and index values as inputs, determine the corresponding output quantized values.

```
codebook = single([-2.1655 -1.3238 -0.7365 -0.2249 0.2726, ...
    0.7844 1.3610 2.1599]);
indices = uint8([1 3 5 7 6 4 2 0]);
sqdec = dsp.ScalarQuantizerDecoder;
sqdec.CodebookSource = 'Input port';
qout = sqdec(indices, codebook);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Scalar Quantizer Decoder` block reference page. The object properties correspond to the block parameters, except:
There is no object property that directly corresponds to the **Action for out of range index value** block parameter. The object sets any index values less than `0` to `0` and any index values greater than or equal to N to $N-1$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.ScalarQuantizerEncoder | dsp.VectorQuantizerDecoder

Introduced in R2012a

step

System object: dsp.ScalarQuantizerDecoder
Package: dsp

Decode using scalar quantization

Syntax

```
Q = step(sqdec, I)
Q = step(sqdec,I,C)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Q = step(sqdec, I)` returns the quantized output values `Q` corresponding to the input indices `I`. The data type of `I` can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. The `OutputDataType` property determines the data type for `Q`.

`Q = step(sqdec,I,C)` uses input `C` as the codebook values when you set the `CodebookSource` property to `Input port`. The data type of `C` can be `double`, `single`, or `fixed-point`. The output `Q` has the same data type as the codebook input `C`.

dsp.ScalarQuantizerEncoder System object

Package: dsp

Associate input value with index value of quantization region

Description

The `ScalarQuantizerEncoder` object encodes each input value by associating that value with the index value of the quantization region. Then, the object outputs the index of the associated region.

To encode an input value by associating it with an index value of the quantization region:

1. Define and set up your scalar quantizer encoder. See "Construction" on page 4-1534.
2. Call step to encode the input value according to the properties of `dsp.ScalarQuantizerEncoder`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`sqenc = dsp.ScalarQuantizerEncoder` returns a scalar quantizer encoder System object, `sqenc`. This object maps each input value to a quantization region by comparing the input value to the user-specified boundary points.

`sqenc = dsp.ScalarQuantizerEncoder('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a scalar quantizer encoder object, `sqenc`, with each specified property set to the specified value.

Properties

BoundaryPointsSource

Source of boundary points

Specify how to determine the boundary points and codebook values as `Property` or `Input port`. The default is `Property`.

Partitioning

Quantizer is bounded or unbounded

Specify the quantizer as `Bounded` or `Unbounded`. The default is `Bounded`.

BoundaryPoints

Boundary points of quantizer regions

Specify the boundary points of quantizer regions as a vector. The vector values must be in ascending order. Let `[p0 p1 p2 p3 ... pN]` denote the boundary points property in the quantizer. If the quantizer is bounded, the object uses this property to specify `[p0 p1 p2 p3 ... pN]`. If the quantizer is unbounded, the object uses this property to specify `[p1 p2 p3 ... p(N-1)]` and sets `p0 = -Inf` and `pN = +Inf`. This property applies when you set the `BoundaryPointsSource on page 4-1535` property to `Property`. The default is `1:10`. This property is tunable.

SearchMethod

Find quantizer index by linear or binary search

Specify whether to find the appropriate quantizer index using a linear search or a binary search as one of `Linear` or `Binary`. The computational cost of the linear search method is of the order P and the computational cost of the binary search method is of the order

$$\log_2(\mathrm{P})$$

where P is the number of boundary points. The default is `Linear`.

TiebreakerRule

Behavior when input equals boundary point

Specify whether the input value is assigned to the lower indexed region or higher indexed region when the input value equals boundary point by selecting `Choose the lower index` or `Choose the higher index`. The default is `Choose the lower index`.

CodewordOutputPort

Enable output of codeword value

Set this property to `true` to output the codeword values that correspond to each index value. The default is `false`.

QuantizationErrorOutputPort

Enable output of quantization error

Set this property to `true` to output the quantization error for each input value. The quantization error is the difference between the input value and the quantized output value. The default is `false`.

Codebook

Codebook

Specify the codebook as a vector of quantized output values that correspond to each region. If the `Partitioning` property is `Bounded` and the boundary points vector has length N, you must set this property to a vector of length N-1. If the `Partitioning` property is `Unbounded` and the boundary points vector has length N, you must set this property to a vector of length N+1. This property applies when you set the `BoundaryPointsSource on page 4-1535` property to `Property` and either the `CodewordOutputPort on page 4-1536` property or the `QuantizationErrorOutputPort on page 4-1536` property is `true`. The default is `1.5:9.5`. This property is tunable.

ClippingStatusOutputPort

Enable output of clipping status

Set this property to `true` to output the clipping status. The output is a 1 when an input value is outside the range defined by the `BoundaryPoints on page 4-1535` property.

When the value is inside the range, the exception output is a 0. This property applies when you set the `Partitioning on page 4-1535` property to `Bounded`. The default is `false`.

`OutputIndexDataType`

Data type of the index output

Specify the data type of the index output from the object as: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`. The default is `int32`.

Fixed-Point Properties

`RoundingMethod`

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest` or `Zero`. The default is `Floor`.

`OverflowAction`

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

Methods

step	Encode using scalar quantization

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

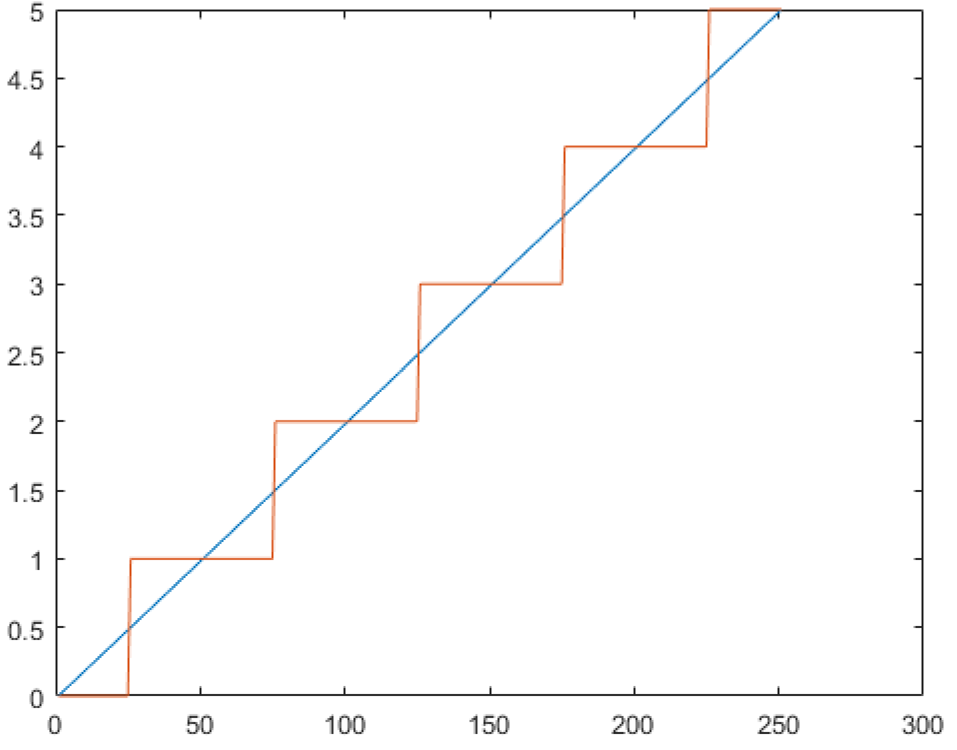
Examples

Quantize the Varying Fractional Inputs

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Quantize the varying fractional inputs between zero and five to the closest integers, and then plot the results.

```
sqenc = dsp.ScalarQuantizerEncoder;
sqenc.BoundaryPoints = [-.001 .499 1.499 ...
    2.499 3.499 4.499 5.001];
sqenc.CodewordOutputPort = true;
sqenc.Codebook = [0 1 2 3 4 5];
input = (0:0.02:5)';
[index, quantizedValue] = sqenc(input);
plot(1:length(input), [input quantizedValue]);
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Scalar Quantizer Encoder` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

dsp.ScalarQuantizerDecoder | dsp.VectorQuantizerEncoder

Introduced in R2012a

step

System object: dsp.ScalarQuantizerEncoder
Package: dsp

Encode using scalar quantization

Syntax

```
INDEX= step(sqenc,INPUT)
[...] = step(sqenc,INPUT,BPOINTS)
[...] = step(sqenc,INPUT,BPOINTS,CODEBOOK)
[...,CODEWORD] = step(sqenc, ...)
[..., QERR] = step(sqenc, ...)
[...,CLIPSTATUS] = step(sqenc, ...)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`INDEX= step(sqenc,INPUT)` returns the `INDEX` of the quantization region to which the `INPUT` belongs. The input data, boundary points, codebook values, quantized output values, and the quantization error must have the same data type whenever they are present.

`[...] = step(sqenc,INPUT,BPOINTS)` when the `BoundaryPointsSource` property is `Input port`, uses input `BPOINTS` as the boundary points .

`[...] = step(sqenc,INPUT,BPOINTS,CODEBOOK)` uses input `BPOINTS` as the boundary points and input `CODEBOOK` as the codebook when the `BoundaryPointsSource` property is `Input port` and either the `CodewordOutputPort` property or the `QuantizationErrorOutputPort` property is `true`.

`[...,CODEWORD] = step(sqenc, ...)` outputs the `CODEWORD` values that corresponds to each index value when the `CodewordOutputPort` property is `true`.

`[..., QERR] = step(sqenc, ...)` outputs the quantization error `QERR` for each input value when the `QuantizationErrorOutputPort` property is `true`.

`[...,CLIPSTATUS] = step(sqenc, ...)` also returns output `CLIPSTATUS` as the clipping status output port for each input value when the `Partitioning` property is `Bounded` and the `ClippingStatusOutputPort` property is `true`. If an input value is outside the range defined by the `BoundaryPoints` property, `CLIPSTATUS` is `true`. If an input value is inside the range, `CLIPSTATUS` is `false`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.SignalSink System object

Package: dsp

Log simulation data in buffer

Description

The `SignalSink` object logs MATLAB simulation data. This object accepts any numeric data type.

To log MATLAB simulation data :

1. Define and set up your signal sink. See "Construction" on page 4-1543.
2. Call step to log the simulation data according to the properties of `dsp.SignalSink`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = dsp.SignalSink` returns a signal sink, `H`, that logs 2-D input data in the object.

`H = dsp.SignalSink('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a signal sink, `H`, with each specified property set to the specified value.

Properties

`BufferLength`

Maximum number of input frames to log

Specify the maximum number of frames to log. The object always preserves the most recent data in the buffer. When you specify a buffer length that is greater than the input length, the object pads the end of the logged data with zeros. To capture all input data without extra padding, set the `BufferLength` property to `inf`. The default is `inf`.

`Decimation`

Decimation factor

Setting this property to any positive integer d causes the signal sink to write data at every dth sample. The default is `1`.

`FrameHandlingMode`

Output dimensionality for frame-based inputs

Set the dimension of the output array for frame-based inputs as `2-D array (concatenate)` or `3-D array (separate)`. Concatenation occurs along the first dimension for `2-D array (concatenate)`. The default is `2-D array (concatenate)`.

`Buffer`

Logged Data (read only)

The signal sink writes simulation data into a buffer. Specify the maximum length of the buffer with the `BufferLength` on page 4-1543 property.

Methods

reset	Reset internal states of signal logger object
step	Store signal in buffer

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Log Input Data

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
hlog = dsp.SignalSink;
 for i=1:10
     y = sin(i);
     hlog(y);
 end
log = hlog.Buffer;
display(log);
```

```
log =

    0.8415
    0.9093
    0.1411
   -0.7568
   -0.9589
   -0.2794
    0.6570
    0.9894
    0.4121
   -0.5440
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the To Workspace block reference page. The object properties correspond to the block properties, except:

The object always generates fixed-point output for fixed-point input.

See Also

dsp.SignalSource

Introduced in R2012b

reset

System object: dsp.SignalSink
Package: dsp

Reset internal states of signal logger object

Syntax

`reset(H)`

Description

`reset(H)` sets the internal states of the `SignalSink` object `H` to their initial values.

step

System object: dsp.SignalSink
Package: dsp

Store signal in buffer

Syntax

`step(H,Y)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`step(H,Y)` buffers the signal `Y`. The buffer may be accessed at any time from the `Buffer` property of `H`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.SignalSource System object

Package: dsp

Import variable from workspace

Description

The `SignalSource` object imports a variable from the MATLAB workspace.

To import a variable from the MATLAB workspace:

1 Define and set up your signal source. See “Construction” on page 4-1549.
2 Call step to import the variable according to the properties of `dsp.SignalSource`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`src = dsp.SignalSource` returns a signal source System object, `src`, that outputs the variable one sample or frame at a time.

`src = dsp.SignalSource('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a signal source object, `src`, with each specified property set to the specified value.

`src = dsp.SignalSource(signal,spf,'`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a signal source object, `src`, with the `Signal` on page 4-1550 property set to `signal`, the `SamplesPerFrame` on page 4-1550 property set to `spf`, and other specified properties set to the specified values.

Properties

Signal

Variable or expression containing the signal

Specify the name of the workspace variable from which to import the signal, or a valid expression specifying the signal. The default is [1:10].

SamplesPerFrame

Number of samples per output frame

Specify the number of samples to buffer into each output frame. This property must be 1 when you specify a 3-D array in the Signal on page 4-1550 property. The default is 1.

SignalEndAction

Action after final signal values are generated

Specify the output after all of the specified signal samples have been generated as one of Set to zero, Hold final value, or Cyclic repetition. The default is Set to zero.

Methods

isDone	End-of-file status for signal reader object
reset	Reset internal states of signal reader object
step	Read one sample or frame of signal

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Create a Signal Source

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject() becomes step(myObject).

Create a signal source to output one sample at a time.

```
src1 = dsp.SignalSource;
src1.Signal = randn(1024, 1);
y1 = zeros(1024,1);
idx = 1;
while(~isDone(src1))
    y1(idx) = src1();
    idx = idx+1;
end
```

Create a signal source to output vectors.

```
src2 = dsp.SignalSource(randn(1024, 1), 128);
y2 = src2(); % y2 is a 128-by-1 frame of samples
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Signal From Workspace` block reference page. The object properties correspond to the block parameters, except:
The object does not have properties that correspond to the **Sample time** or **Warn when frame size does not evenly divide input length** block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.SignalSink

Introduced in R2012b

isDone

System object: dsp.SignalSource
Package: dsp

End-of-file status for signal reader object

Syntax

`isDone(src)`

Description

`isDone(src)` returns a logical value indicating whether or not the `SignalSource` object, `src`, has reached the end of the imported signal. If the `SignalEndAction` property is set to `Cyclic repetition`, this method will return `true` every time the reader reaches the end.

reset

System object: dsp.SignalSource
Package: dsp

Reset internal states of signal reader object

Syntax

`reset(src)`

Description

`reset(src)` resets the signal reader object `src`, to start reading from the beginning of the imported signal.

step

System object: dsp.SignalSource
Package: dsp

Read one sample or frame of signal

Syntax

```
Y = step(src)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`Y = step(src)` outputs one sample or frame of data, `Y`, from each column of the imported signal. The imported signal is the variable or expression you specify for the `Signal` property of the `SignalSource` System object `src`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.SineWave System object

Package: dsp

Discrete-time sinusoid

Description

The `SineWave` object generates a discrete-time sinusoid. The sine wave object generates a real–valued sinusoid or a complex exponential. A real-valued, discrete-time sinusoid is defined as:

$$y(n) = A\sin(2\pi fn + \phi)$$

where A is the amplitude, f is the frequency in hertz, and φ is the initial phase, or phase offset, in radians. A complex exponential is defined as:

$$y(n) = Ae^{j(2\pi fn+\phi)}$$

For both real and complex sinusoids, the amplitude, frequency, and phase offsets can be scalars or length-N vectors, where N is the desired number of channels in the output. When you specify at least one of these properties as a length-N vector, scalar values specified for the other properties are applied to each of the N channels.

To generate a discrete-time sinusoid:

1 Define and set up your sine wave. See “Construction” on page 4-1557.
2 Call step to generate the sinusoid according to the properties of `dsp.SineWave`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`sine = dsp.SineWave` returns a sine wave object, `sine`, that generates a real-valued sinusoid with an amplitude of 1, a frequency of 100 Hz, and a phase offset of 0. By default, the sine wave object only generates one sample.

`sine = dsp.SineWave('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a sine wave object, `sine`, with each property set to the specified value.

`sine = dsp.SineWave(AMP,FREQ,PHASE, '`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a sine wave object, `sine`, with the `Amplitude` property set to `AMP`, the `Frequency` property set to `FREQ`, the `PhaseOffset` property set to `PHASE`, and the other specified properties set to the specified values.

Properties

`Amplitude`

Amplitude of the sine wave

Specify the amplitude as a length-*N* vector containing the amplitudes of the sine waves in each of *N* output channels, or a scalar to apply to all *N* channels. The vector length must equal that specified for the `Frequency` on page 4-1557 and `PhaseOffset` on page 4-1557 properties. The default value is 1. This property is tunable when `Method` on page 4-1558 property is `Differential` or `Trigonometric function`.

`Frequency`

Frequency of the sine wave

Specify a length-*N* vector containing frequencies, in hertz, of the sine waves in each of *N* output channels, or a scalar to apply to all *N* channels. The vector length must equal that specified for the `Amplitude` on page 4-1557 and `PhaseOffset` on page 4-1557 properties. You can specify positive, zero, or negative frequencies. The default is 100. This property is nontunable.

`PhaseOffset`

Phase offset of the sine wave in radians

A length-*N* vector containing the phase offsets, in radians, of the sine waves in each of *N* output channels, or a scalar to apply to all *N*channels. The vector length must equal that specified for the `Amplitude` on page 4-1557 and `Frequency` on page 4-1557 properties. The default value is 0. This property is nontunable.

ComplexOutput

Indicates whether the sine wave is complex or real

Set to `true` to output a complex exponential. The default value is `false`.

Method

Method used to generate sinusoids

The sinusoids are generated by either the `Trigonometric function` , `Table lookup`, or `Differential` methods . The `Trigonometric function` method computes the sinusoid by sampling the continuous-time sinusoid. The `table lookup` method precomputes the unique samples of every output sinusoid at the start of the simulation, and recalls the samples from memory as needed. The `differential` method uses an incremental algorithm. This algorithm computes the output samples based on the output values computed at the previous sample time and precomputed update terms. The default value is `Trigonometric function`.

TableOptimization

Optimizes the table of sine values for speed or memory

Optimizes the table of sine values for `Speed` or `Memory`. When optimized for speed, the table contains *k* elements, and when optimized for memory, the table contains *k/4* elements, where *k* is the number of input samples in one full period of the sine wave. This property applies only when the `Method` on page 4-1558 property is `Table lookup`. The default value is `Speed`.

SampleRate

Sampling rate for the sine wave

Specify the sampling rate of the output, in hertz, as a positive numeric scalar. The default is 1000.

SamplesPerFrame

Number of samples per frame

Specify the number of consecutive samples from each sinusoid to buffer into the output frame. The default is 1.

`OutputDataType`

Output data type

Specify the output data type as `double`, `single`, or `Custom`. The default value is `double`.

Fixed-Point Properties

`CustomOutputDataType`

Output word and fraction lengths

Specify the output fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `OutputDataType` on page 4-1559 property is `Custom`. The default value is `numerictype([]1,16)`.

Methods

reset	Reset sine wave to the beginning
step	Discrete-time sine wave

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

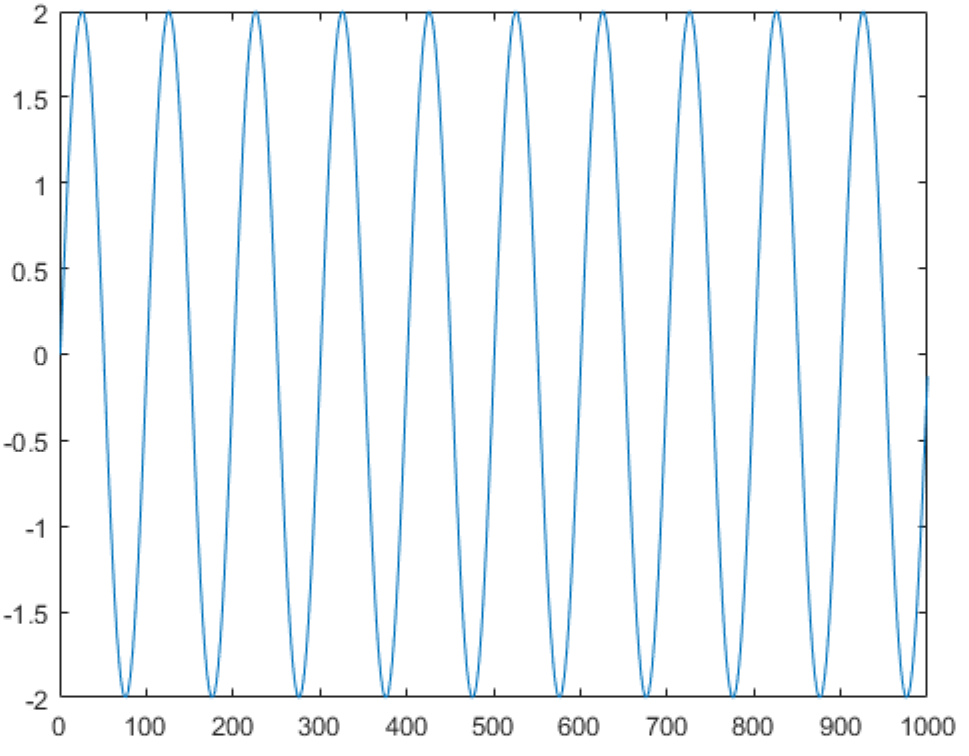
Examples

Generate a Sine Wave Signal

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject() becomes step(myObject).

Generate a sine wave with an amplitude of 2, frequency of 10 Hz, and an initial phase of 0.

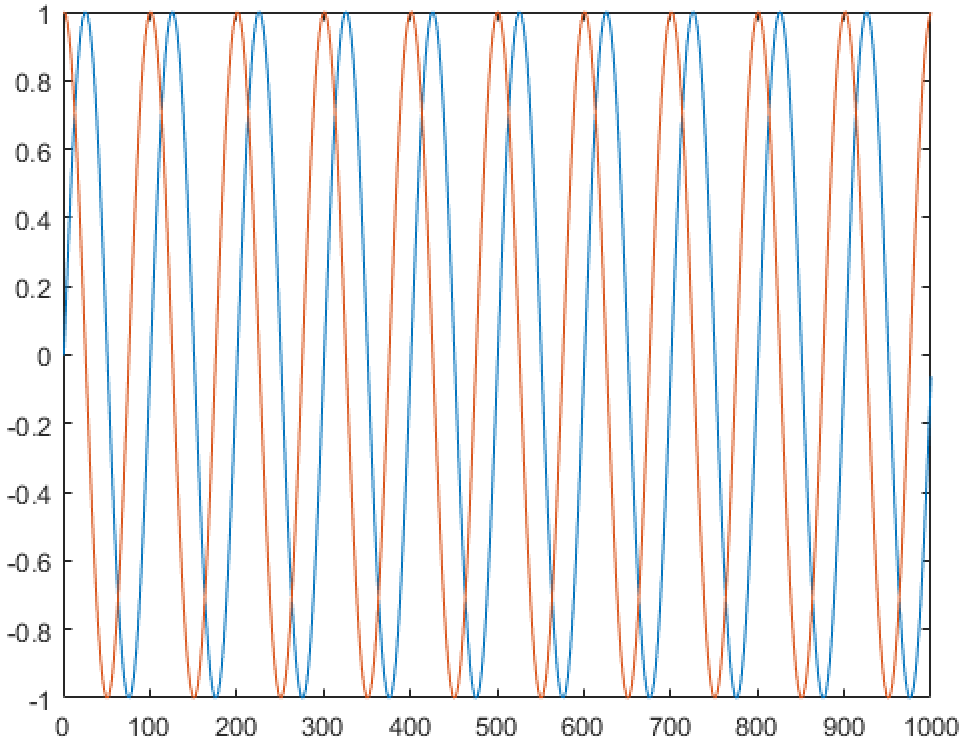
```
sine1 = dsp.SineWave(2, 10);
sine1.SamplesPerFrame = 1000;
y = sine1();
plot(y);
```



Generate two sine waves offset by a phase of pi/2 radians.

```
sine2 = dsp.SineWave;
sine2.Frequency = 10;
sine2.PhaseOffset = [0 pi/2];
```

```
sine2.SamplesPerFrame = 1000;
y = sine2();
plot(y);
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Sine Wave` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This object has no tunable properties for code generation.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.Chirp | dsp.NCO

Introduced in R2012a

reset

System object: dsp.SineWave
Package: dsp

Reset sine wave to the beginning

Syntax

`reset(sine)`

Description

`reset(sine)` resets the sine wave object to the beginning (*n=0*). If you invoke the `step` method without first invoking the `reset` method, the sine wave object generates additional samples of the original sinusoid.

Reset the SineWave Signal

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For `dsp.ColoredNoise` System object™, myObject() becomes step(myObject). For all other objects, myObject(x) becomes step(myObject,x).

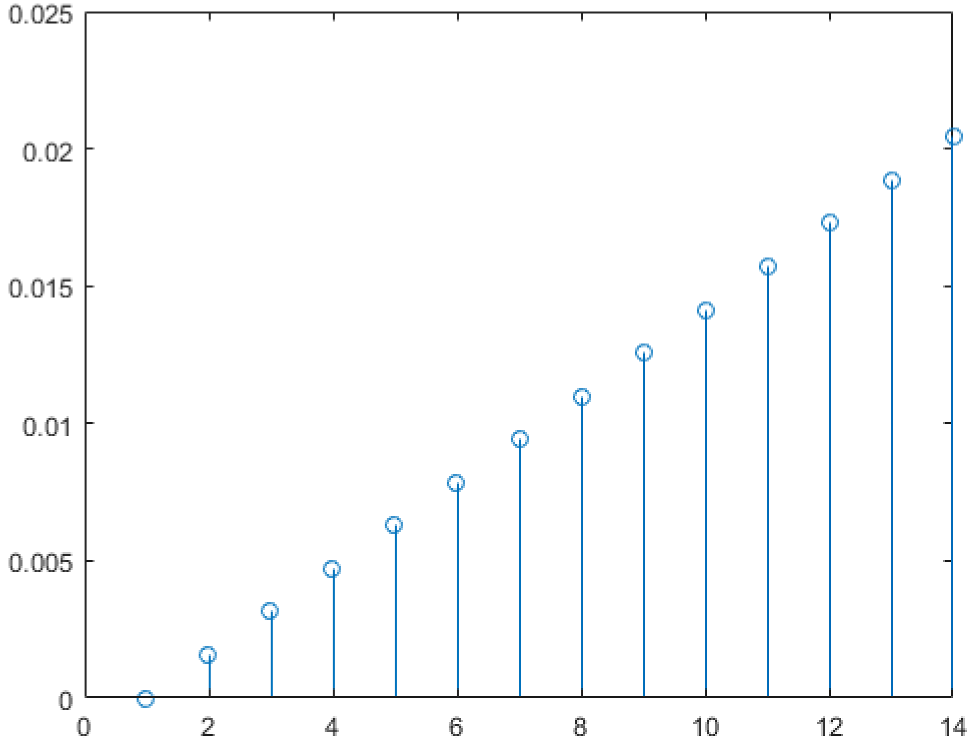
```
sine = dsp.SineWave('Amplitude',1,'Frequency',1/4,'SamplesPerFrame',7);
```

7 samples of a sine wave with frequency of 1/4 Hz

```
y = sine();
```

Call step method without reset

```
y1 = sine();
stem([y; y1])
```



y1 is a continuation of y. Now reset

```
reset(sine);
y2 = sine();
```

y2 starts at n = 0 and equals y

```
isequal(y,y2)
```

```
ans =

  logical
```

1

step

System object: dsp.SineWave
Package: dsp

Discrete-time sine wave

Syntax

`Y = step(sine)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`Y = step(sine)` produces the sine wave `Y` using the specifications in the object `sine`.

dsp.SpectrumAnalyzer System object

Package: dsp

Display frequency spectrum of time-domain signals

Description

The `SpectrumAnalyzer` object, referred to as the scope, displays the frequency spectrum of time-domain signals. This scope supports variable-size input, which allows the input frame size to change. Frame size is the first dimension of the input vector. The number of input channels must remain constant.

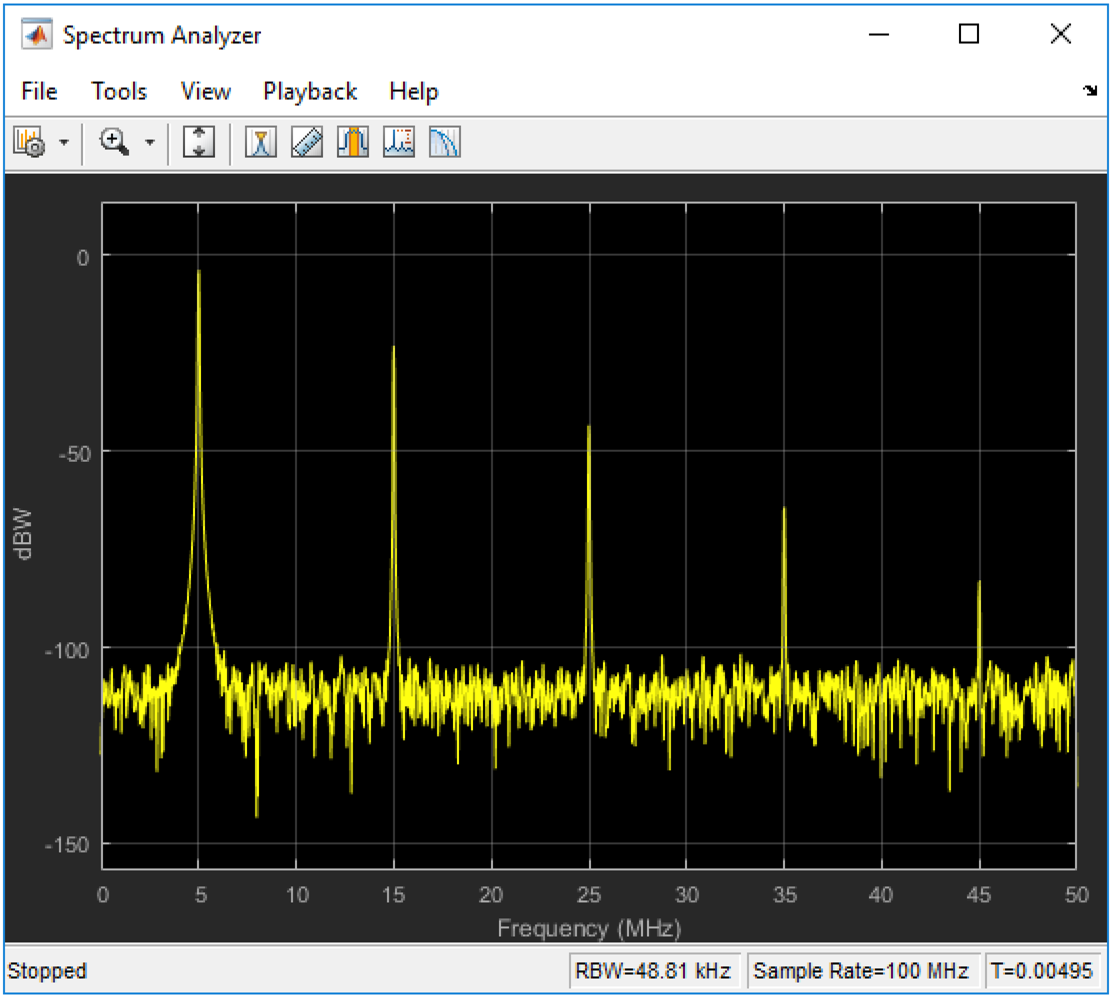
To display the spectra of signals in the Spectrum Analyzer:

1. Define and set up your Spectrum Analyzer. See "Construction" on page 4-1569.
2. Call step to display the frequency spectrum of the signals in the Spectrum Analyzer figure. The number of inputs to the step method must match the value of the NumInputPorts property. The input to every port must have same frame length, but can have a different number of channels. Each input is an individual channel of input data. For example, if the first input is a signal vector, it is Channel 1. If the second input is a two-column matrix, it is Channel 2 and Channel 3.

Use the MATLAB `clear` function to close the Spectrum Analyzer figure window and clear its associated data. Use the hide method to hide the Spectrum Analyzer window and the show method to make it visible.

Use the MATLAB `mcc` function to compile code containing a Spectrum Analyzer.

Note: You cannot open Spectrum Analyzer configuration dialogs if you have more than one compiled component in your application.



See the following sections for more information on the Spectrum Analyzer Graphical User Interface:

- "Signal Display" on page 4-1596
- "Spectrum Settings" on page 4-1601
- "Measurements Panels" on page 4-1608
- "Visuals — Spectrum Properties" on page 4-1623

- “Style Dialog Box” on page 2-1587
- “Tools — Axes Scaling Properties” on page 4-1627
- “Algorithms” on page 2-1592

Note: For information about the Spectrum Analyzer block, see `Spectrum Analyzer`.

Note: If you own the MATLAB Coder product, you can generate C or C++ code from MATLAB code in which an instance of this system object is created. When you do so, the scope system object is automatically declared as an *extrinsic* variable. In this manner, you are able to see the scope display in the same way that you would see a figure using the `plot` function, without directly generating code from it. For the full list of system objects supporting code generation, see “DSP System Toolbox” (MATLAB Coder).

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = dsp.SpectrumAnalyzer` creates a Spectrum Analyzer System object, `H`. This object displays the frequency spectrum of real- and complex-valued floating- and fixed-point signals.

`H = dsp.SpectrumAnalyzer(ports)` creates a Spectrum Analyzer with the specified number of input `ports`. This value is stored in the `NumInputPorts` property.

`H = dsp.SpectrumAnalyzer('`*`Name`*`',`*`Value`*`,...)` creates a Spectrum Analyzer System object, `H`, with each specified property *`Name`* set to the specified value. You can specify *`Name`*–*`Value`* arguments in any order.

Properties

`AxesLayout`

Orientation of the spectrum and spectrogram

Specify the layout type as one of `'Horizontal'` or `'Vertical'`. A vertical layout stacks the spectrum above the spectrogram. A horizontal layout puts the two views side-by-side. This property applies when the `ViewType` is set to `'Spectrum and spectrogram'`.

This property is Tunable (Simulink).

Default: `'Vertical'`

CenterFrequency

Frequency over which frequency span is centered

Specify as a real scalar the center frequency, in hertz, of the frequency span over which the Spectrum Analyzer computes and plots the spectrum. This property applies when you set the FrequencySpan property to `'Span and center frequency'`. The overall frequency span, defined by the Span and CenterFrequency properties, must fall within the Nyquist frequency interval. When the PlotAsTwoSidedSpectrum property is set to `true`, the interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. When the `PlotAsTwoSidedSpectrum` property is set to `false`, the interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz.

This property is Tunable (Simulink).

Default: `0`

ChannelNames

Channel names

Specify as a cell array of character vectors the input channels names to display in the legend, the Style dialog, and in some Measurements panels. The channel names only appear in the legend when the `ShowLegend` property is `true`. The default is an empty cell array, which labels the channels as `Channel 1`, `Channel 2`, etc.

This property is Tunable (Simulink).

FFTLength

FFT length

Specify as a positive, scalar integer the length of the FFT that the Spectrum Analyzer uses to compute spectral estimates. This property applies only when you set the FrequencyResolutionMethod property to `'WindowLength'` and the FFTLengthSource property to `'Property'`. The `FFTLength` must be greater than or equal to the WindowLength. You cannot control the FFT length when the FrequencyResolutionMethod property is `'RBW'`. In that case, the FFT length is set as the window length required to achieve the specified resolution bandwidth value or 1024, whichever is larger.

This property is Tunable (Simulink).

Default: `128`

`FFTLengthSource`

Source of the FFT length value

Specify the source of the FFT length value as one of `'Auto'` or `'Property'`.

- When you set this property to `'Auto'`, the Spectrum Analyzer sets the FFT length to the window length specified in the WindowLength property or to 1024, whichever is larger.
- When you set this property to `'Property'`, specify the number of FFT points using the FFTLength property. The `FFTLength` must be greater than the `WindowLength`.

This property applies only when you set the FrequencyResolutionMethod property to `'WindowLength'`.

`FFTLengthSource` property is Tunable (Simulink).

Default: `'Auto'`

`FrequencyOffset`

Frequency offset

Specify as a real, scalar value to apply the same frequency offset, in Hertz, to all channels. To apply a specific frequency offset for each channel, specify a vector of real, scalar values. The vector length must be equal to number of input channels. The *frequency*-axis values are offset by the values specified in this property. The overall span must fall within the *Nyquist* frequency interval. When the PlotAsTwoSidedSpectrum property is `true`, the Nyquist interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. When you set the `PlotAsTwoSidedSpectrum` property to `false`, the Nyquist interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. You can control the overall span in different ways based on how you set the FrequencySpan property.

This property is Tunable (Simulink).

Default: `0`

`FrequencyResolutionMethod`

Frequency resolution method

Specify how to control the frequency resolution of the spectrum analyzer as either `'RBW'` or `'WindowLength'`. When you set this property to `'RBW'`, the RBWSource and RBW properties control the frequency resolution (in Hz) of the analyzer. When you set this property to `'WindowLength'`, the WindowLength property controls the frequency resolution.

In this case, the frequency resolution of the analyzer is defined as $\frac{NENBW * Fs}{N_{window}}$ where *NENBW* is the normalized effective noise bandwidth of the window currently specified in the Window property. You can control the number of FFT points only when the `FrequencyResolutionMethod` property is `'WindowLength'`. When the `FrequencyResolutionMethod` property is `'RBW'`, the FFT length is the window length that results from achieving the specified RBW value or 1024, whichever is larger.

This property is Tunable (Simulink).

Default: `'RBW'`

`FrequencyScale`

Frequency scale

Specify the frequency scale as either `'Linear'` or `'Log'`. When you set the `FrequencyScale` property to `'Log'`, the Spectrum Analyzer displays the frequencies on the *x*-axis on a logarithmic scale. To use the `'Log'` setting, you must also set the PlotAsTwoSidedSpectrum property to `false`. When the `PlotAsTwoSidedSpectrum` property is `true`, you must set this property to `'Linear'`.

This property is Tunable (Simulink).

Default: `'Linear'`

FrequencySpan

Frequency span mode

Specify the frequency span mode as one of `'Full'`, `'Span and center frequency'`, or `'Start and stop frequencies'`.

- When you set this property to `'Full'`, the Spectrum Analyzer computes and plots the spectrum over the entire *Nyquist* frequency interval. When the PlotAsTwoSidedSpectrum property is `true`, the Nyquist interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. If you set the `PlotAsTwoSidedSpectrum` property to `false`, the Nyquist interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz.
- When you set this property to `'Span and center frequency'`, the Spectrum Analyzer computes and plots the spectrum over the interval specified by the Span and CenterFrequency properties.
- When you set this property to `'Start and stop frequencies'`, the Spectrum Analyzer computes and plots the spectrum over the interval specified by the StartFrequency and StopFrequency properties.

This property is Tunable (Simulink).

Default: `'Full'`

Name

Caption to display on Spectrum Analyzer window

Specify as a character vector the caption to display on the scope window. This property is Tunable (Simulink).

Default: `'Spectrum Analyzer'`

`NumInputPorts`

Number of input ports

Number of input ports, specified as an integer. Each signal coming through an input port becomes a separate channel in the Spectrum Analyzer.

Default: `1`

`OverlapPercent`

Overlap percentage

Specify as a real, scalar value, the percentage overlap between the previous and current buffered data segments. The overlap creates a window segment that is used to compute a spectral estimate. The value must be greater than or equal to zero and less than 100. This property is Tunable (Simulink).

Default: `0`

`PlotAsTwoSidedSpectrum`

Two sided spectrum flag

Set this property to `true` to compute and plot two-sided spectral estimates. Set this property to `false` to compute and plot one-sided spectral estimates. If you set this property to `false`, then the input signal must be real valued. When the input signal is complex valued, you must set this property to `true`.

When this property is `false`, Spectrum Analyzer uses *power folding*. The *y*-axis values are twice the amplitude that they would be if this property were set to `true`, except at 0 and the Nyquist frequency. A one-sided PSD contains the total power of the signal in the frequency interval from DC to half of the Nyquist rate. For more information, see the `pwelch` function reference page.

Default: `true`

PlotMaxHoldTrace

Max-hold trace flag

Set this property to `true` to compute and plot the maximum-hold spectrum of each input channel. The maximum-hold spectrum at each frequency bin is computed by keeping the maximum value of all the power spectrum estimates. When you toggle this property, the Spectrum Analyzer resets its maximum-hold computations. This property applies only when you set the ViewType property to `'Spectrum'` or `'Spectrum and spectrogram'`.

This property is Tunable (Simulink).

See also: PlotMinHoldTrace and PlotNormalTrace.

Default: `false`

PlotMinHoldTrace

Min-hold trace flag

Set this property to `true` to compute and plot the minimum-hold spectrum of each input channel. The minimum-hold spectrum at each frequency bin is computed by keeping the minimum value of all the power spectrum estimates. When you toggle this property, the Spectrum Analyzer resets its minimum-hold computations. This property applies only when you set the ViewType property to `'Spectrum'` or `'Spectrum and spectrogram'`.

This property is Tunable (Simulink).

See also: PlotMaxHoldTrace andPlotNormalTrace.

Default: `false`

PlotNormalTrace

Normal trace flag

Set this property to `false` to remove the display of the normal traces. These traces display the free-running spectral estimates. Note that even when the traces are removed from the display, the Spectrum Analyzer continues its spectral computations. This property applies only when you set the ViewType property to `'Spectrum'` or `'Spectrum and spectrogram'`.

This property is Tunable (Simulink).

See also: PlotMaxHoldTrace and PlotMinHoldTrace.

Default: `true`

PlotType

Plot type for normal traces

Specify the type of plot to use for displaying normal traces as either `'Line'` or `'Stem'`. *Normal* traces are traces that display free-running spectral estimates. This property applies only when you set the ViewType property to `'Spectrum'` or `'Spectrum and spectrogram'` and the PlotNormalTrace property to `true`.

This property is Tunable (Simulink).

Default: `'Line'`

Position

Spectrum Analyzer window position in pixels

Specify, in pixels, the size and location of the scope window as a 4-element double vector of the form [left bottom width height]. You can place the scope window in a specific position on your screen by modifying the values to this property. This property is Tunable (Simulink).

Default: The default depends on your screen resolution. By default, the Spectrum Analyzer window appears in the center of your screen with a width of `800` pixels and height of `450` pixels.

RBW

Resolution bandwidth

Specify as a real, positive scalar the resolution bandwidth (RBW), in hertz. RBW controls the spectral resolution of Spectrum Analyzer. This property applies only when you set the FrequencyResolutionMethod property to `'RBW'` and the RBWSource property to `'Property'`. You must specify a value to ensure that there are at least two RBW intervals over the specified frequency span. Thus, the ratio of the overall span to RBW must be greater than two: $\frac{span}{RBW} > 2$. You can specify the overall span in different ways based on how you set the `FrequencySpan` on page 4-1573 property.

This property is Tunable (Simulink).

See also: RBWSource.

Default: `9.76`

RBWSource

Source of resolution bandwidth value

Specify the source of the resolution bandwidth (RBW) as either `'Auto'` or `'Property'`. This property is relevant only when you set the FrequencyResolutionMethod property to `'RBW'`.

- When you set this property to `'Auto'`, the Spectrum Analyzer adjusts the spectral estimation resolution to ensures that there are 1024 RBW intervals over the defined frequency span.
- When you set this property to `'Property'`, specify the resolution bandwidth directly using the RBW property.

This property is Tunable (Simulink).

See also: RBW.

Default: `'Auto'`

ReducePlotRate

Reduce plot rate to improve performance

When you set this property to `true`, the scope logs data for later use and updates the display at fixed intervals of time. Data occurring between these fixed intervals might not be plotted. When you set this property to `false`, the scope updates every time it computes the power spectrum. Use the `false` setting when you do not want to miss any spectral updates at the expense of slower simulation speed. The simulation speed is faster when this property is set to `true`. This property is Tunable (Simulink).

Default: `true`

`ReferenceLoad`

Reference load

Specify as a real, positive scalar the load, in ohms, that the Spectrum Analyzer uses as a reference to compute power values.

This property is Tunable (Simulink).

Default: `1`

`SampleRate`

Sample rate of input

Specify the sample rate, in hertz, of the input signals.

The sample rate must be a finite numeric scalar.

Default: `10e3`

`ShowGrid`

Option to enable or disable grid display

When you set this property to `true`, the grid appears. When you set this property to `false`, the grid is hidden. This property is Tunable (Simulink).

Default: `false`

`ShowLegend`

Show or hide legend

When you set this property to `true`, the scope displays a legend with the input channels labels specified in the `ChannelNames` on page 4-1570 property. When you set this property to `false`, the scope does not display a legend.

This property applies only when you set the `ViewType` property to `'Spectrum'` or `'Spectrum and spectrogram'`. This property is Tunable (Simulink).

Default: `false`

SidelobeAttenuation

Sidelobe attenuation of window

Specify as a real, positive scalar the window sidelobe attenuation, in decibels (dB). This property applies only when you set the Window property to `'Chebyshev'` or `'Kaiser'`. The value must be greater than or equal to `45`.

This property is Tunable (Simulink).

Default: `60`

Span

Frequency span over which spectrum is computed and plotted

Specify as a real, positive scalar the frequency span, in hertz, over which the Spectrum Analyzer computes and plots the spectrum. This property applies only when you set the FrequencySpan property to `'Span and center frequency'`. The overall span, defined by this property and the CenterFrequency property, must fall within the *Nyquist* frequency interval. When the PlotAsTwoSidedSpectrum property is `true`, the Nyquist interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. When you set the `PlotAsTwoSidedSpectrum` property to `false`, the Nyquist interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz.

This property is Tunable (Simulink).

Default: `10e3`

SpectralAverages

Number of spectral averages

Specify as a positive, scalar integer the number of spectral averages. This property applies only when you set the ViewType property to `'Spectrum'`. The Spectrum Analyzer computes the current power spectrum estimate by computing a running average of the last *N* power spectrum estimates. This property defines the number of spectral averages, *N*.

This property is Tunable (Simulink).

Default: 1

SpectralMask

Spectral mask lines for a spectrum plot.

Specify whether to display upper and lower spectral masks lines on a spectrum plot. This property uses SpectralMaskSpecfication properties to enable and configure the spectral masks. The SpectralMaskSpecfication properties are:

- EnabledMasks — Masks to enable, specified as a character vector. Valid values are 'None', 'Upper', 'Lower', or 'Upper and lower'.

  Default: 'None'
- UpperMask — Upper limit spectral mask, specified as a scalar or two-column matrix. If UpperMask is a scalar, the upper limit mask uses the power value of the scalar for all frequency values applicable to the Spectrum Analyzer. If UpperMask is a matrix, the first column contains the frequency values (Hz), which correspond to the *x*-axis values. The second column contains the power values, which correspond to the associated *y*-axis values. To apply offsets to the power and frequency values, use ReferenceLevel and MaskFrequencyOffset property values, respectively.

  Default: Inf
- LowerMask — Lower limit spectral mask, specified as a scalar or two-column matrix. If LowerMask is a scalar, the lower limit mask uses the power value of the scalar for all frequency values applicable to the Spectrum Analyzer. If LowerMask is a matrix, the first column contains the frequency values (Hz), which correspond to the *x*-axis values. The second column contains the power values, which correspond to the associated *y*-axis values. To apply offsets to the power and frequency values, use ReferenceLevel and MaskFrequencyOffset property values, respectively.

  Default: Inf
- ReferenceLevel — Reference level for mask power values, specified as either 'Custom' or 'Spectrum peak'. When ReferenceLevel is 'Custom', the CustomReferenceLevel property value is used as the reference to the power values, in dBr, in the UpperMask and LowerMask properties. When ReferenceLevel is 'Spectrum peak', the peak value of the current spectrum of the SelectedChannel is used.

  Default: 'Custom'

- `CustomReferenceLevel` — Custom reference level, specified as a real value, in the same units as the power units. The reference level is the value to which to reference the power values in the `UpperMask` and `LowerMask` properties. This property applies when `ReferenceLevel` is set to `'Custom'`. This property uses the same units as the `PowerUnits` property of the Spectrum Analyzer.

  Default: `0`
- `SelectedChannel` — Input channel with peak spectrum to use as the mask reference level, specified as an integer. This property applies when `ReferenceLevel` is set to `'Spectrum peak'`.

  Default: `1`
- `MaskFrequencyOffset` — Frequency offset, specified as a finite, numeric scalar. Frequency offset is the amount of offset to apply to frequency values in the `UpperMask` and `LowerMask` properties.

  Default: `0`

All `SpectralMaskSpecfication` properties are Tunable (Simulink).

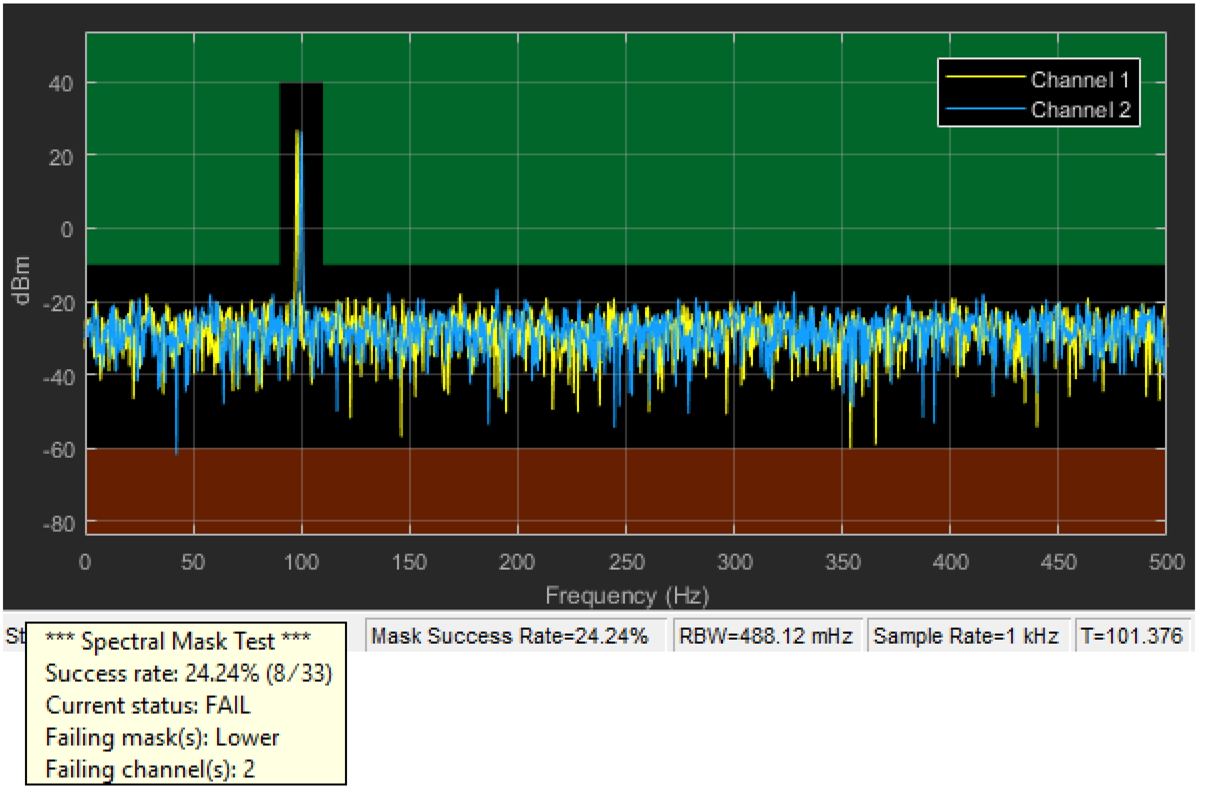
You can check the status of the spectral mask using a method or an event listener.

- To get the status of the spectral masks, call getSpectralMaskStatus.
- To perform an action every time the mask fails, use the `MaskTestFailed` event. To trigger a function when the mask fails, create a listener to the `MaskTestFailed` event and define a callback function to trigger. For more details about using events, see “Events” (MATLAB).

This example shows how to add a spectral mask to an existing `dsp.SpectrumAnalyzer` System object scope and get the status with getSpectralMaskStatus.

```
sine = dsp.SineWave('Frequency',[98 100],'SampleRate',1000);
sine.SamplesPerFrame = 1024;
scope = dsp.SpectrumAnalyzer('SampleRate',sine.SampleRate, ...
       'PlotAsTwoSidedSpectrum',false,'ShowLegend',true);
for i=1:100
    scope(sine() + 0.05*randn(1024,2));
end

scope.SpectralMask.EnabledMasks = 'Upper and lower';
upperMask = [0 -10; 90 -10; 90 40; 110 40; 110 -10; 500 -10];
```

```
set(scope.SpectralMask,'UpperMask',upperMask,'LowerMask',-60);

res = getSpectralMaskStatus(scope)
release(scope);

res =
  struct with fields:

    IsCurrentlyPassing: 0
        NumPassedTests: 8
         NumTotalTests: 33
           SuccessRate: 24.2424
          FailingMasks: 'Lower'
       FailingChannels: 2
        SimulationTime: 101.3760
```



Masks are overlaid on the spectrum. If the mask is green, the signal is passing. If the mask is red, the signal is failing.

At the bottom of the window, the **Mask success rate** shows what percentage of the time the mask is succeeding. A tooltip gives you details about which mask is failing, how many times the mask failed, and which channels are causing the failure.

SpectrumType

Type of spectrum to show

Specify the spectrum type as one of `'Power'`, `'Power density'`, or `'RMS'`.

- `'Power'`, the Spectrum Analyzer shows the power spectrum.
- `'Power density'`, the Spectrum Analyzer shows the power spectral density. The power spectral density is the magnitude squared of the spectrum normalized to a bandwidth of 1 hertz.
- `'RMS'`, the Spectrum Analyzer shows the root mean squared. The root mean squared shows the square root of the sum of the squared means. This option is useful when viewing the frequency of voltage or current signals.

This property is Tunable (Simulink).

Default: `'Power'`

SpectrumUnits

Units of the spectrum

Specify the units in which the Spectrum Analyzer displays power values as either `'dBm'`, `'dBW'`, `'Watts'`, `'Vrms'`, or `'dBV'`.

The available spectrum units depends on the value of `SpectrumType`.

SpectrumType	**Allowed SpectrumUnits**
`Power` or `Power density`	`'dBm'`, `'dBW'`, `'Watts'`
`RMS`	`'Vrms'`, `'dBV'`

This property is Tunable (Simulink).

Default: `'dBm'`

`StartFrequency`

Start frequency over which spectrum is computed

Specify as a real scalar the start frequency, in hertz, over which the Spectrum Analyzer computes and plots the spectrum. This property applies only when you set the FrequencySpan property to `'Start and stop frequencies'`. The overall span, which is defined by StopFrequency and this property, must fall within the *Nyquist* frequency interval. When the PlotAsTwoSidedSpectrum property is `true`, the Nyquist interval is $\left[-\frac{SampleRate}{2},\frac{SampleRate}{2}\right]+FrequencyOffset$ hertz. If you set the `PlotAsTwoSidedSpectrum` property to `false`, the Nyquist interval is $\left[0,\frac{SampleRate}{2}\right]+FrequencyOffset$ hertz.

This property is Tunable (Simulink).

Default: `-5e3`

`StopFrequency`

Stop frequency over which spectrum is computed

Specify as a real scalar the stop frequency, in hertz, over which the Spectrum Analyzer computes and plots the spectrum. This property applies only when you set the FrequencySpan property to `'Start and stop frequencies'`. The overall span, defined by this property and the StartFrequency property, must fall within the *Nyquist* frequency interval. When the PlotAsTwoSidedSpectrum property is set to `true`, the interval is $\left[-\frac{SampleRate}{2},\frac{SampleRate}{2}\right]+FrequencyOffset$ hertz. When the `PlotAsTwoSidedSpectrum` property is set to `false`, the interval is $\left[0,\frac{SampleRate}{2}\right]+FrequencyOffset$ hertz.

This property is Tunable (Simulink).

Default: `5e3`

TimeResolution

Time resolution

Specify the time resolution of each spectrogram line as a positive scalar, expressed in seconds. This property applies when you set the ViewType property to `'Spectrogram'` or `'Spectrum and spectrogram'`, and the TimeResolutionSource property to `'Property'`.

The time resolution value is determined based on frequency resolution method, the RBW setting, and the time resolution setting.

Frequency Resolution	RBW Setting	Time Resolution Setting	Time Resolution
RBW (Hz)	`Auto`	`Auto`	1/RBW s
RBW (Hz)	`Auto`	Manually entered	Time Resolution s
RBW (Hz)	Manually entered	`Auto`	1/RBW s
RBW (Hz)	Manually entered	Manually entered	Must be equal to or greater than the minimum attainable time resolution, 1/RBW s. Several spectral estimates are combined into one spectrogram line to obtain the desired time resolution. Interpolation is used to obtain time resolution values that are not integer multiples of 1/RBW s.
Window length	—	`Auto`	1/RBW s RBW = *(NENBW**Fs)/Window Length, where *NENBW* is the normalized effective noise bandwidth of the specified window.
Window length	—	Manually entered	Must be equal to or greater than the minimum attainable

Frequency Resolution	RBW Setting	Time Resolution Setting	Time Resolution
			time resolution, (*NENBW**Fs)/ Window Length. Several spectral estimates are combined into one spectrogram line to obtain the desired time resolution. Interpolation is used to obtain time resolution values that are not integer multiples of 1/RBW s.

This property is Tunable (Simulink).

Default: `0.001`

`TimeResolutionSource`

Source of the time resolution value.

Specify the source for the time resolution of each spectrogram line as either `'Auto'` or `'Property'`. This property applies when you set the ViewType property to `'Spectrogram'` or `'Spectrum and spectrogram'`. See the time resolution table at TimeResolution.

This property is Tunable (Simulink).

Default: `'Auto'`

`TimeSpan`

Time span

Specify the time span of the spectrogram display as a positive scalar in seconds. This property applies when you set the ViewType property to `'Spectrogram'` or `'Spectrum and spectrogram'` and the TimeSpanSource property to `'Property'`. You must set the time span to be at least twice as large as the duration of the number of samples required for a spectral update.

This property is Tunable (Simulink).

Default: `0.1`

`TimeSpanSource`

Source of time span value

Specify the source for the time span of the spectrogram as either `'Auto'` or `'Property'`. This property applies when you set the ViewType property to `'Spectrogram'` or `'Spectrum and spectrogram'`. If you set this property to `'Auto'`, the spectrogram displays 100 spectrogram lines at any given time. If you set this property to `'Property'`, the spectrogram uses the time duration you specify in seconds in the TimeSpan property.

This property is Tunable (Simulink).

Default: `'Auto'`

`Title`

Display title

Specify the display title as a character vector. Enter `%<SignalLabel>` to use the signal labels in the Simulink Model as the axes titles. This property is Tunable (Simulink).

Default: `''`

`ViewType`

Viewer type

Specify the spectrum type as one of `'Spectrum'`, `'Spectrogram'`, or `'Spectrum and spectrogram'`.

- `'Spectrum'` — the Spectrum Analyzer shows the spectrum.
- `'Spectrogram'` — the Spectrum Analyzer opens a spectrogram view, which shows frequency content over time. Each line of the spectrogram is one periodogram. Time scrolls from the bottom to the top of the display. The most recent spectrogram update is at the bottom of the display.
- `'Spectrum and Spectrogram'` — the Spectrum Analyzer shows a dual view of a spectrum and spectrogram.

This property is Tunable (Simulink).

Default: `'Spectrum'`

Window

Window function

Specify a window function for the spectral estimator as one of the options in the following table. For information on any window function, follow the link to the corresponding function reference in the Signal Processing Toolbox documentation.

Window Option	Corresponding Function in Signal Processing Toolbox
'Rectangular'	rectwin
'Chebyshev'	chebwin
'Flat Top'	flattopwin
'Hamming'	hamming
'Hann'	hann
'Kaiser'	kaiser

This property is Tunable (Simulink).

Default: 'Hann'

WindowLength

The window length

Control the frequency resolution by specifying the window length, in samples used to compute the spectral estimates. The window length must be an integer scalar greater than 2. This property applies only when you set the FrequencyResolutionMethod property to 'WindowLength', which controls the frequency resolution based on your window length setting. The resulting resolution bandwidth (RBW) is $\frac{NENBW * Fs}{N_{window}}$

where N_{window} is the WindowLength value.

This property is Tunable (Simulink).

See also: Window.

Default: 1024

`YLabel`

The label for the *y*-axis

Specify the text for the scope to display to the left of the *y*-axis. Tunable (Simulink)

This property applies only when you set the ViewType property to `'Spectrum'` or `'Spectrum and spectrogram'`. Regardless of this property, Spectrum Analyzer always displays power units as one of `'dBm'`, `'dBW'`, `'Watts'`, `'dBm/Hz'`, `'dBW/Hz'`, `'Watts/Hz'`, `'Vrms'`, `'dBV'`.

See also: SpectrumUnits.

Default: `''`

`YLimits`

The limits for the *y*-axis

Specify the *y*-axis limits as a 2-element numeric vector, `[ymin ymax]`. This property is Tunable (Simulink).

This property applies only when you set the ViewType property to `'Spectrum'` or `'Spectrum and spectrogram'`. The units directly depend upon the SpectrumUnits property.

Default: `[-80, 20]`

Methods

getSpectralMaskStatus	Get test results of current spectral mask
hide	Hide Spectrum Analyzer window
reset	Reset internal states of Spectrum Analyzer object
show	Make Spectrum Analyzer window visible
step	Update spectrum in Spectrum Analyzer figure

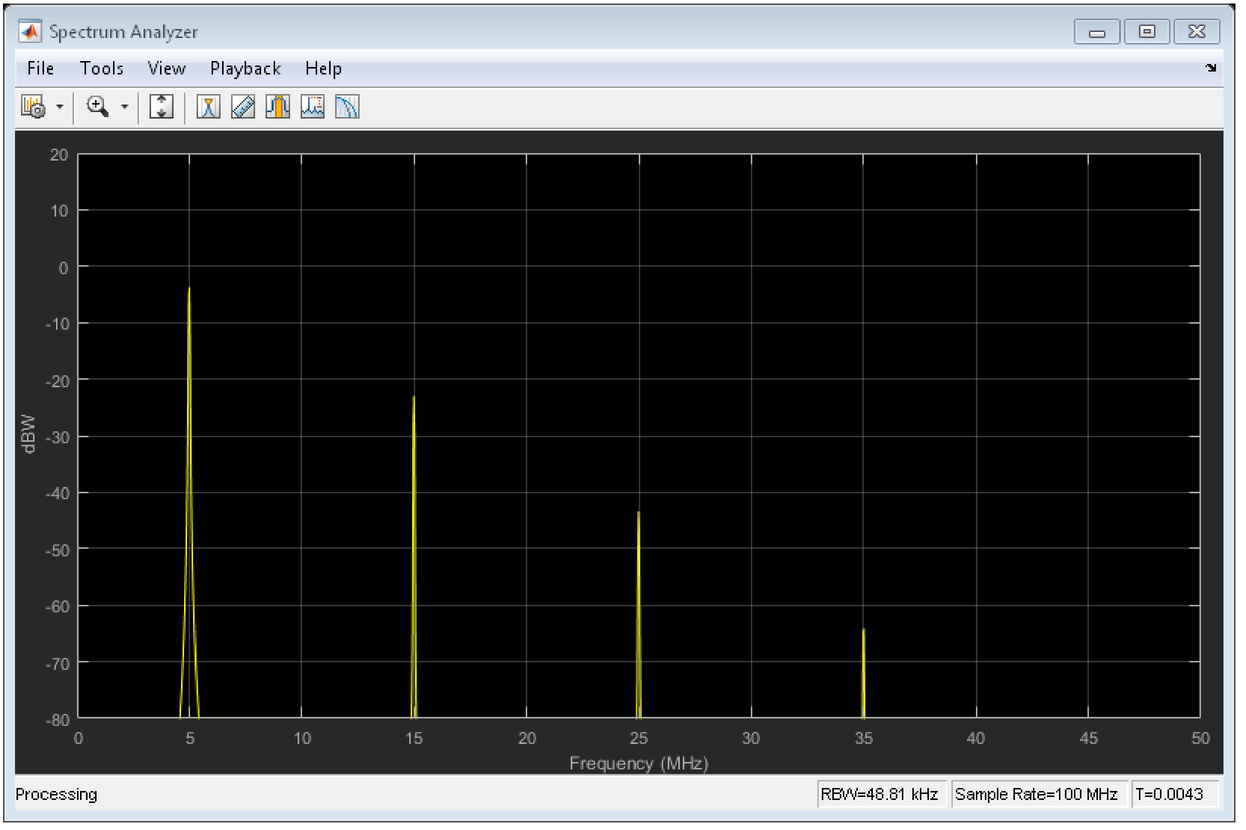
Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Spectrum Analyzer for One-Sided Power Spectrum

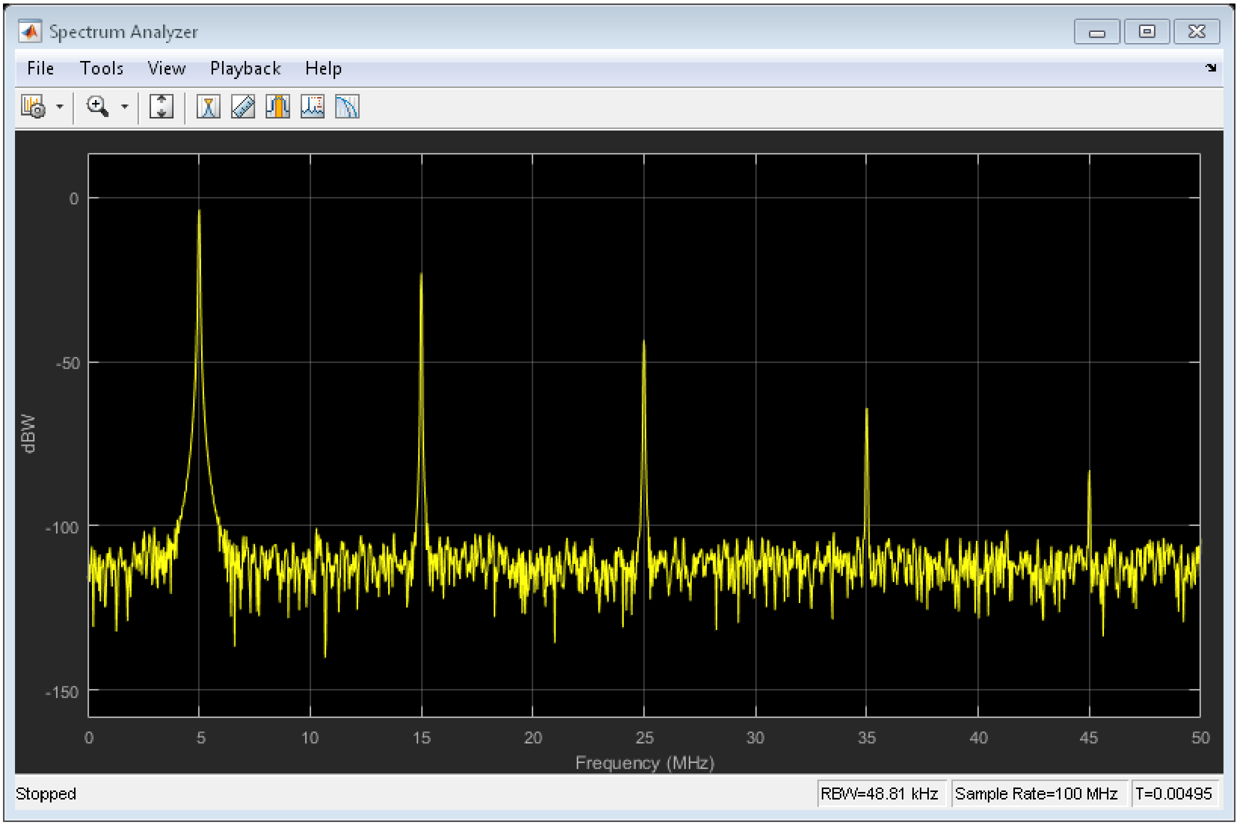
View a one-sided power spectrum made from the sum of fixe real sine waves with different amplitudes and frequencies.

```
Fs = 100e6;  % Sampling frequency
fSz = 5000;  % Frame size

sin1 = dsp.SineWave(1e0,  5e6,0,'SamplesPerFrame',fSz,'SampleRate',Fs);
sin2 = dsp.SineWave(1e-1,15e6,0,'SamplesPerFrame',fSz,'SampleRate',Fs);
sin3 = dsp.SineWave(1e-2,25e6,0,'SamplesPerFrame',fSz,'SampleRate',Fs);
sin4 = dsp.SineWave(1e-3,35e6,0,'SamplesPerFrame',fSz,'SampleRate',Fs);
sin5 = dsp.SineWave(1e-4,45e6,0,'SamplesPerFrame',fSz,'SampleRate',Fs);

scope = dsp.SpectrumAnalyzer;
scope.SampleRate = Fs;
scope.SpectralAverages = 1;
scope.PlotAsTwoSidedSpectrum = false;
scope.RBWSource = 'Auto';
scope.PowerUnits = 'dBW';
for idx = 1:1e2
     y1 = sin1();
     y2 = sin2();
     y3 = sin3();
     y4 = sin4();
     y5 = sin5();
     scope(y1+y2+y3+y4+y5+0.0001*randn(fSz,1));
end
```



Run the `release` method to let property values and input characteristics change. The scope automatically scales the axes.

```
release(scope)
```



Run the `clear` function to close the Spectrum Analyzer window.

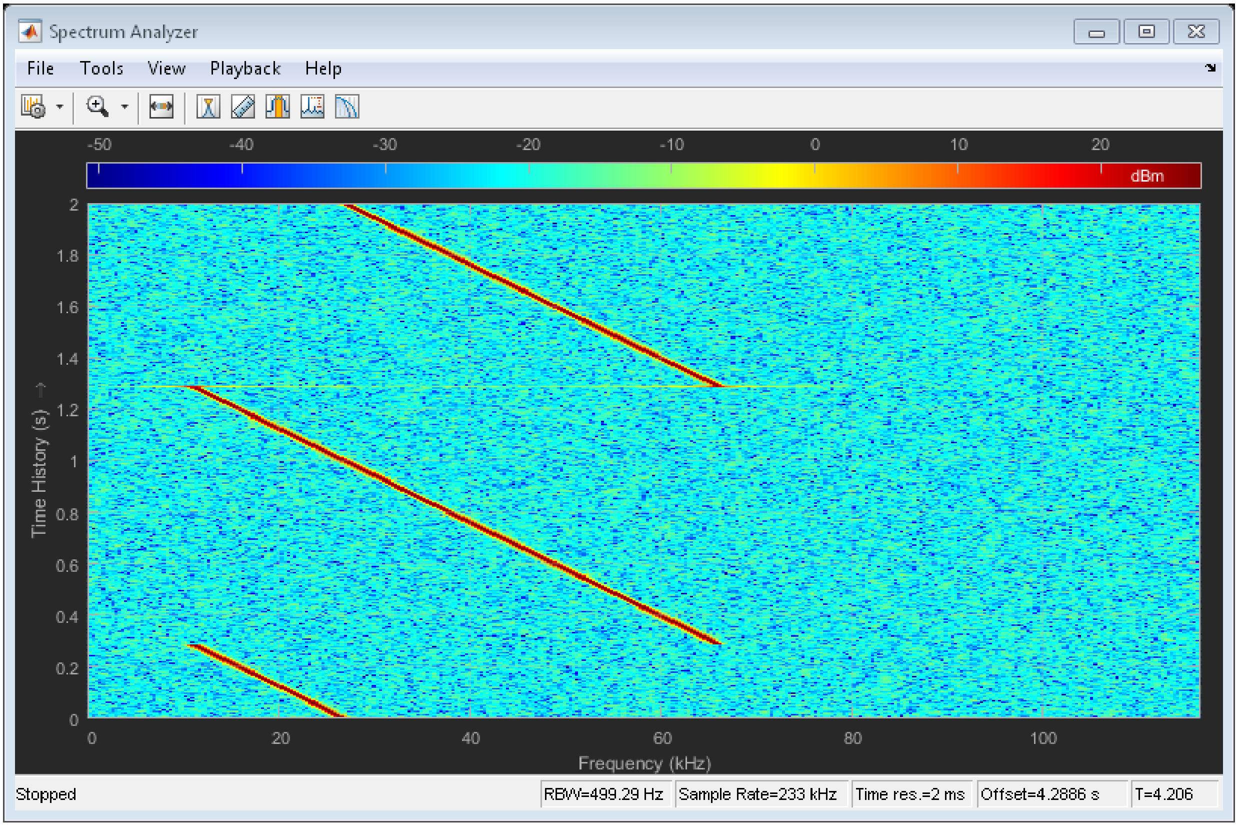
```
clear('scope');
```

Spectrogram of Chirp Signal

This example shows the spectrogram for a chirp signal with added random noise.

```
Fs = 233e3;
frameSize = 20e3;
chirp = dsp.Chirp('SampleRate',Fs,...
  'SamplesPerFrame',frameSize,...
  'InitialFrequency',11e3,...
  'TargetFrequency',11e3+55e3);
```

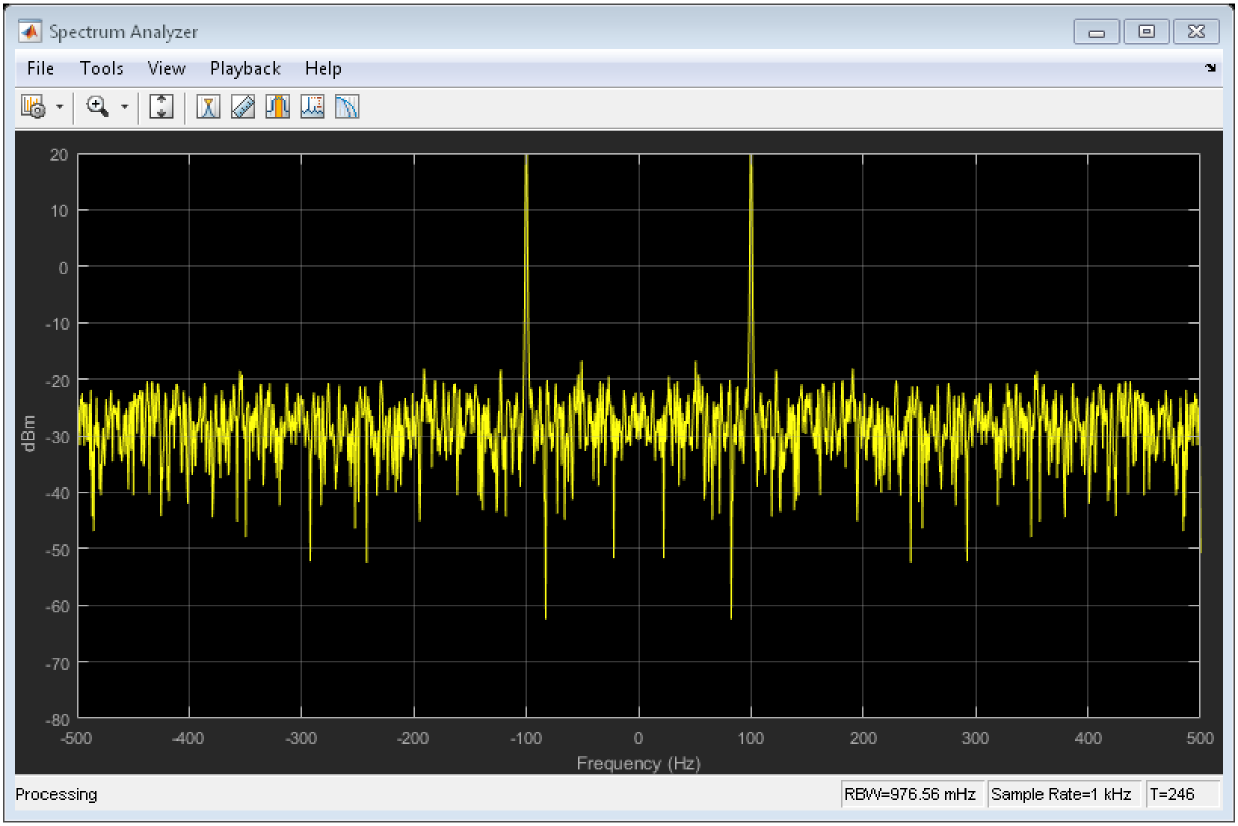
```
scope = dsp.SpectrumAnalyzer('SampleRate',Fs);
scope.SpectrumType = 'Spectrogram';
scope.RBWSource = 'Property';
scope.RBW = 500;
scope.TimeSpanSource = 'Property';
scope.TimeSpan = 2;
scope.PlotAsTwoSidedSpectrum = false;

for idx = 1:50
  y = chirp()+ 0.05*randn(frameSize,1);
  scope(y);
end
release(scope)
```



Spectrum Analyzer For Two-Sided Power Spectrum

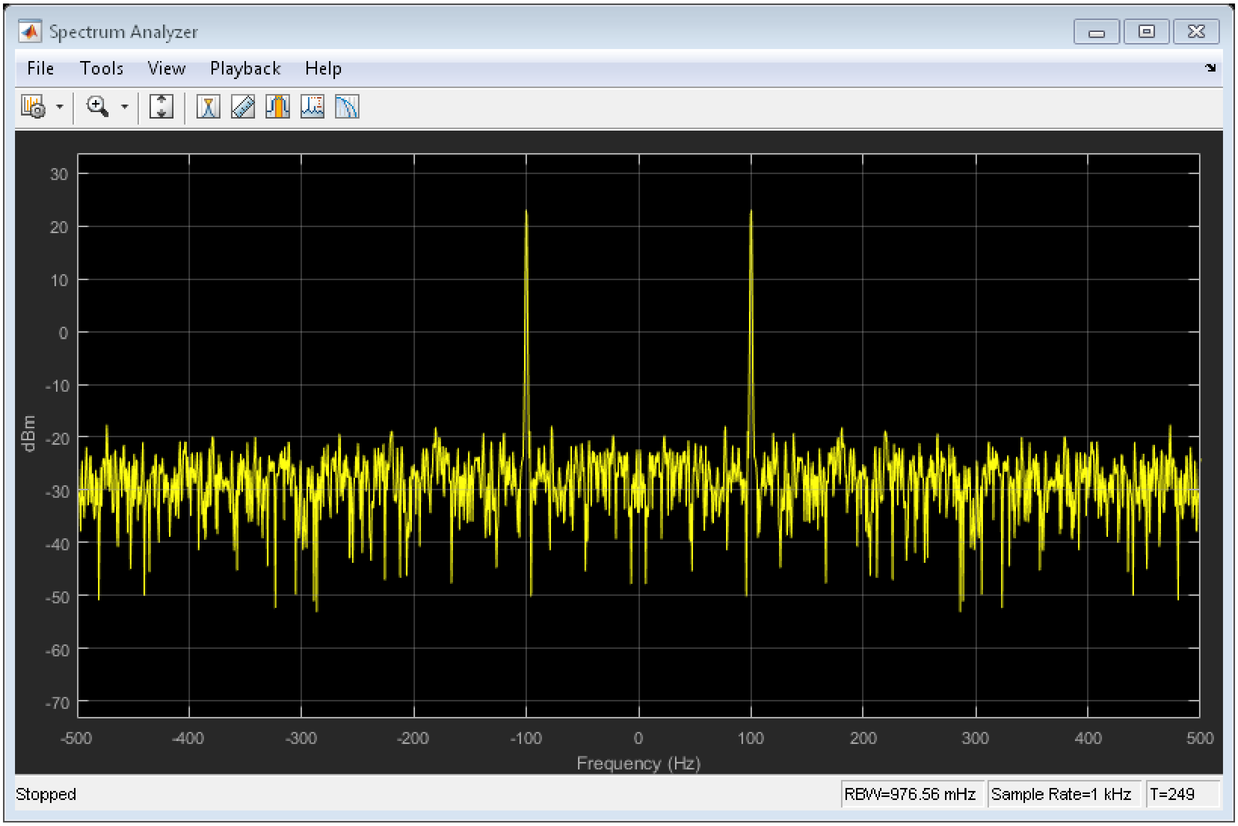
View a two-sided power spectrum of a sine wave with noise on the Spectrum Analyzer.

```
sin = dsp.SineWave('Frequency',100,'SampleRate',1000);
sin.SamplesPerFrame = 1000;
scope = dsp.SpectrumAnalyzer('SampleRate',sin.SampleRate);
for ii = 1:250
  x = sin() + 0.05*randn(1000,1);
  scope(x);
end
```



Run the `release` method to change property values and input characteristics. The scope automatically scales the axes. It updates the display one more time if any data is in the internal buffer.

```
release(scope);
```

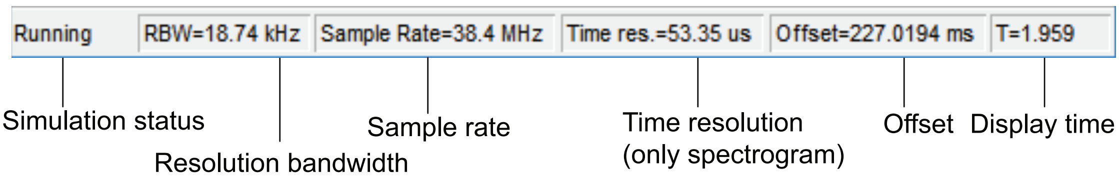


Run the MATLAB `clear` function to close the Spectrum Analyzer window.

```
clear('scope');
```

Signal Display

The Spectrum Analyzer indicates the spectrum computation settings that are represented in the current display. Check the *Resolution Bandwidth*, *Time Resolution*, and *Offset* indicators in the scope status bar for this information. The values specified by these indicators may be changed by modifying parameters in the **Spectrum Settings** panel. You can also view the object state and the amount of time data that correspond to the current display. Check the *Simulation status* and *Display time* indicators for this information. The following figure highlights these aspects of the Spectrum Analyzer window.



- *Resolution Bandwidth* — The smallest positive frequency or frequency interval that can be resolved.

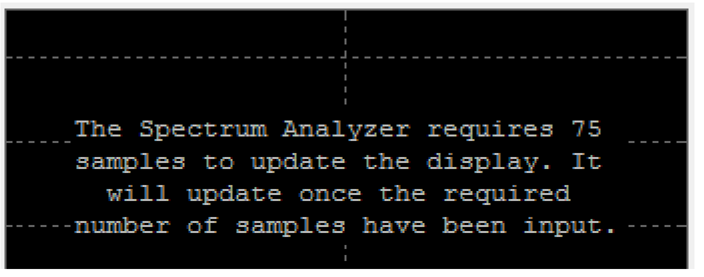
  Details

  Spectrum Analyzer sets the resolution bandwidth based on the `FrequencyResolutionMethod` property setting on the **Main options** pane of the **Spectrum Settings** panel. If `FrequencyResolutionMethod` is `RBW (Hz)` then the specified value of RBW is used. You can also get or set this value from the RBW property when RBWSource is set to `'Property'`. By default, the `RBW (Hz)` parameter on the **Main options** pane and the related `RBWSource` property are set to `'Auto'`. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified **Frequency Span**.

  You can set the resolution bandwidth to whatever value you choose. For this reason, there is a minimum boundary on the number of input samples required to compute a spectral update. This number of input samples required to compute one spectral update is shown as **Samples/update** in the **Main options** pane. This value is directly related to *RBW* by the following equation:

  $$N_{samples} = \frac{\left(1 - \frac{O_p}{100}\right) \times NENBW \times F_s}{RBW}$$

  . Overlap percentage, O_p, is the value of the **Overlap %** parameter in the **Window Options** pane of the **Spectrum Settings** panel. *NENBW* is the normalized effective noise bandwidth, a factor of the windowing method used, which is shown in the **Window Options** pane. F_s is the sample rate. In some cases, the number of samples provided in the input are not sufficient to achieve the resolution bandwidth that you specify. When this situation occurs, Spectrum Analyzer shows a warning message on the display.



Spectrum Analyzer removes this message and displays a spectral estimate as soon as enough data has been input.

If the `FrequencyResolutionMethod` property setting on the **Main options** pane of the **Spectrum Settings** is `Window length`, you specify the window length and the resulting RBW is $\frac{NENBW * Fs}{N_{window}}$. The **Samples/update** in this case is directly related to *RBW* by the following equation: $N_{samples} = \left(1 - \frac{O_p}{100}\right) N_{window}$

- *Time Resolution* — The time resolution for a spectrogram line.

  Details

  Time resolution is the amount of data, in seconds, used to compute a spectrogram line. The **Time Resolution** parameter is available only when the spectrum **Type** is `Spectrogram`. The *minimum attainable resolution* is the amount of data time required to compute a single spectral estimate. When the SpectrumType property is set to `'Spectrogram'`, you can get or set the minimum attainable resolution value from the TimeResolution property. See the time resolution table in the `TimeResolution` property description.

- *Offset* — The constant frequency offset to apply to the entire spectrum or a vector of frequency offsets to apply to each spectrum for multiple inputs.

Details

Spectrum Analyzer adds this constant offset or the vector of offsets to the values on the *frequency*-axis using the value of **Offset** on the **Trace options** pane of the **Spectrum Settings** panel. You can also set the offset from the FrequencyOffset property. The offset is the current time value at the middle of the interval of the line displayed at 0 seconds. The actual time of a particular spectrogram line is the offset minus the *y*-axis time listing. The offset is displayed on the plot only when the spectrum **Type** is `Spectrogram`.

- *Simulation Status* — Provides the current status of the model simulation.

Details

The status can be one of the following conditions:

- `Processing` — Occurs after you construct the `SpectrumAnalyzer` object and before you run the `release` method.
- `Stopped` — Occurs after you run the `release` method.

The *Simulation Status* is part of the *Status Bar* in the Spectrum Analyzer window. You can choose to hide or display the entire *Status Bar*. From the Spectrum Analyzer menu, select **View > Status Bar**.

- *Display time* — The amount of time that has progressed since the last update to the Spectrum Analyzer display.

Details

Every time you call the step method, the simulation time increases by the number of rows in the input signal divided by the sample rate, as given by the following formula: $t_{sim} = t_{sim} + \frac{\text{length(0:length(xsine))-1}}{\text{SampleRate}}$. At the beginning of a simulation, you can modify the **SampleRate** parameter on the **Main options** pane of the **Spectrum Settings** panel. You can also set the sample rate using the SampleRate property. The display time is updated each time the display is updated. When ReducePlotRate is `true`, the simulation time and display time might differ. If at the end of a for loop that includes the Spectrum Analyzer, the times differ, you can call the `release` method to update the display with any data left in the buffer. Note, however, that if the remaining data is not a complete window interval, the display is not updated.

The *Display time* indicator is a component of the *Status Bar* in the Spectrum Analyzer window. You can choose to hide or display the entire *Status Bar*. From the Spectrum Analyzer menu, select **View** > **Status Bar** .

- *Frequency span* — The range of values shown on the *frequency*-axis on the Spectrum Analyzer window.

Details

Spectrum Analyzer sets the frequency span using the values of parameters on the **Main options** pane of the **Spectrum Settings** panel.

- `Span(Hz)` and `CF(Hz)` visible — The *Frequency span* value equals the **Span** parameter in the **Main options** pane. You can also get or set this value from the Span property when the FrequencySpan property is set to `'Span and Center Frequency'`.
- `FStart(Hz)` and `FStop(Hz)` — The *Frequency span* value equals the difference of the **FStop** and **FStart** parameters in the **Main options** pane, as given by the formula: $f_{span} = f_{stop} - f_{start}$. You can also get or set these values from the StartFrequency and StopFrequency properties when the `FrequencySpan` property is set to `'Start and stop frequencies'`.

By default, the **Full Span** check box in the **Main options** pane is enabled, and its equivalent FrequencySpan property is set to `'Full'`. In this case, the Spectrum Analyzer computes and plots the spectrum over the entire *Nyquist* frequency interval. When the **Two-sided spectrum** check box in the **Trace options** pane is enabled, and its equivalent PlotAsTwoSidedSpectrum property is `true`, the Nyquist interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. If you set the `PlotAsTwoSidedSpectrum` property to `false`, the Nyquist interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz.

For more information, see “Spectrum Settings” on page 4-1601.

Reduce Plot Rate to Improve Performance

By default, Spectrum Analyzer updates the display at fixed intervals of time at a rate not exceeding 20 hertz. If you want Spectrum Analyzer to plot a spectrum on every simulation time step, you can disable the **Reduce Plot Rate to Improve Performance** option. In the Spectrum Analyzer menu, select **Simulation** > **Reduce Plot Rate to Improve Performance** to clear the check box. Tunable (Simulink).

Note: When this check box is selected, Spectrum Analyzer may display a misleading spectrum in some situations. For example, if the input signal is wide-band with non-stationary behavior, such as a chirp signal, Spectrum Analyzer might display a stationary spectrum. The reason for this behavior is that Spectrum Analyzer buffers the input signal data and only updates the display periodically at approximately 20 times per second. Therefore, Spectrum Analyzer does not render changes to the spectrum that occur and elapse between updates, which gives the impression of an incorrect spectrum. To ensure that spectral estimates are as accurate as possible, clear the **Reduce Plot Rate to Improve Performance** check box. When you clear this box, Spectrum Analyzer calculates spectra whenever there is enough data, rendering results correctly.

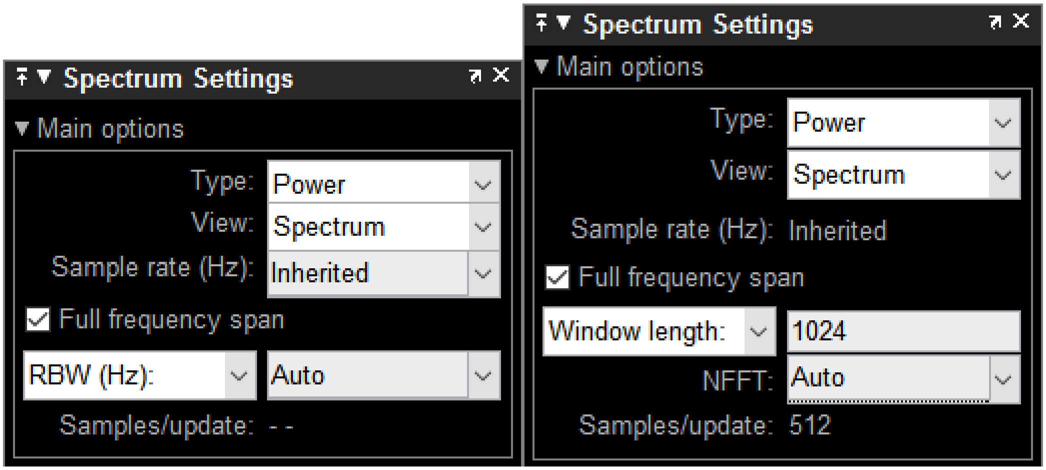
Spectrum Settings

The **Spectrum Settings** panel appears at the right side of the Spectrum Analyzer figure. This panel enables you to modify settings to control the manner in which the spectrum is calculated. You can choose to hide or display the **Spectrum Settings** panel. In the Spectrum Analyzer menu, select **View** > **Spectrum Settings**. Alternatively, in the Spectrum Analyzer toolbar, select the Spectrum Settings  button.

The **Spectrum Settings** panel is separated into four expandable panes, labeled **Main options**, **Spectrogram options**, **Window options**, and **Trace options**.

Main Options Pane

The **Main Options** pane enables you to modify or view the main options.



- **Type** — The type of spectrum to display. Tunable (Simulink)
  - `Power` — power spectrum
  - `Power density` — power spectral density, the magnitude of the spectrum normalized to a bandwidth of 1 hertz
  - `RMS` — root mean squared of the spectrum
- **View** — The spectrum view to display. Tunable (Simulink)
  - `Spectrum` — frequency spectrum
  - `Spectrogram` — frequency spectrum over time. The most recent spectrogram update is at the bottom of the display and time scrolls from the bottom to the top of the display.
  - `Spectrum and spectrogram` — duel view of spectrum and spectrogram.
- **Sample rate (Hz)** — The sample rate, in hertz, of the input signals. Tunable (Simulink)
- **Full frequency span** — Compute and plot the spectrum over the entire *Nyquist* frequency interval.

  By default, when the **Two-sided spectrum** check box is also enabled, the Nyquist interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$

hertz. If you clear the **Two-sided spectrum** check box, the Nyquist interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz.

Enabling this check box is equivalent to setting the FrequencySpan property to `'Full'`..

- **Span (Hz)** and **CF (Hz)**, or **FStart (Hz)** and **FStop (Hz)** — When **Span (Hz)** is showing in the **Main Options** pane, you define the range of values shown on the *frequency*-axis on the Spectrum Analyzer window using frequency span and center frequency. From the drop-down list, select **FStart (Hz)** to define the range of *frequency*-axis values using start frequency and stop frequency instead.

  - **Span (Hz)** — The frequency span, in hertz. This parameter defines the range of values shown on the *frequency*-axis on the Spectrum Analyzer window. When this parameter is set to a numeric value, then it is equivalent to the Span property when the FrequencySpan property is set to `'Span and Center Frequency'`. Tunable (Simulink).
  - **CF (Hz)** — The center frequency, in hertz. This parameter defines the value shown at the middle point of the *frequency*-axis on the Spectrum Analyzer window. This parameter is equivalent to the CenterFrequency property when the FrequencySpan property is set to `'Span and Center Frequency'`. Tunable (Simulink).
  - **FStart (Hz)** — The start frequency, in hertz. This parameter defines the value shown at the leftmost side of the *frequency*-axis on the Spectrum Analyzer window. This parameter is equivalent to the `StartFrequency` on page 4-1584 property when the FrequencySpan property is set to `'Start and stop frequencies'`. Tunable (Simulink).
  - **FStop (Hz)** — The stop frequency, in hertz. The parameter defines the value shown at the rightmost side of the *frequency*-axis on the Spectrum Analyzer window. This parameter is equivalent to the StopFrequency property when the FrequencySpan property is set to `'Start and stop frequencies'`. Tunable (Simulink).

- **RBW (Hz) / Window length** — The frequency resolution method.

  If set to **RBW (Hz)**, the resolution bandwidth, in hertz. This parameter defines the smallest positive frequency that can be resolved. By default, this parameter is set to `Auto`. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 *RBW* intervals over the specified *Frequency span*. This case is equivalent to when you set the RBWSource property to `'Auto'`.

If you set this parameter to a numeric value, then you must specify a value that ensures there are at least two *RBW* intervals over the specified frequency span. In other words, the ratio of the overall span to *RBW* must be at least two: $\frac{span}{RBW} > 2$. In this case, this parameter is equivalent to the RBW property, when `RBWSource` is set to `'Property'`.

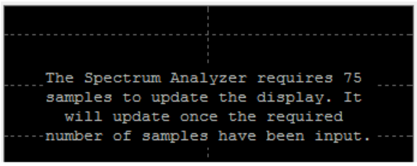
If set to **Window length**, the length of the window, in samples, to compute the spectral estimates. The window length must be an integer scalar greater than 2.

. Tunable (Simulink).

- **NFFT** — The number of Fast Fourier Transform (FFT) points. This parameter is available only when the frequency resolution method is **Window length**. This parameter defines the length of the FFT that Spectrum Analyzer uses to compute spectral estimates. Acceptable options are `Auto` or a positive, scalar integer. The **NFFT** (FFTLength) value must be greater than or equal to the WindowLength. By default, when **NFFT** is set to `Auto`, Spectrum Analyzer sets the number of FFT points to the window length or 1024, whichever is larger. This case is equivalent to when you set the FFTLengthSource property to `'Auto'`.

  When this parameter is set to a positive integer, this parameter is equivalent to the `n` parameter that you can set when you run the MATLAB `fft` function. In this case, this parameter is also equivalent to `FFTLength` when you set `FFTLengthSource` to `'Property'`. Tunable (Simulink).
- **Samples/update** — The number of input samples required to compute one spectral update. You cannot modify this parameter; it is shown here for display purposes only. This parameter is directly related to *RBW* by the following equation: $N_{samples} = \frac{\left(1 - \frac{O_p}{100}\right) \times NENBW \times F_s}{RBW}$ or to the window length by this equation: $N_{samples} = \left(1 - \frac{O_p}{100}\right) x \ WindowLength$. *NENBW* is the normalized effective noise bandwidth, a factor of the windowing method used, which is shown in the **Window Options** pane. F_s is the sample rate. If the number of samples provided in the input

are not sufficient to achieve the resolution bandwidth that you specify, Spectrum Analyzer produces a message on the display as shown in the following figure.



Spectrogram Options Pane

The **Spectrogram options** pane contains options for modifying the spectrogram. These options only display when the **Type** is `Spectrogram` or `Spectrum and spectrogram`

- **Channel** — Select the signal channel for which the spectrogram settings apply. This option displays only if there is more than one signal channel input.
- **Time res. (s)** — The time resolution, in seconds. This parameter is available only in spectrogram mode. *Time resolution* is the amount of data, in seconds, used to compute a spectrogram line. The *minimum attainable resolution* is the amount of time required to compute a single spectral estimate. The tooltip displays the minimum attainable resolution based on the current settings. You can get or set the minimum attainable resolution value from the TimeResolution property. Tunable (Simulink).
- **Time span (s)** — The time span over which the Spectrum Analyzer displays the spectrogram, in seconds. This parameter is available only in spectrogram mode. The *time span* is the product of the number of spectral lines you want and the time resolution. The tooltip displays the minimum allowable time span, given the current settings. If the time span is set to `Auto`, 100 spectral lines are used. This parameter is equivalent to the TimeSpan property. Tunable (Simulink).

Window Options Pane

The **Window Options** pane enables you to modify the window options.

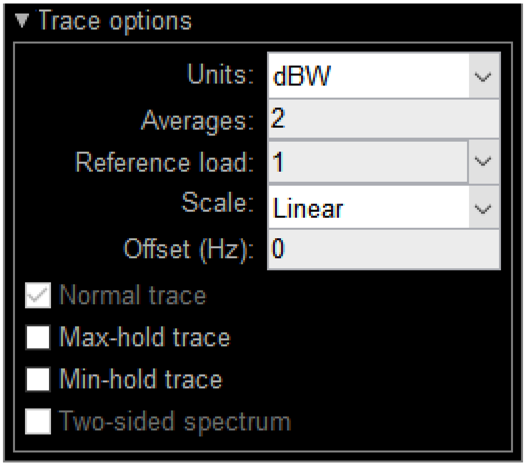
- **Overlap (%)** — The segment overlap percentage. This parameter defines the amount of overlap between the previous and current buffered data segments. The value must be greater than or equal to zero and less than 100. This parameter is equivalent to the OverlapPercent property. Tunable (Simulink).
- **Window** — The windowing method to apply to the spectrum. Windowing is used to control the effect of sidelobes in spectral estimation. The window you specify affects

the segment length required to achieve a resolution bandwidth and the required number of samples per update. For more information about windowing, see the `Window Function` block. This parameter is equivalent to the Window property. Tunable (Simulink).

- **Attenuation (dB)** — The sidelobe attenuation, in decibels (dB). This property applies only when you set the **Window** parameter to `Chebyshev` or `Kaiser`. You must specify a value greater than or equal to `45`. If you use a Chebyshev window and set the sidelobe attenuation to less than 35 dB, the desired resolution bandwidth may not be achieved. This parameter is equivalent to the SidelobeAttenuation property. Tunable (Simulink).
- **NENBW** — Normalized Effective Noise Bandwidth of the window. You cannot modify this parameter; it is a readout shown here for display purposes only. This parameter is a measure of the noise performance of the window. It is the width of a rectangular filter that accumulates the same noise power with the same peak power gain. NENBW can be calculated from the windowing function using the following equation: $NENBW = N_{window} \dfrac{\sum_{n=1}^{N_{window}} w^2(n)}{\left[\sum_{n=1}^{N_{window}} w(n)\right]^2}$. The rectangular window has the smallest NENBW, with a value of 1. All other windows have a larger NENBW value. For example, the Hann window has an NENBW value of approximately 1.5.

Trace Options Pane

The **Trace Options** pane enables you to modify the trace options.



- **Units** — The units of the spectrum. The available values depends on the value of **Type**. Available options include `dBm`, `dBW`, `Watts`, `Vrms`, and `dBV`. This parameter is equivalent to the SpectrumUnits property. Tunable (Simulink).
- **Averages** — Specify as a positive, scalar integer the number of spectral averages. This parameter applies only when the Spectrum **View** is `Spectrum` or `Spectrum and spectrogram`. Spectrum Analyzer computes the current power spectrum estimate by averaging the last *N* power spectrum estimates. This property defines the number of spectral averages, *N*. This parameter is equivalent to the SpectralAverages property. Tunable (Simulink).
- **Reference load** — The reference load, in ohms, used to scale the spectrum. Specify as a real, positive scalar the load, in ohms, that the Spectrum Analyzer uses as a reference to compute power values. This parameter is equivalent to the ReferenceLoad property. Tunable (Simulink).
- **Scale** — Linear or logarithmic scale. When the frequency span contains negative frequency values, Spectrum Analyzer disables the logarithmic option. This parameter is equivalent to the FrequencyScale property. Tunable (Simulink).
- **Offset** — The constant frequency offset to apply to the entire spectrum or a vector of frequencies to apply to each spectrum for multiple inputs. The constant offset parameter is added to the values on the *frequency*-axis in the Spectrum Analyzer window. It is not used in any spectral computations. You must take this parameter into consideration when you set the **Span (Hz)** and **CF (Hz)** parameters to ensure that the frequency span is within Nyquist limits. The Nyquist interval is

$\left[-\frac{SampleRate}{2},\frac{SampleRate}{2}\right]+FrequencyOffset$ hertz if **Two-sided spectrum** is selected, and $\left[0,\frac{SampleRate}{2}\right]+FrequencyOffset$ hertz otherwise. This parameter is equivalent to the FrequencyOffset property.

- **Normal trace** — Normal trace view. This parameter applies only when the Spectrum **View** is `Spectrum` or `Spectrum and spectrogram`. By default, when this check box is enabled, Spectrum Analyzer calculates and plots the power spectrum or power spectrum density computed by the *Welch* estimator. The Welch estimator performs a smoothing operation by averaging a number of spectral estimates. To clear this check box, you must first select either the **Max hold trace** or the **Min hold trace** check box. This parameter is equivalent to the PlotNormalTrace property. Tunable (Simulink).
- **Max hold trace** — Maximum hold trace view. This parameter applies only when the Spectrum **View** is `Spectrum` or `Spectrum and spectrogram`. Select this check box to enable Spectrum Analyzer to plot the maximum spectral values of all the estimates obtained. This parameter is equivalent to the PlotMaxHoldTrace property. Tunable (Simulink).
- **Min hold trace** — Minimum hold trace view. This parameter applies only when the Spectrum **View** is `Spectrum` or `Spectrum and spectrogram`. Select this check box to enable Spectrum Analyzer to plot the minimum spectral values of all the estimates obtained. This parameter is equivalent to the PlotMinHoldTrace property. Tunable (Simulink).
- **Two-sided spectrum** — Select this check box to enable two-sided spectrum view. In this view, both negative and positive frequencies are shown. If you clear this check box, Spectrum Analyzer shows a one-sided spectrum with only positive frequencies. Spectrum Analyzer requires that this parameter is selected when the input signal is complex-valued. When your signal includes DC input, you should display a two-sided spectrum to resolve low frequency power and DC power. This parameter is equivalent to the property.

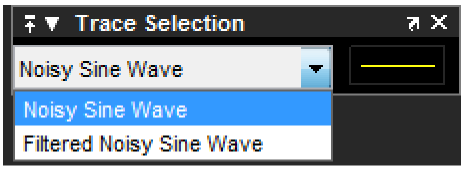
Measurements Panels

The Measurements panels are the panels that appear to the right side of the Spectrum Analyzer figure.

Note: When **View** is `Spectrogram`, the Measurements panels are not available.

Trace Selection Panel

When you use the scope to view multiple signals, the Trace Selection panel appears if you have more than one signal displayed and you click any of the other Measurements panels. The Measurements panels display information about only the signal chosen in this panel. Choose the signal name for which you would like to display time domain measurements. See the following figure.



You can choose to hide or display the **Trace Selection** panel. In the Scope menu, select **Tools** > **Measurements** > **Trace Selection**.

Cursor Measurements Panel

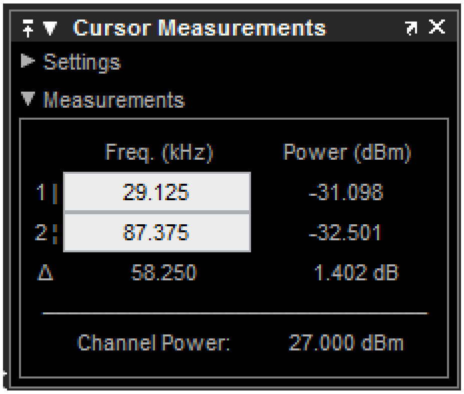
The **Cursor Measurements** panel displays screen cursors. The panel provides two types of cursors for measuring signals. Waveform cursors are vertical cursors that track along the signal. Screen cursors are both horizontal and vertical cursors that you can place anywhere in the display.

Note: If a data point in your signal has more than one value, the cursor measurement at that point is undefined and no cursor value is displayed.

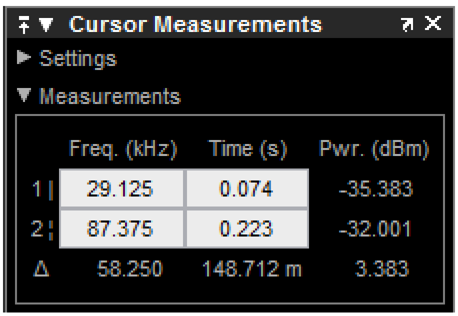
In the Scope menu, select **Tools** > **Measurements** > **Cursor Measurements**.

Alternatively, in the Scope toolbar, click the Cursor Measurements  button.

The **Cursor Measurements** panel appears as follows for the spectrum and dual view.



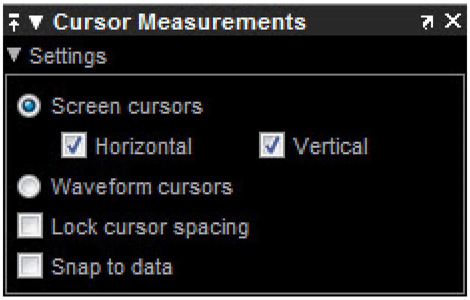
The **Cursor Measurements** panel appears as follows for the spectrogram view. You must pause the spectrogram display before you can use cursors.



The **Cursor Measurements** panel is separated into two panes, labeled **Settings** and **Measurements**. You can expand each pane to see the available options.

You can use the mouse or the left and right arrow keys to move vertical or waveform cursors and the up and down arrow keys for horizontal cursors.

The **Settings** pane enables you to modify the type of screen cursors used for calculating measurements. When more than one signal is displayed, you can assign cursors to each trace individually.



- **Screen Cursors** — Shows screen cursors (for spectrum and dual view only).
- **Horizontal** — Shows horizontal screen cursors (for spectrum and dual view only).
- **Vertical** — Shows vertical screen cursors (for spectrum and dual view only).
- **Waveform Cursors** — Shows cursors that attach to the input signals (for spectrum and dual view only).
- **Lock Cursor Spacing** — Locks the frequency difference between the two cursors.
- **Snap to Data** — Positions the cursors on signal data points.

Measurements Pane

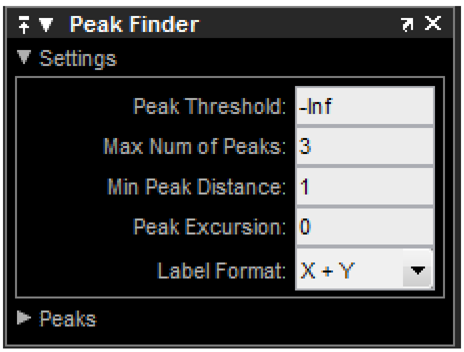
The **Measurements** pane displays the frequency (Hz) , time (s), and power (dBm) value measurements. Time is displayed only in spectrogram mode. **Channel Power** shows the total power between the cursors.

- **1** — Shows or enables you to modify the frequency, time (for spectrograms only), or both, at cursor number one.
- **2** — Shows or enables you to modify the frequency, time (for spectrograms only), or both, at cursor number two.
- **Δ** — Shows the absolute value of the difference in the frequency, time (for spectrograms only), or both, and power between cursor number one and cursor number two.
- **Channel Power** — Shows the total power in the channel defined by the cursors.

The letter after the value associated with a measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix.

Peak Finder Panel

The **Peak Finder** panel displays the maxima, showing the *x*-axis values at which they occur. Peaks are defined as a local maximum where lower values are present on both sides of a peak. Endpoints are not considered to be peaks. This panel allows you to modify the settings for peak threshold, maximum number of peaks, and peak excursion. You can choose to hide or display the **Peak Finder** panel. In the scope menu, select **Tools** > **Measurements** > **Peak Finder**. Alternatively, in the scope toolbar, select the Peak Finder  button.



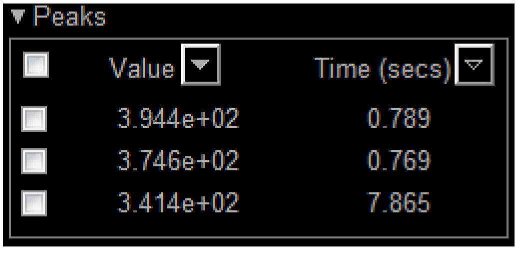
The **Peak finder** panel is separated into two panes, labeled **Settings** and **Peaks**. You can expand each pane to see the available options.

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input signal. For more information on the algorithms this pane uses, see the Signal Processing Toolbox `findpeaks` function reference.

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the `MINPEAKHEIGHT` parameter, which you can set when you run the `findpeaks` function.
- **Max Num of Peaks** — The maximum number of peaks to show. The value you enter must be a scalar integer from 1 through 99. This setting is equivalent to the `NPEAKS` parameter, which you can set when you run the `findpeaks` function.
- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the `MINPEAKDISTANCE` parameter, which you can set when you run the `findpeaks` function.

- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples The peak excursion setting is equivalent to the `THRESHOLD` parameter, which you can set when you run the `findpeaks` function.
- **Label Format** — The coordinates to display next to the calculated peak values on the plot. To see peak values, expand the **Peaks** pane and select the check boxes associated with individual peaks of interest. By default, both *x*-axis and *y*-axis values are displayed on the plot. Select which axes values you want to display next to each peak symbol on the display.
  - `X+Y` — Display both *x*-axis and *y*-axis values.
  - `X` — Display only *x*-axis values.
  - `Y` — Display only *y*-axis values.

The **Peaks** pane displays all of the largest calculated peak values. It also shows the coordinates at which the peaks occur, using the parameters you define in the **Settings** pane. You set the **Max Num of Peaks** parameter to specify the number of peaks shown in the list.



The numerical values displayed in the **Value** column are equivalent to the `pks` output argument returned when you run the `findpeaks` function. The numerical values displayed in the second column are similar to the `locs` output argument returned when you run the `findpeaks` function.

The Peak Finder displays the peak values in the **Peaks** pane. By default, the **Peak Finder** panel displays the largest calculated peak values in the **Peaks** pane in decreasing order of peak height. Use the sort descending button ( ) to rearrange the category and order by which Peak Finder displays peak values. Click this button again to sort the peaks in ascending order instead. When you do so, the arrow changes direction to become the sort ascending button ( ). A filled sort button indicates that the peak values are currently sorted in the direction of the button arrow. If the sort button is not filled

( ), then the peak values are sorted in the opposite direction of the button arrow. The **Max Num of Peaks** parameter still controls the number of peaks listed.

Use the check boxes to control which peak values are shown on the display. By default, all check boxes are cleared and the **Peak Finder** panel hides all the peak values. To show all the peak values on the display, select the check box in the top-left corner of the **Peaks** pane. To hide all the peak values on the display, clear this check box. To show an individual peak, select the check box directly to the left of its **Value** listing. To hide an individual peak, clear the check box directly to the left of its **Value** listing.

The Peaks are valid for any units of the input signal. The letter after the value associated with each measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Channel Measurements Panel

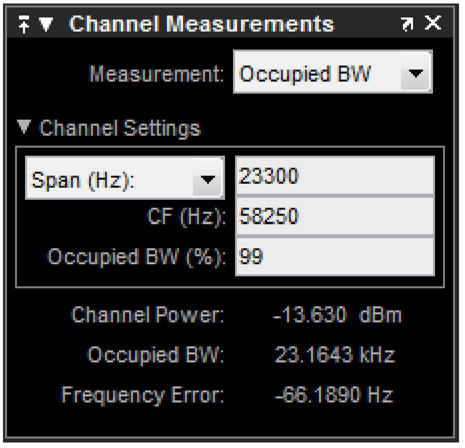
The **Channel Measurements** panel displays occupied bandwidth or adjacent channel power ratio (ACPR) measurements. You can choose to hide or display this pane in the Scope menu by selecting **Tools** > **Measurements** > **Channel Measurements**.

Alternatively, in the Scope toolbar, click the Cursor Measurements button.

In addition to the measurements, the **Channel Measurements** panel has an expandable **Channel Settings** pane.

- **Measurement** — The type of measurement data to display. Available options are `Occupied BW` or `ACPR`. See "Algorithms" on page 2-1592 for information on how Occupied BW is calculated. ACPR is the adjacent channel power ratio, which is the ratio of the main channel power to the adjacent channel power.

When you select `Occupied BW` as the **Measurement**, the following fields appear.

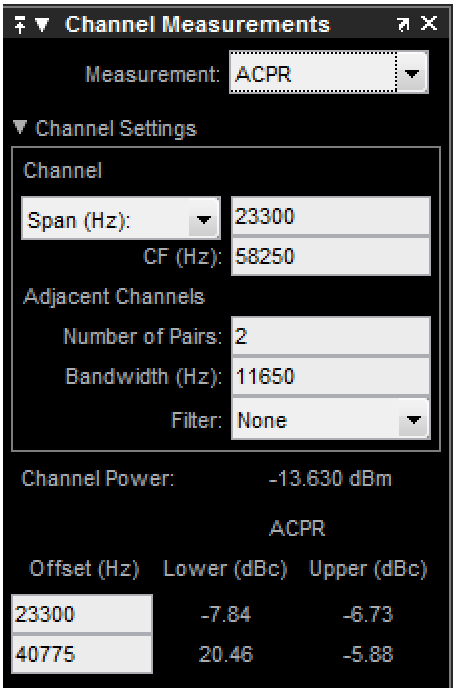


- **Channel Settings** — Enables you to modify the parameters for calculating the channel measurements.

  Channel Settings for Occupied BW

  - Select the frequency span of the channel, `Span(Hz)`, and specify the center frequency **CF (Hz)** of the channel. Alternatively, select the starting frequency, `FStart(Hz)`, and specify the starting frequency and ending frequency (**FStop (Hz)**) values of the channel.
  - **CF (Hz)** — The center frequency of the channel.
  - **Occupied BW (%)** — The percentage of the total integrated power of the spectrum centered on the selected channel frequency over which to compute the occupied bandwidth.
- **Channel Power** — The total power in the channel.
- **Occupied BW** — The bandwidth containing the specified **Occupied BW (%)** of the total power of the spectrum. This setting is available only if you select `Occupied BW` as the **Measurement** type.
- **Frequency Error** — The difference between the center of the occupied band and the center frequency (**CF**) of the channel. This setting is available only if you select `Occupied BW` as the **Measurement** type.

When you select `ACPR` as the **Measurement**, the following fields appear.



- **Channel Settings** — Enables you to modify the parameters for calculating the channel measurements.

  ### Channel Settings for ACPR

  - Select the frequency span of the channel, `Span (Hz)`, and specify the center frequency **CF (Hz)** of the channel. Alternatively, select the starting frequency, `FStart(Hz)`, and specify the starting frequency and ending frequency (**FStop (Hz)**) values of the channel.
  - **CF (Hz)** — The center frequency of the channel.
  - **Number of Pairs** — The number of pairs of adjacent channels.
  - **Bandwidth (Hz)** — The bandwidth of the adjacent channels.
  - **Filter** — The filter to use for both main and adjacent channels. Available filters are `None`, `Gaussian`, and `RRC` (root-raised cosine).
- **Channel Power** — The total power in the channel.

- **Offset (Hz)** — The center frequency of the adjacent channel with respect to the center frequency of the main channel. This setting is available only if you select `ACPR` as the **Measurement** type.
- **Lower (dBc)** — The power ratio of the lower sideband to the main channel. This setting is available only if you select `ACPR` as the **Measurement** type.
- **Upper (dBc)** — The power ratio of the upper sideband to the main channel. This setting is available only if you select `ACPR` as the **Measurement** type.

Distortion Measurements Panel

The **Distortion Measurements** panel displays harmonic distortion and intermodulation distortion measurements. You can choose to hide or display this panel in the Scope menu by selecting **Tools** > **Measurements** > **Distortion Measurements**. Alternatively, in the Scope toolbar, click the Distortion Measurements  button.

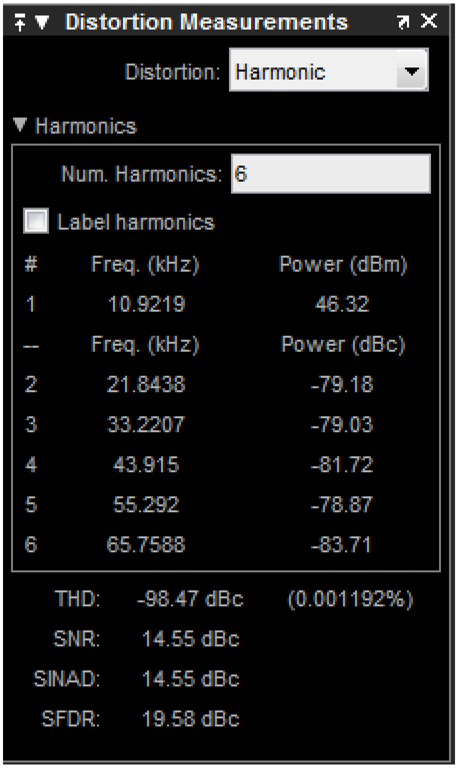
The **Distortion Measurements** panel has an expandable **Harmonics** pane, which shows measurement results for the specified number of harmonics.

Note: For an accurate measurement, ensure that the fundamental signal (for harmonics) or primary tones (for intermodulation) is larger than any spurious or harmonic content. To do so, you may need to adjust the resolution bandwidth (RBW) of the spectrum analyzer. Make sure that the bandwidth is low enough to isolate the signal and harmonics from spurious and noise content. In general, you should set the RBW so that there is at least a 10dB separation between the peaks of the sinusoids and the noise floor. You may also need to select a different spectral window to obtain a valid measurement.

- **Distortion** — The type of distortion measurements to display. Available options are `Harmonic` or `Intermodulation`. Select `Harmonic` if your system input is a single sinusoid. Select `Intermodulation` if your system input is two equal amplitude sinusoids. Intermodulation can help you determine distortion when only a small portion of the available bandwidth will be used.

  See “Algorithms” on page 2-1592 for information on how distortion measurements are calculated.

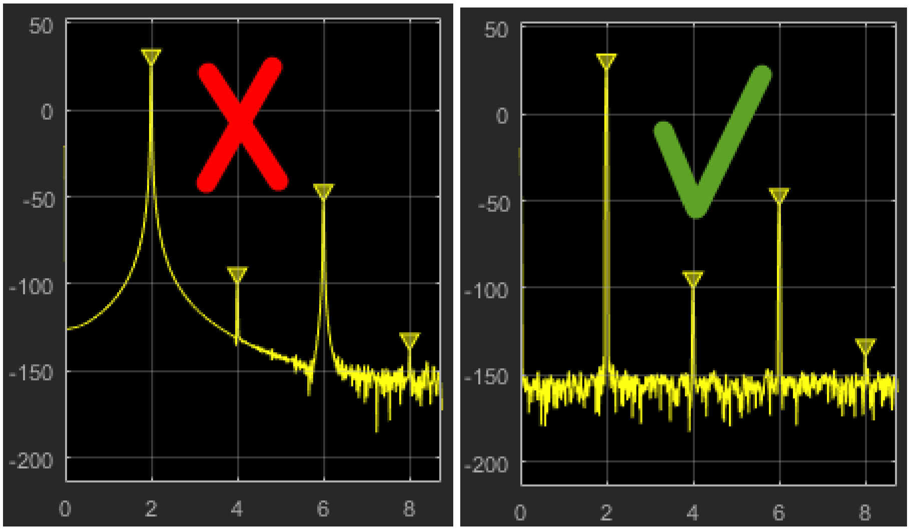
When you select `Harmonic` as the **Distortion**, the following fields appear.



The harmonic distortion measurement automatically locates the largest sinusoidal component (fundamental signal frequency). It then computes the harmonic frequencies and power in each harmonic in your signal. Any DC component is ignored. Any harmonics that are outside the spectrum analyzer's frequency span are not included in the measurements. Adjust your frequency span so that it includes all the desired harmonics.

Note: To best view the harmonics, make sure that your fundamental frequency is set high enough to resolve the harmonics. However, this frequency should not be so high that aliasing occurs. For the best display of harmonic distortion, your plot should not show skirts, which indicate frequency leakage. Additionally, the noise floor should be visible.

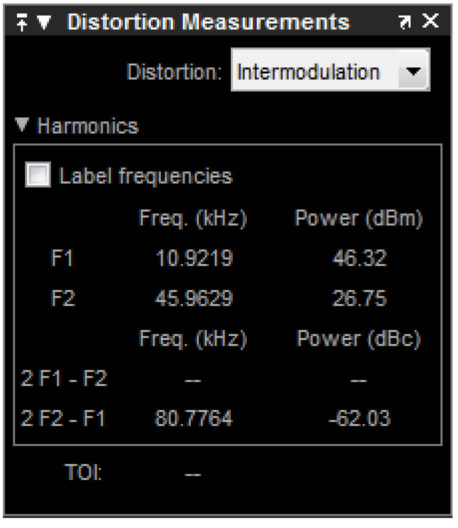
For a better display, try a Kaiser window with a large sidelobe attenuation (e.g. between 100–300 db).



- **Num. Harmonics** — Number of harmonics to display, including the fundamental frequency. Valid values of **Num. Harmonics** are from 2 to 99. The default value is 6.
- **Label Harmonics** — Select **Label Harmonics** to add numerical labels to each harmonic in the spectrum display.
- **1** — The fundamental frequency, in hertz, and its power, in decibels of the measured power referenced to one milliwatt (dBm).
- **2, 3, ...** — The harmonics frequencies, in hertz, and their power in decibels relative to the carrier (dBc). If the harmonics are at the same level or exceed the fundamental frequency, reduce the input power.
- **THD** — The total harmonic distortion. This value represents the ratio of the power in the harmonics, D, to the power in the fundamental frequency, S. If the noise power is too high in relation to the harmonics, the THD value is not accurate. In this case, lower the resolution bandwidth or select a different spectral window. $THD = 10\log_{10}(D/S)$.
- **SNR** — Signal-to-noise ratio (SNR). This value represents the ratio of power in the fundamental frequency, S, to the power of all nonharmonic content, N, including spurious signals, in decibels relative to the carrier (dBc). $SNR = 10\log_{10}(S/N)$. If you see – – as the reported SNR, your signal's total non-harmonic content is less than 30% of the total signal.

- **SINAD** — Signal-to-noise-and-distortion. This value represents the ratio of the power in the fundamental frequency, S to all other content (including noise, N, and harmonic distortion, D), in decibels relative to the carrier (dBc). $SINAD = 10\log_{10}(S/(N+D))$.
- **SFDR** — Spurious free dynamic range (SFDR). This value represents the ratio of the power in the fundamental frequency, S, to power of the largest spurious signal, R, regardless of where it falls in the frequency spectrum. The worst spurious signal may or may not be a harmonic of the original signal. SFDR represents the smallest value of a signal that can be distinguished from a large interfering signal. SFDR includes harmonics. $SFDR = 10\log_{10}(S/R)$.

When you select `Intermodulation` as the **Distortion**, the following fields appear.



The intermodulation distortion measurement automatically locates the fundamental, first-order frequencies (F1 and F2). It then computes the frequencies of the third-order intermodulation products (2*F1-F2 and 2*F2-F1).

- **Label frequencies** — Select **Label frequencies** to add numerical labels to the first-order intermodulation product and third-order frequencies in the spectrum analyzer display.
- **F1** — Lower fundamental first-order frequency

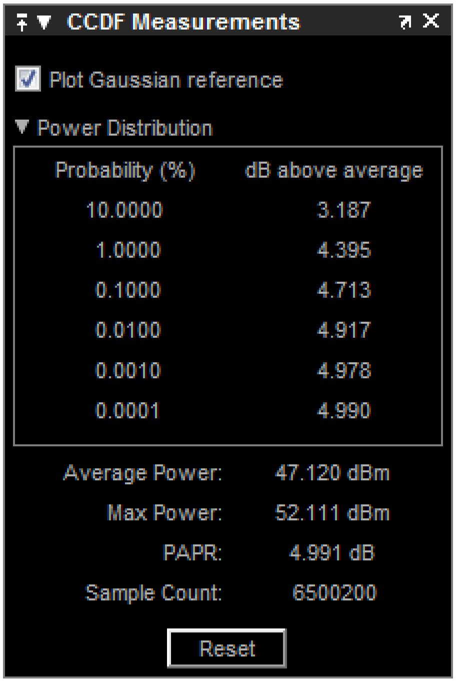
- **F2** — Upper fundamental first-order frequency
- **2F1 - F2** — Lower intermodulation product from third-order harmonics
- **2F2 - F1** — Upper intermodulation product from third-order harmonics
- **TOI** — Third-order intercept point. If the noise power is too high in relation to the harmonics, the TOI value will not be accurate. In this case, you should lower the resolution bandwidth or select a different spectral window. If the TOI has the same amplitude as the input two-tone signal, reduce the power of that input signal.

CCDF Measurements Panel

The **CCDF Measurements** panel displays complimentary cumulative distribution function measurements. CCDF measurements in this scope show the probability of a signal's instantaneous power being a specified level above the signal's average power. These measurements are useful indicators of a signal's dynamic range.

To compute the CCDF measurements, each input sample is quantized to 0.01 dB increments. Using a histogram 100 dB wide (10,000 points at 0.01 dB increments), the largest peak encountered is placed in the last bin of the histogram. If a new peak is encountered, the histogram shifts to make room for that new peak.

You can choose to hide or display this panel in the Scope menu by selecting **Tools** > **Measurements** > **CCDF Measurements**. Alternatively, in the Scope toolbar, click the CCDF Measurements  button.



- **Plot Gaussian reference** — Select **Plot Gaussian reference** to show the Gaussian white noise reference signal on the plot.
- **Probability (%)** — The percentage of the signal that contains the power level above the value listed in the **dB above average** column
- **dB above average** — The expected minimum power level at the associated **Probability (%)**.
- **Average Power** — The average power level of the signal since the start of simulation or from the last reset.

  Max Power — The maximum power level of the signal since the start of simulation or from the last reset.
- **PAPR** — The ratio of the peak power to the average power of the signal. PAPR should be less that 100 dB to obtain accurate CCDF measurements. If PAPR is above 100 dB, only the highest 100 dB power levels are plotted in the display and shown in the distribution table.
- **Sample Count** — The total number of samples used to compute the CCDF.

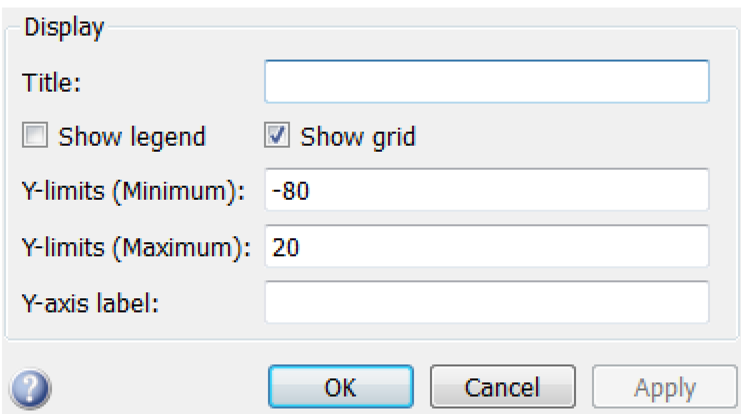
- **Reset** — Clear all current CCDF measurements and restart.

Visuals — Spectrum Properties

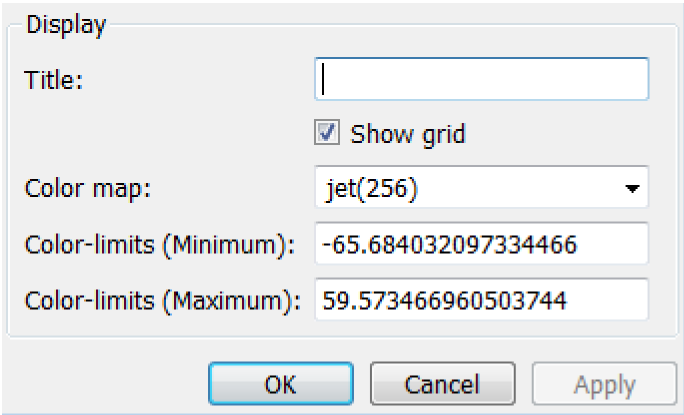
The Visuals—Spectrum Properties dialog box controls the visual configuration settings of the Spectrum Analyzer display. From the Spectrum Analyzer menu, select **View** > **Configuration Properties** to open this dialog box. Alternatively, in the Spectrum Analyzer toolbar, click the Configuration Properties  button.

Display Pane

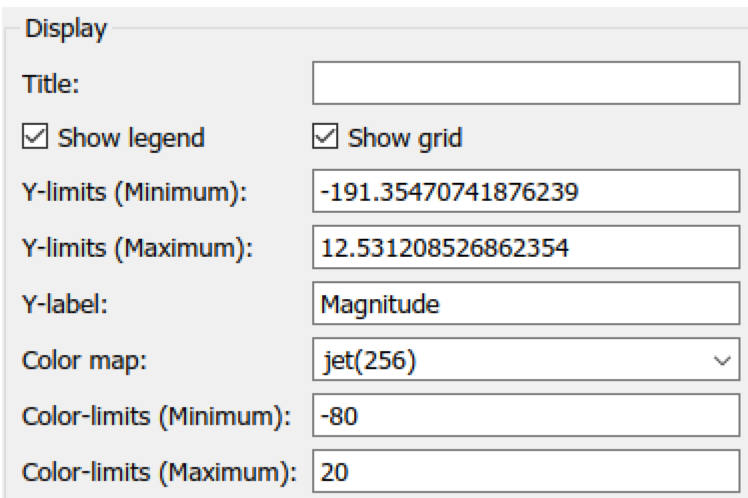
When the Spectrum **View** is `Spectrum`, the **Display** pane of the Visuals—Spectrum Properties dialog box appears as follows:



When the Spectrum **View** is `Spectrogram` the **Display** pane of the Visuals—Spectrum Properties dialog box appears as follows:



When the Spectrum **View** is `Spectrum or spectroogram` the **Display** pane of the Visuals—Spectrum Properties dialog box appears as follows:



Title

Specify the display title as a character vector. Enter %<SignalLabel> to use the signal labels in the Simulink Model as the axes titles. This property is Tunable (Simulink).

By default, the display has no title.

Show legend

Select this check box to show the legend in the display. The channel legend displays a name for each channel of each input signal. When the legend appears, you can place it anywhere inside of the scope window. To turn off the legend, clear the **Show legend** check box.

You can edit the name of any channel in the legend by double-clicking the current name and entering a new channel name. By default, if the signal has multiple channels, the scope uses an index number to identify each channel of that signal. To change the appearance of any channel of any input signal in the scope window, from the scope menu, select **View** > **Style**.

The legend lets you modify what signals are shown. To show only one signal, click the signal name. To toggle a signal on/off, right-click the signal name.

This parameter applies only when the **View Type** is `Spectrum` or `Spectrum and spectrogram`. Tunable (Simulink)

Show grid

When you select this check box, a grid appears in the display of the scope figure. To hide the grid, clear this check box. Tunable (Simulink)

Y-limits (Minimum)

Specify the minimum value of the *y*-axis. Tunable (Simulink)

Y-limits (Maximum)

Specify the maximum value of the *y*-axis. Tunable (Simulink)

Y-axis label

Specify the text for the scope to display to the left of the *y*-axis. Regardless of this property, Spectrum Analyzer always displays power units after this text as one of `'dBm'`, `'dBW'`, `'Watts'`, `'dBm/Hz'`, `'dBW/Hz'`, `'Watts/Hz'`, `Vrms`, or `dBV`Tunable (Simulink).

Color map

Select the color map for the spectrogram, or enter a 3-column matrix expression for the color map. See `colormap` for information. Tunable (Simulink).

Color-limits (Minimum)

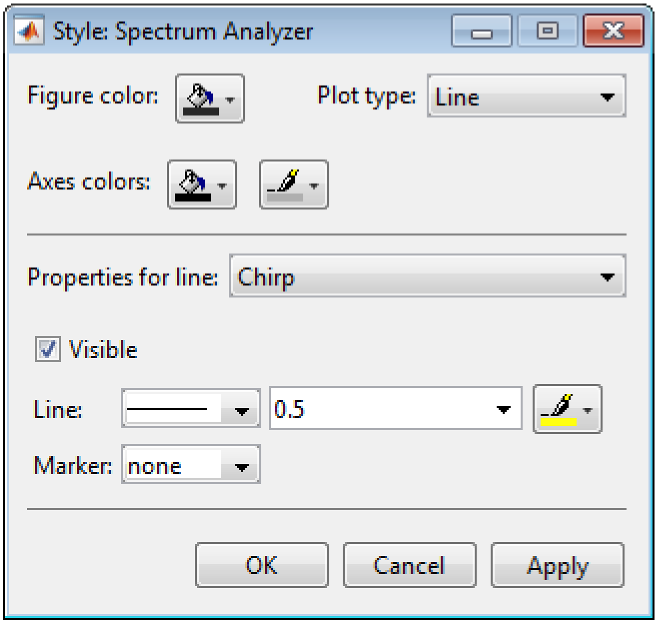
Set the signal power for the minimum color value of the spectrogram. Tunable (Simulink).

Color-limits (Maximum)

Set the signal power for the maximum color value of the spectrogram. Tunable (Simulink).

Style Dialog Box

In the **Style** dialog box, you can customize the style of spectrum display. This dialog box is not available for the spectrogram view. You are able to change the color of the figure, the background and foreground colors of the axes, and properties of the lines. From the Spectrum Analyzer menu, select **View** > **Style** to open this dialog box.



Properties

The **Style** dialog box allows you to modify the following properties of the Spectrum Analyzer figure:

Figure color

Specify the color that you want to apply to the background of the scope figure. By default, the figure color is gray.

Plot type

Specify whether to display a `Line` or `Stem` plot.

Axes colors

Specify the color that you want to apply to the background of the axes.

Properties for line

Specify the channel for which you want to modify the visibility, line properties, and marker properties.

Visible

Specify whether the selected channel should be visible. If you clear this check box, the line disappears.

Line

Specify the line style, line width, and line color for the selected channel.

Marker

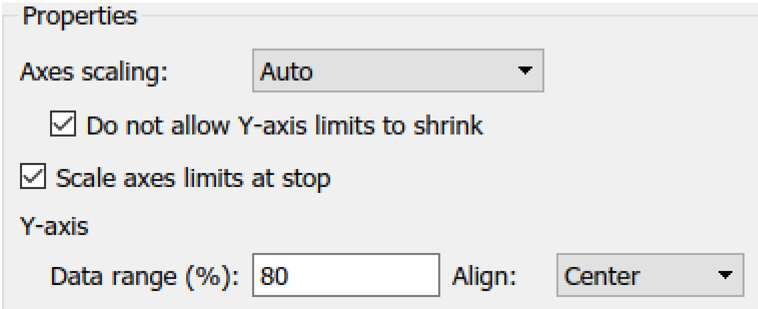
Specify marks for the selected channel to show at its data points. This parameter is similar to the `Marker` property for plot objects. You can choose any of the marker symbols from the dropdown.

Tools — Axes Scaling Properties

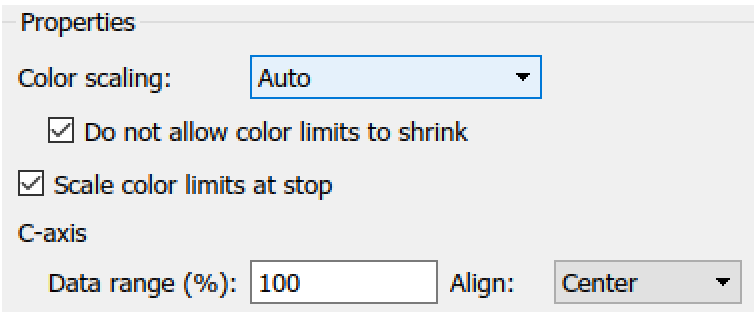
The Tools—Axes Scaling Properties dialog box allows you to zoom in on and zoom out of your data automatically. You can also scale the axes of the Spectrum Analyzer. In the Spectrum Analyzer menu, select **Tools** > **Scaling Properties** to open this dialog box.

Properties

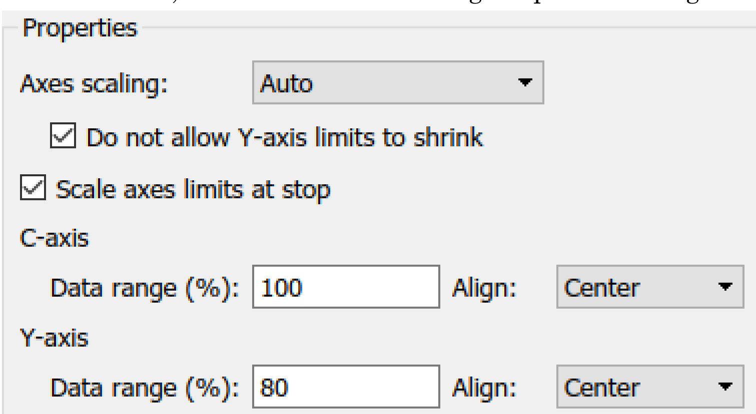
For the spectrum view, the Tools—Axes Scaling Properties dialog box appears as:



For spectrogram view, the Tools—Axes Scaling Properties dialog box appears as:



For dual view, the Tools—Axes Scaling Properties dialog box appears as:



Note: The X-axis scaling options only apply when using CCDF measurements.

Axes scaling/Color scaling

Specify when the scope automatically scales the axes. If the spectrogram is displayed, specify when the scope automatically scales the color. You can select one of the following options:

- `Manual` — When you select this option, the scope does not automatically scale the axes or color. You can manually scale the axes or color in any of the following ways:
  - Select **Tools** > **Scaling Properties**.
  - Press one of the **Scale Axis Limits** toolbar buttons.
  - When the scope figure is the active window, press **Ctrl** and **A** simultaneously.
- `Auto` — When you select this option, the scope scales the axes or color as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** or **Do not allow color limits to shrink** .
- `After N Updates` — Selecting this option causes the scope to scale the axes or color after a specified number of updates. This option is useful and more efficient when your scope display starts with one axis scale, but quickly reaches a different steady state axis scale. Selecting this option shows the **Number of updates** edit box.

By default, this parameter is set to `Auto`, and the scope does not shrink the *y*-axis limits when scaling the axes or color. Tunable (Simulink).

Do not allow Y-axis limits to shrink / Do not allow color limits to shrink

When you select this property, the *y*-axis are allowed to grow during axes scaling operations. If the spectrogram is displayed, selecting this property allows the color limits to grow during axis scaling. If you clear this check box, the *y*-axis or color limits can shrink during axes scaling operations.

This property appears only when you select `Auto` for the **Axis scaling** or **Color scaling** property. When you set the **Axes scaling** or **Color scaling** property to `Manual` or `After N Updates`, the *y*-axis or color limits can shrink. Tunable (Simulink).

Number of updates

Specify as a positive integer the number of updates after which to scale the axes. If the spectrogram is displayed, this property specifies the number of updates after which to scale the color. This property appears only when you select `After N Updates` for the **Axes scaling** or **Color scaling** property. Tunable (Simulink).

Scale axes limits at stop/Scale color limits at stop

Select this check box to scale the axes when the simulation stops. If the spectrogram is displayed, select this check box to scale the color when the simulation stops. The *y*-axis is always scaled. The *x*-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

Y-axis Data range (%) / Color-limits Data range

Set the percentage of the *y*-axis that the scope uses to display the data when scaling the axes. If the spectrogram is displayed, set the percentage of the power values range within the colormap. Valid values are from 1 through 100. For example, if you set this property to `100`, the Scope scales the *y*-axis limits such that your data uses the entire *y*-axis range. If you then set this property to `30`, the scope increases the *y*-axis range or color such that your data uses only 30% of the *y*-axis range or color. Tunable (Simulink).

Y-axis Align / Color-limits Align

Specify where the scope aligns your data along the *y*-axis when it scales the axes. If the spectrogram is displayed, specify where the scope aligns your data along the *y*-axis when it scales the color. You can select `Top`, `Center`, or `Bottom`. Tunable (Simulink).

Autoscale X-axis limits

Check this box to allow the scope to scale the *x*-axis limits when it scales the axes. If **Axes scaling** is set to `Auto`, checking **Autoscale X-axis limits** only scales the data currently within the axes, not the entire signal in the data buffer. If **Autoscale X-axis limits** is on and the resulting axis is greater than the span of the scope, trigger position markers will not be displayed. Triggers are controlled using the Trigger Measurements panel. Tunable (Simulink).

X-axis Data range (%)

Set the percentage of the *x*-axis that the scope uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the scope scales the *x*-axis limits such that your data uses the entire*x*-axis range. If you then set this property to `30`, the scope increases the *x*-axis range such that your data uses only 30% of the *x*-axis range. Use the *x*-axis **Align** property to specify data placement along the *x*-axis.

This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

X-axis Align

Specify how the scope aligns your data along the *x*-axis: `Left`, `Center`, or `Right`. This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

Algorithms

Spectrum Analyzer uses the `RBW` or the `Window Length` setting in the **Spectrum Settings** panel to determine the data window length. The value of the `FrequencyResolutionMethod` property determines whether RBW or window length is used. Then, it partitions the input signal into a number of windowed data segments. Finally, Spectrum Analyzer uses the modified periodogram method to compute spectral updates, averaging the windowed periodograms for each segment.

1. Spectrum Analyzer requires that a minimum number of samples have been provided before it computes a spectral estimate. This number of input samples required to compute one spectral update is shown as **Samples/update** in the **Main options** pane. This value is directly related to resolution bandwidth, *RBW*, by the following equation or to the window length, by the equation shown in step 1b.

$$N_{samples} = \frac{\left(1 - \frac{O_p}{100}\right) \times NENBW \times F_s}{RBW}$$

   The normalized effective noise bandwidth, *NENBW*, is a factor that depends on the windowing method. Spectrum Analyzer shows NENBW in the **Window Options** pane of the **Spectrum Settings** panel. Overlap percentage, O_p, is the value of the **Overlap %** parameter in the **Window Options** pane of the **Spectrum Settings** panel. F_s is the sample rate of the input signal. Spectrum Analyzer shows sample rate in the **Main Options** pane of the **Spectrum Settings** panel.

   a. When in RBW mode, the window length required to compute one spectral update, N_{window}, is directly related to the resolution bandwidth and normalized effective noise bandwidth by the following equation.

$$N_{window} = \frac{NENBW \times F_s}{RBW}$$

When in WindowLength mode, the window length is used as specified.

b The number of input samples required to compute one spectral update, $N_{samples}$, is directly related to the window length and the amount of overlap by the following equation.

$$N_{samples} = \left(1 - \frac{O_p}{100}\right) N_{window}$$

When you increase the overlap percentage, fewer new input samples are needed to compute a new spectral update. For example, if the window length is 100, then the number of input samples required to compute one spectral update is given as shown in the following table.

O_p	$N_{samples}$
0%	100
50%	50
80%	20

c The normalized effective noise bandwidth, *NENBW*, is a window parameter determined by the window length, N_{window}, and the type of window used. If $w(n)$ denotes the vector of N_{window} window coefficients, then *NENBW* is given by the following equation.

$$NENBW = N_{window} \frac{\sum_{n=1}^{N_{window}} w^2(n)}{\left[\sum_{n=1}^{N_{window}} w(n)\right]^2}$$

d When in RBW mode, you can set the resolution bandwidth using the value of the **RBW** parameter on the **Main options** pane of the **Spectrum Settings** panel. You must specify a value to ensure that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two, as given in the following equation.

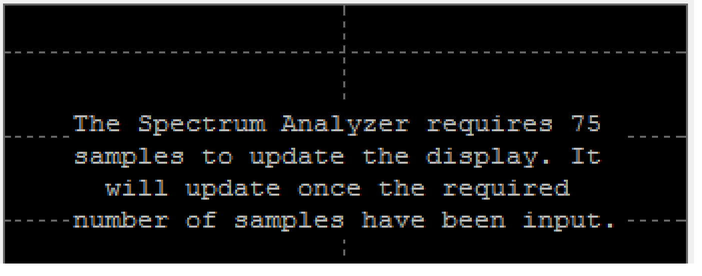
$$\frac{span}{RBW} > 2$$

By default, the **RBW** parameter on the **Main options** pane is set to `Auto`. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified frequency span. Thus, when you set **RBW** to `Auto`, it is calculated by the following equation. $RBW_{auto} = \frac{span}{1024}$

e When in window length mode, you specify N_{window} and the resulting RBW is

$$\frac{NENBW * Fs}{N_{window}}$$

In some cases, the number of samples provided in the input are not sufficient to achieve the resolution bandwidth that you specify. When this situation occurs, Spectrum Analyzer produces a message on the display, as shown in the following figure.



Spectrum Analyzer removes this message and displays a spectral estimate as soon as enough data has been input. Notice that this behavior differs from the Spectrum Scope block in versions R2012b and earlier. If the **Buffer input** check box was selected, the Spectrum Scope block computed a spectral update using the number of samples given by the **Buffer size** parameter. Otherwise, the Spectrum Scope block computed a spectral update using the number of samples in each frame.

2 Spectrum Analyzer calculates and plots the power spectrum, power spectrum density, and RMS computed by the modified *Periodogram* estimator. For more information about the Periodogram method, see `periodogram` in the Signal Processing Toolbox documentation.

Power Spectral Density — The power spectral density (PSD) is given by the following equation.

$$PSD(f) = \frac{1}{P}\sum_{p=1}^{P}\frac{\left|\sum_{n=1}^{N_{FFT}} x^{(p)}[n]e^{-j2\pi f(n-1)T}\right|^2}{F_s \times \sum_{n=1}^{N_{window}} w^2[n]}$$

In this equation, $x[n]$ is the discrete input signal. On every input signal frame, Spectrum Analyzer generates as many overlapping windows as possible, each window denoted as $x^{(p)}[n]$, and computes their periodograms. Spectrum Analyzer displays a running average of the P most current periodograms.

Power Spectrum — The power spectrum is the product of the power spectral density and the resolution bandwidth, as given by the following equation.

$$P_{spectrum}(f) = PSD(f) \times RBW = PSD(f) \times \frac{F_s \times NENBW}{N_{window}} = \frac{1}{P}\sum_{p=1}^{P}\frac{\left|\sum_{n=1}^{N_{FFT}} x^{(p)}[n]e^{-j2\pi f(n-1)T}\right|^2}{\left[\sum_{n=1}^{N_{window}} w[n]\right]^2}$$

Spectrogram — You can plot any power as a spectrogram. Each line of the spectrogram is one periodogram. The time resolution of each line is 1/RBW, which is the minimum attainable resolution. Achieving the resolution you want may require combining several periodograms may be combined. You then use interpolation to calculate noninteger values of 1/RBW. In the spectrogram display, time scrolls from bottom to top, so the most recent data is shown at the bottom of the display. The offset shows the time value at which the center of the most current spectrogram line occurred.

Note: The number of FFT points (N_{fft}) is independent of the window length (N_{window}). You can set them to different values provided that N_{fft} is greater than or equal to N_{window}.

The *Occupied BW* is calculated as follows.

- Calculate the total power in the measured frequency range.
- Determine the lower frequency value. Starting at the lowest frequency in the range and moving upward, the power distributed in each frequency is summed until this sum is $\frac{100 \text{ - } Occupied\ BW\ \%}{2}$ of the total power.
- Determine the upper frequency value. Starting at the highest frequency in the range and moving downward, the power distributed in each frequency is summed until it reaches $\frac{100 \text{ - } Occupied\ BW\ \%}{2}$ of the total power.
- The bandwidth between the lower and upper power frequency values is the occupied bandwidth.
- The frequency halfway between the lower and upper frequency values is the center frequency.

The *Distortion Measurements* are computed as follows.

1. Spectral content is estimated by finding peaks in the spectrum. When the algorithm detects a peak, it ignores all adjacent content that decreases monotonically from the peak. After recording the width of the peak, it clears all monotonically decreasing values (that is, it treats all of these values as if they belong to the peak). Using this method, all spectral content centered at DC (0 Hz) is removed from the spectrum and the amount of bandwidth cleared (W_0) is recorded.
2. The fundamental power (P_1) is determined from the remaining maximum value of the displayed spectrum. A local estimate (Fe_1) of the fundamental frequency is made by computing the central moment of the power in the vicinity of the peak. The bandwidth of the fundamental power content (W_1) is recorded. Then, the power associated from the fundamental is removed as in step 1.
3. The power and width of the second, and higher order harmonics (P_2, W_2, P_3, W_3, etc.) are determined in succession by examining the frequencies closest to the appropriate multiple of the local estimate (Fe_1). Any spectral content that decreases in a monotonically about the harmonic frequency is removed from the spectrum first before proceeding to the next harmonic.

4 Once the DC, fundamental, and harmonic content is removed from the spectrum, the power of the remaining spectrum is examined for its sum ($P_{remaining}$) peak value ($P_{maxspur}$), and its median value ($P_{estnoise}$).

5 The sum of all the removed bandwidth is computed as $W_{sum} = W_0+W_1+W_2+...+W_n$.

The sum of powers of the second and higher order harmonics are computed as $P_{harmonic} = P_2+P_3+P_4+...+P_n$.

6 The sum of the noise power is then estimated as $P_{noise} = (P_{remaining}$*dF + $P_{estnoise}*W_{sum})/RBW$, where dF is the absolute difference between frequency bins, and RBW is the resolution bandwidth of the window.

7 The metrics for SNR, THD, SINAD, and SFDR are then computed from the estimates.

$$THD = 10 \cdot \log_{10}\left(\frac{P_{harmonic}}{P_1}\right)$$

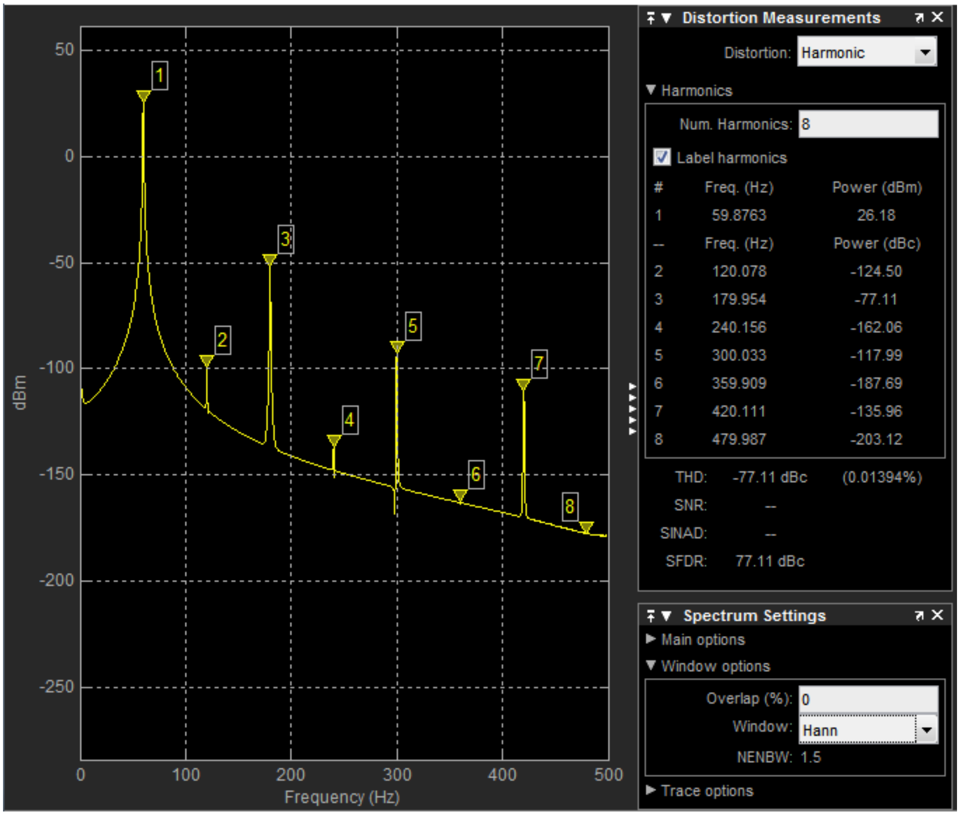
$$SINAD = 10 \cdot \log_{10}\left(\frac{P_1}{P_{harmonic} + P_{noise}}\right)$$

$$SNR = 10 \cdot \log_{10}\left(\frac{P_1}{P_{noise}}\right)$$

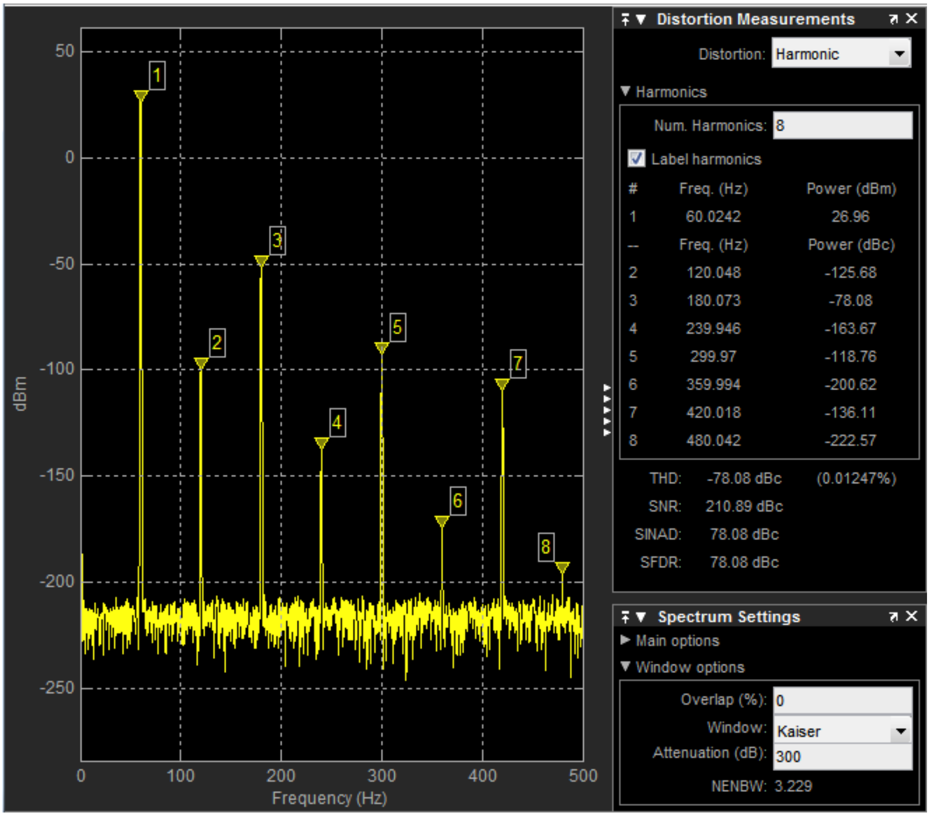
$$SFDR = 10 \cdot \log_{10}\left(\frac{P_1}{\max\left(P_{maxspur}, \max\left(P_2, P_3, \ldots, P_n\right)\right)}\right)$$

The following considerations apply to *Distortion Measurements*.

- The harmonic distortion measurements use the spectrum trace shown in the display as the input to the measurements. The default `Hann` window setting of the Spectrum Analyzer may exhibit leakage that can completely mask the noise floor of the measured signal.



The harmonic measurements attempt to correct for leakage by ignoring all frequency content that decreases monotonically away from the maximum of harmonic peaks. If the window leakage covers more than 70% of the frequency bandwidth in your spectrum, you may see a blank reading (–) reported for **SNR** and **SINAD**. Consider using a Kaiser window with a high attenuation (up to 330dB) to minimize spectral leakage if your application can tolerate the increased equivalent noise bandwidth (ENBW) of the Kaiser window.



- The DC component is ignored.
- After windowing, the width of each harmonic component masks the noise power in the neighborhood of the fundamental frequency and harmonics. To estimate the noise power in each region, Spectrum Analyzer computes the median noise level in the nonharmonic areas of the spectrum. It then extrapolates that value into each region.
- *N*th order intermodulation products occur at

  $A*F1 + B*F2$

  where $F1$ and $F2$ are the sinusoid input frequencies and $|A| + |B| = N$. A and B are integer values.

- For intermodulation measurements, the third-order intercept (TOI) point is computed as follows, where P is power in decibels of the measured power referenced to one milliwatt (dBm).:
  - $TOI_{lower} = P_{F1} + (P_{F2} - P_{(2F1\text{-}F2)})/2$
  - $TOI_{upper} = P_{F2} + (P_{F1} - P_{(2F2\text{-}F1)})/2$
  - $TOI \ = + (TOI_{lower} + TOI_{upper})/2$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supports MEX code generation by treating the calls to the object as extrinsic. Does not support code generation for standalone applications.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`Spectrum Analyzer` | dsp.TimeScope | dsp.LogicAnalyzer | dsp.ArrayPlot

Introduced in R2012b

getSpectralMaskStatus

System object: dsp.SpectrumAnalyzer
Package: dsp

Get test results of current spectral mask

Syntax

```
results = getSpectralMaskStatus(scope)
```

Description

`results = getSpectralMaskStatus(scope)` returns the current status of the spectral mask on the spectrum analyzer, `scope`, in a structure, `results`.

`results` has the following information.

Field	Description
`IsCurrentlyPassing`	Indicator of whether one or more masks are currently passing `1` — All masks are passing `0` — One or more masks are failing
`NumPassedTests`	Number of mask tests that have passed
`NumTotalTests`	Total number of mask tests
`SuccessRate`	Percentage of tests that have passed
`FailingChannels`	Array of channel numbers that are currently failing the mask test
`FailingMasks`	Character array of which masks are currently failing: `'None'`, `'Upper'`, `'Lower'`, or `'Upper and lower'`
`SimulationTime`	Simulation time

See Also

See Also

dsp.SpectrumAnalyzer | spbscopes.SpectrumAnalyzerConfiguration

hide

System object: dsp.SpectrumAnalyzer
Package: dsp

Hide Spectrum Analyzer window

Syntax

```
hide(scope)
```

Description

`hide(scope)` hides the Spectrum Analyzer window, associated with System object, `scope`.

See Also

dsp.SpectrumAnalyzer.show | dsp.SpectrumAnalyzer

reset

System object: dsp.SpectrumAnalyzer
Package: dsp

Reset internal states of Spectrum Analyzer object

Syntax

`reset(scope)`

Description

`reset(scope)` sets the internal states of the `SpectrumAnalyzer` object `scope` to their initial values.

You should call the `reset` method after calling the `step` method when you want to clear the Spectrum Analyzer figure displays, prior to releasing system resources. This action enables you to start a simulation from the beginning. When you call the `reset` method, the displays will become blank again. In this sense, its functionality is similar to that of the MATLAB `clf` function. Do not call the `reset` method after calling the `release` method.

Algorithms

In operation, the `reset` method is similar to a consecutive execution of the `mdlTerminate` function and the `mdlInitializeConditions` function.

See Also

dsp.SpectrumAnalyzer

show

System object: dsp.SpectrumAnalyzer
Package: dsp

Make Spectrum Analyzer window visible

Syntax

```
show(scope)
```

Description

`show(scope)` makes the Spectrum Analyzer window, associated with System object, `scope`, visible.

See Also

dsp.SpectrumAnalyzer.hide | dsp.SpectrumAnalyzer

step

System object: dsp.SpectrumAnalyzer
Package: dsp

Update spectrum in Spectrum Analyzer figure

Syntax

`step(scope,signal)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`step(scope,signal)` updates the spectrum of the signal, `signal`, in the Spectrum Analyzer figure.

dsp.SpectrumEstimator System object

Package: dsp

Estimate power spectrum or power density spectrum

Description

The `dsp.SpectrumEstimator` computes the power spectrum or the power density spectrum of a signal, using the Welch algorithm and the filter bank approach.

When you choose the Welch method, the object computes the averaged modified periodograms to compute the spectral estimate. When you choose the filter bank approach, an analysis filter bank splits the broadband input signal into multiple narrow subbands. The object computes the power in each narrow frequency band and the computed value is the spectral estimate over the respective frequency band. The filter bank approach produces a spectral estimate with a higher resolution, a more accurate noise floor, and more precise peaks than the Welch method, with low or no spectral leakage. They come at the expense of increased computation and slower tracking.

The spectrum can be expressed in Watts or in decibels. This object can also estimate the max-hold and min-hold spectra of the signal.

To estimate the power density spectrum:

1. Define and set up your spectrum estimator object. See “Construction” on page 4-1647.
2. Call step to estimate the power density spectrum according to the properties of the object.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`SE = dsp.SpectrumEstimator` returns a System object, `SE`, that computes the frequency power spectrum or the power density spectrum of real or complex signals. This System object uses the Welch's averaged modified periodogram method and filter bank based spectral estimation method.

`SE = dsp.SpectrumEstimator('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a `Spectrum Estimator` System object, `SE`, with each specified property name set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1,Value1`, ..., `NameN,ValueN`).

Properties

`SampleRate`

Sample rate of input

Specify the sample rate of the input in, hertz, as a finite numeric scalar. The default value is `1 Hz`. The sample rate is the rate at which the signal is sampled in time.

`SpectrumType`

Spectrum type

Specify the spectrum type as one of `'Power' | 'Power density'`. When the spectrum type is `'Power'`, the power density spectrum is scaled by the equivalent noise bandwidth of the window (in Hz). The default value is `'Power'`. This property is tunable.

`SpectralAverages`

Number of spectral averages

Specify the number of spectral averages as a positive, integer scalar. The Spectrum Estimator computes the current power spectrum or power density spectrum estimate by averaging the last `N` estimates. `N` is the number of spectral averages defined in the `SpectralAverages` property. The default value is `8`.

`FFTLengthSource`

Source of the FFT length value

Specify the source of the FFT length value as one of `'Auto'` | `'Property'`. The default value is `'Auto'`. If you set this property to `'Auto'`, the Spectrum Estimator sets the FFT length to the input frame size. If you set this property to `'Property'`, then you specify the number of FFT points using the `FFTLength` property.

FFTLength

FFT Length

Specify the length of the FFT that the Spectrum Estimator uses to compute spectral estimates as a positive, integer scalar. This property applies when you set the `FFTLengthSource` property to `'Property'`. The default value is `128`.

Method

Welch or filter bank

Specify the spectral estimation method as one of `'Welch'` | `'Filter bank'`.

- `'Welch'` — The object uses Welch's averaged modified periodograms method.
- `'Filter bank'` — An analysis filter bank splits the broadband input signal into multiple narrow subbands. The object computes the power in each narrow frequency band and the computed value is the spectral estimate over the respective frequency band.

  The default is `'Welch'`.

NumTapsPerBand

Number of filter taps per frequency band

Specify the number of filter coefficients, or taps, for each frequency band. This value corresponds to the number of filter coefficients per polyphase branch. The total number of filter coefficients is given by `NumTapsPerBand` × `FFTLength`. This property applies when you set `Method` to `'Filter bank'`. The default is 12.

Window

Window function

Specify a window function for the spectral estimator as one of `'Rectangular'` | `'Chebyshev'` | `'Flat Top'` | `'Hamming'` | `'Hann'` | `'Kaiser'`. The default value is `'Hann'`.

SidelobeAttenuation

Side lobe attenuation of window

Specify the side lobe attenuation of the window as a real, positive scalar, in decibels (dB). This property applies when you set the `Window` property to `'Chebyshev'` or `'Kaiser'`. The default value is `60 dB`.

FrequencyRange

Frequency range of the spectrum estimate

Specify the frequency range of the spectrum estimator as one of `'twosided' | 'onesided' | 'centered'`.

If you set the `FrequencyRange` to `'onesided'`, then the Spectrum Estimator computes the one-sided spectrum of a real input signal. When the FFT length, `NFFT`, is even, the spectrum estimate has length (`NFFT`/2) + 1 and is computed over the frequency range `[0 SampleRate/2]`, where `SampleRate` is the sample rate of the input signal. When `NFFT` is odd, the spectrum estimate has length (`NFFT` + 1)/2 and is computed over the frequency range [`0 SampleRate/2`).

If you set the `FrequencyRange` to `'twosided'`, then the Spectrum Estimator computes the two-sided spectrum of a complex or real input signal. The length of the spectrum estimate is equal to the FFT length. The spectrum estimate is computed over the frequency range [`0 SampleRate`), where `SampleRate` is the sample rate of the input signal.

If you set the `FrequencyRange` to `'centered'`, then the Spectrum Estimator computes the centered two-sided spectrum of a complex or real input signal. The length of the spectrum estimate is equal to the FFT length. The spectrum estimate is computed over the frequency range (`-SampleRate/2 SampleRate/2`] when the FFT length is even and `(-SampleRate/2 SampleRate/2)` when the FFT length is odd.

The default value is `'twosided'`.

PowerUnits

Power units

Specify the units used to measure power as one of `'Watts' | 'dBW' | 'dBm'`. The default value is `'Watts'`.

`ReferenceLoad`

Reference load

Specify the load that the spectrum estimator uses as a reference to compute power values as a real, positive scalar in ohms. The default value is `1 ohm`.

`OutputMaxHoldSpectrum`

Output max-hold spectrum

Set this property to `true` so that the Spectrum Estimator computes and outputs the max-hold spectrum of each input channel. The max-hold spectrum is computed by keeping, at each frequency bin, the maximum value of all the power spectrum estimates. The default is `false`.

`OutputMinHoldSpectrum`

Output min-hold spectrum

Set this property to `true` so that the Spectrum Estimator computes and outputs the min-hold spectrum of each input channel. The min-hold spectrum is computed by keeping, at each frequency bin, the minimum value of all the power spectrum estimates. The default is `false`.

Methods

getFrequencyVector	Get the vector of frequencies at which the spectrum is estimated
getRBW	Get the resolution bandwidth of the spectrum
reset	Reset the internal states of the spectrum estimator
step	Estimate the power spectrum or power-density spectrum of a signal

Common to All System Objects	
`clone`	Create System object with same property values

Common to All System Objects	
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Spectral Estimation Using Filter Bank

Compare spectral estimates of sinusoids embedded in white Gaussian noise using a Hann window based Welch method and filter bank method.

Initialization

Initialize two `dsp.SpectrumEstimator` objects. Specify one estimator to use the Welch-based spectral estimation technique with a Hann window. Specify the other estimator to use an analysis filter bank to perform the spectral estimation. Specify a noisy sine wave input signal with 4 sinusoids at 0.16, 0.2, 0.205, and 0.25 cycles/sample. View the spectral estimate using an array plot.

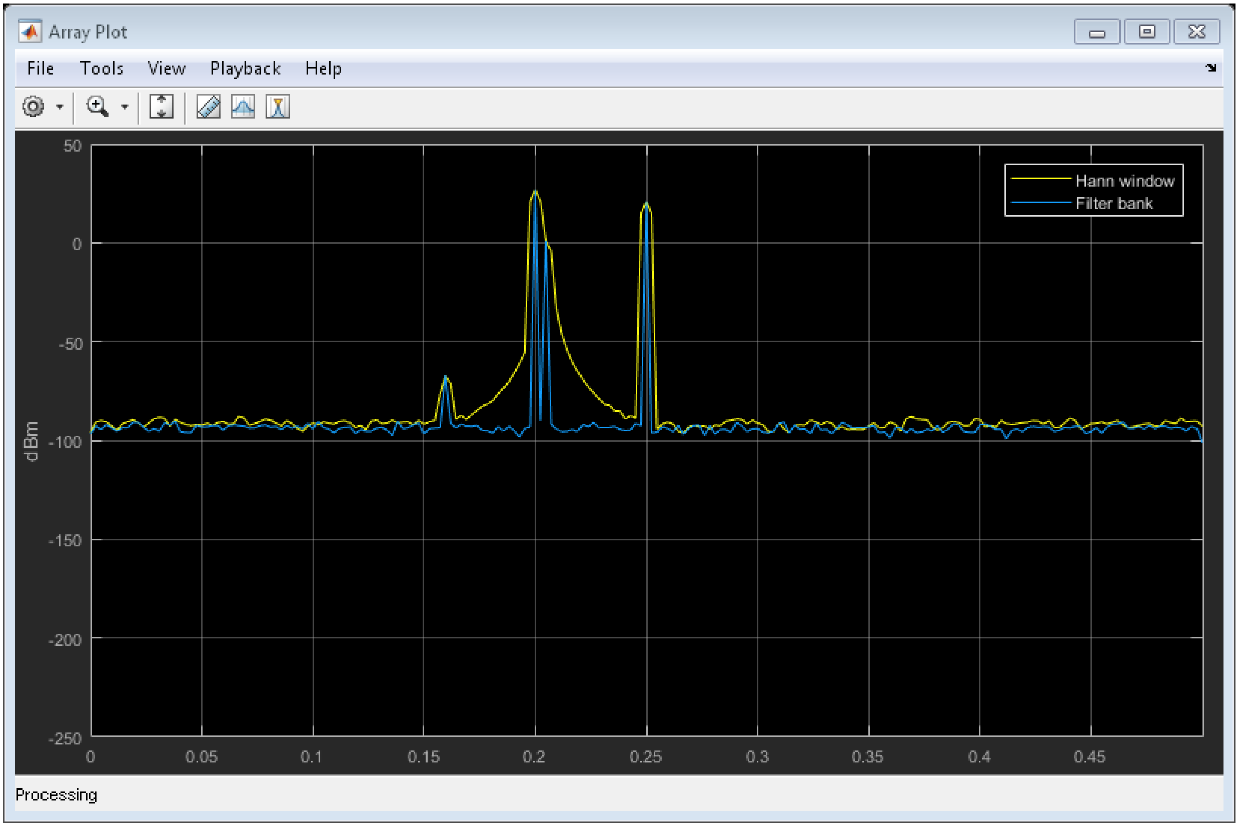
```
FrameSize = 420;
Fs = 1;
sinegen = dsp.SineWave('SampleRate',Fs,...
    'SamplesPerFrame',FrameSize,...
    'Frequency',[0.16 0.2 0.205 0.25],...
    'Amplitude',[2e-5 1  0.05  0.5]);
NoiseVar = 1e-10;
numAvgs = 8;

hannEstimator = dsp.SpectrumEstimator('PowerUnits','dBm',...
    'Window','Hann','FrequencyRange','onesided',...
    'SpectralAverages',numAvgs,'SampleRate',Fs);

filterBankEstimator = dsp.SpectrumEstimator('PowerUnits','dBm',...
    'Method','Filter bank','FrequencyRange','onesided',...
    'SpectralAverages',numAvgs,'SampleRate',Fs);

spectrumPlotter = dsp.ArrayPlot(...
```

```
    'PlotType','Line','SampleIncrement',Fs/FrameSize,...
    'YLimits',[-250,50],'YLabel','dBm',...
    'ShowLegend',true,'ChannelNames',{'Hann window','Filter bank'});
```

Streaming

Stream the input. Compare the spectral estimates computed using the Hann window and the analysis filter bank

```
for i = 1:1000
    x = sum(sinegen(),2) + sqrt(NoiseVar)*randn(FrameSize,1);
    Pse_hann = hannEstimator(x);
    Pfb = filterBankEstimator(x);
    spectrumPlotter([Pse_hann,Pfb]);
end
```



The Hann window misses the peak at 0.205 cycles/sample. In addition, the window has a significant spectral leakage that makes the peak at 0.16 cycles/sample hard to distinguish, and the noise floor is not correct.

The filter bank estimate has a very good resolution with no spectral leakage.

Power and Max-Hold Spectra of Noisy Sine Wave

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Generate a sine wave.

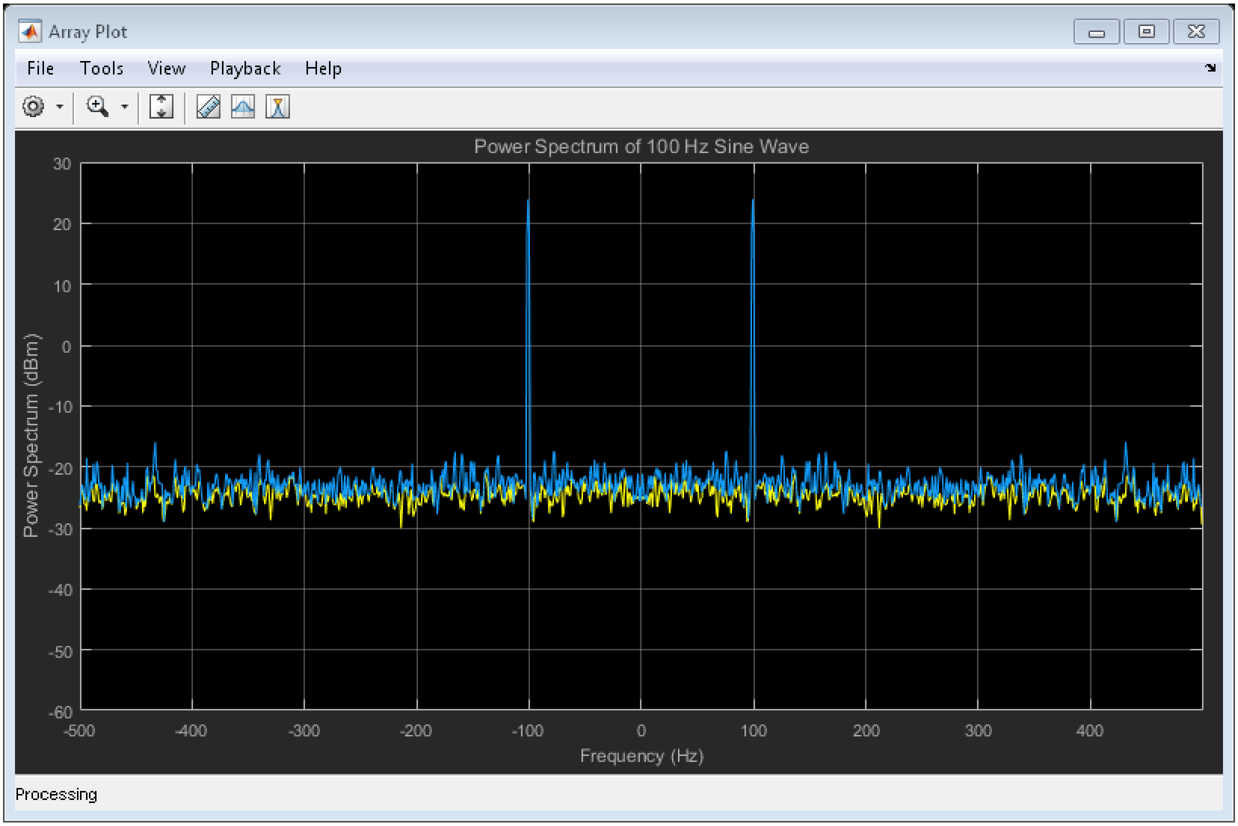
```
SINE = dsp.SineWave('Frequency',100,'SampleRate',1000, ...
    'SamplesPerFrame',1000);
```

Use the Spectrum Estimator to compute the power spectrum and the max-hold spectrum of the sine wave. Use the Array Plot to display the spectra.

```
SE = dsp.SpectrumEstimator('SampleRate',SINE.SampleRate, ...
    'SpectrumType','Power','PowerUnits','dBm', ...
    'FrequencyRange','centered','OutputMaxHoldSpectrum',true);
PLOTTER = dsp.ArrayPlot('PlotType','Line','XOffset',-500, ...
    'YLimits',[-60 30], 'Title','Power Spectrum of 100 Hz Sine Wave', ...
    'YLabel','Power Spectrum (dBm)','XLabel','Frequency (Hz)');
```

Add random noise to the sine wave. Stream in the data, and plot the power spectrum of the signal.

```
for ii = 1:10
    x = SINE() + 0.05*randn(1000,1);
    [Pxx,Pmax] = SE(x);
    PLOTTER([Pxx Pmax]);
end
```



Algorithms

Welch's Method of Averaged Modified Periodograms

When you choose the Welch method, the power spectrum estimate is averaged modified periodograms.

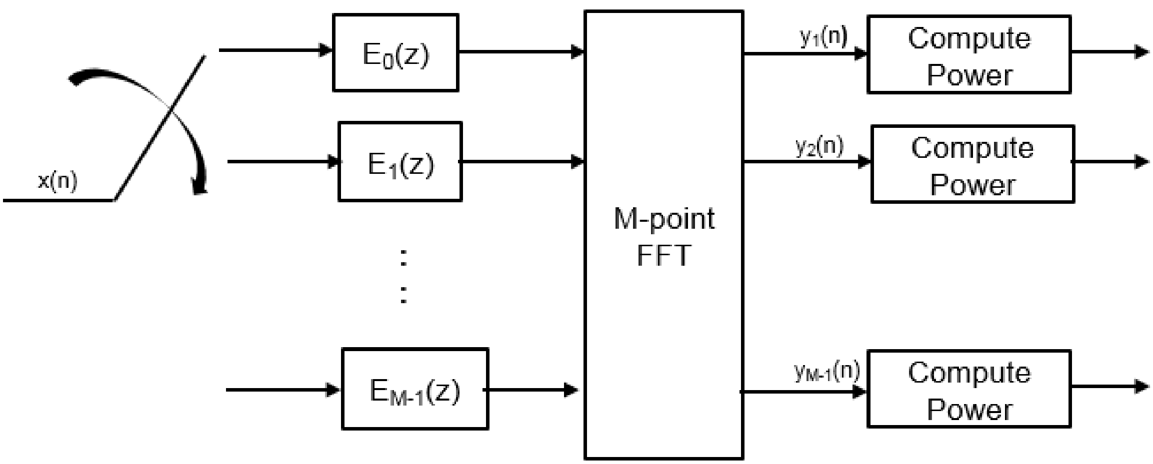
Let `x` be the input frame. Multiply `x` by the window and scale the result by the window power. Take the FFT of the signal, `Y`, and take the square magnitude using `Z = Y*conj(Y)`. Compute the current power spectrum estimate by averaging the last N `Z`s and scaling the answer by the sample rate. N is the number of spectral averages.

For further information refer to the “Algorithms” on page 2-1592 section in Spectrum Analyzer, which uses the same algorithm.

Filter Bank

The spectrum estimator uses an analysis filter bank to estimate the power spectrum. The number of taps per frequency band specifies the number of filter coefficients for each frequency band of the filter bank. The total number of filter coefficients is equal to number of taps per band times the FFT length.

The filter-bank-based spectrum estimator splits the broadband input signal, *x(n)*, into multiple narrow bands and computes the power in each band. Each value becomes the estimate of the power over that narrow frequency band. The power in all the narrow bands forms the spectral estimate vector.



For more details on the implementation, see the “Algorithm” on page 2-214 section in `dsp.Channelizer`.

References

[1] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling.* Hoboken, NJ: John Wiley & Sons, 1996

[2] Kay, Steven M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1999

[3] Stoica, Petre and Randolph L. Moses. *Spectral Analysis of Signals*. Upper Saddle River, NJ: Prentice Hall, 2005

[4] Welch, P. D. “The use of fast Fourier transforms for the estimation of power spectra: A method based on time averaging over short modified periodograms,” *IEEE Transactions on Audio and Electroacoustics*, Vol. 15, 1967, pp. 70–73.

[5] Harris, F.J. *Multirate Signal Processing for Communication Systems*. Prentice Hall. 2004, pp. 208–209.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- When the FFT length is not a power of two, the executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- When the FFT length is a power of two, you can generate standalone C and C++ code from this System object.

See Also

See Also

System Objects
dsp.CrossSpectrumEstimator | dsp.SpectrumAnalyzer | dsp.TransferFunctionEstimator

Blocks
`Spectrum Estimator`

Introduced in R2013b

getFrequencyVector

System object: dsp.SpectrumEstimator
Package: dsp

Get the vector of frequencies at which the spectrum is estimated

Syntax

```
getFrequencyVector(SE)
```

Description

`getFrequencyVector(SE)` returns the vector of frequencies at which the spectrum is estimated.

If you set the `FrequencyRange` to `'onesided'` and the FFT length, `NFFT`, is even, the frequency vector is of length `NFFT/2+1`, and covers the interval `[0,SampleRate/2]`. If you set the `FrequencyRange` to `'onesided'` and `NFFT` is odd, the frequency vector is of length `(NFFT+1)/2` and covers the interval `[0,SampleRate/2]`. If you set the `FrequencyRange` to `'twosided'`, the frequency vector is of length `NFFT` and covers the interval `[0, SampleRate]`. If you set the `FrequencyRange` to `'centered'`, the frequency vector is of length `NFFT` and covers the range `[-SampleRate/2, SampleRate/2]` and `[-SampleRate/2, SampleRate/2]` for even and odd length `NFFT`, respectively.

getRBW

System object: dsp.SpectrumEstimator
Package: dsp

Get the resolution bandwidth of the spectrum

Syntax

`getRBW(SE)`

Description

`getRBW(SE)` returns the resolution bandwidth of the spectrum.

The resolution bandwidth, `RBW`, is the smallest positive frequency, or frequency interval, that can be resolved. It is equal to `ENBW*SampleRate/ L`, where `L` is the input length, and `ENBW` is the two-sided equivalent noise bandwidth of the window (in Hz). For example, if `SampleRate=100`, `L=1024`, and `Window='Hann'`, `RBW=enbw(hann(1024))*100/1024`.

reset

System object: dsp.SpectrumEstimator
Package: dsp

Reset the internal states of the spectrum estimator

Syntax

```
reset(sysObj)
```

Description

`reset(sysObj)` resets the internal states of the System object, `sysObj`, to their initial values. The reset method is always a no-op for unlocked System objects, as the states may not be allocated when the object is not locked.

step

System object: dsp.SpectrumEstimator
Package: dsp

Estimate the power spectrum or power-density spectrum of a signal

Syntax

```
pxx = step(SE,x)
[pxx,pmax] = step(SE,x)
[pxx,pmin] = step(SE,x)
[pxx,pmax,pmin] = step(SE,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`pxx = step(SE,x)` computes the power spectrum or power-density spectrum, `pxx`, of the input signal, `x`. The System object treats the columns of `x` as independent channels.

`[pxx,pmax] = step(SE,x)` also computes the max-hold frequency spectrum, `pmax`, of `x`. To determine the max-hold spectrum, the method keeps the maximum of all the power spectrum estimates computed at each frequency bin. Set `OutputMaxHoldSpectrum` to `true` to obtain the max-hold spectrum.

`[pxx,pmin] = step(SE,x)` also computes the min-hold frequency spectrum, `pmin`, of `x`. To determine the min-hold spectrum, the method keeps the minimum of all the power spectrum estimates computed at each frequency bin. Set `OutputMinHoldSpectrum` to `true` to obtain the min-hold spectrum.

`[pxx,pmax,pmin] = step(SE,x)` computes the power spectrum or power-density spectrum, the max-hold spectrum, and the min-hold spectrum of `x`. Set

OutputMaxHoldSpectrum and OutputMinHoldSpectrum to true to obtain the max-hold and min-hold spectra.

Input Arguments

SE — Frequency spectrum estimator

SpectrumEstimator System object

Frequency spectrum estimate, specified as a SpectrumEstimator System object

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix with floating point or fixed-point precision. The row length of x is the frame size or channel length. Each column of x is treated as a separate channel. The column length of x is the number of channels.

Output Arguments

pxx — Power or power-density spectrum estimate

vector | matrix

Power or power-density spectrum estimate, returned as a vector or matrix of the same size, data type, and complexity as the input signal, x. By default, the unit is 'Watts'. You can also specify the spectrum to be in 'dBm' or 'dBW' through the PowerUnits property of the SpectrumEstimator System object.

pmax — Max-hold spectrum estimate

vector | matrix

Max-hold spectrum estimate, returned as a vector or matrix of the same size, data type, and complexity as the input signal, x.

pmin — Min-hold spectrum estimate

vector | matrix

Min-hold spectrum estimate, returned as a vector or matrix of the same size, data type, and complexity as the input signal, x.

Examples

Power Spectrum of Multichannel Sinusoidal Signal

Generate a three-channel sinusoid sampled at 1 kHz. Specify sinusoidal frequencies of 100, 200, and 300 Hz. The second and third channels have their phases offset from the first by $\pi/2$ and $\pi/4$, respectively.

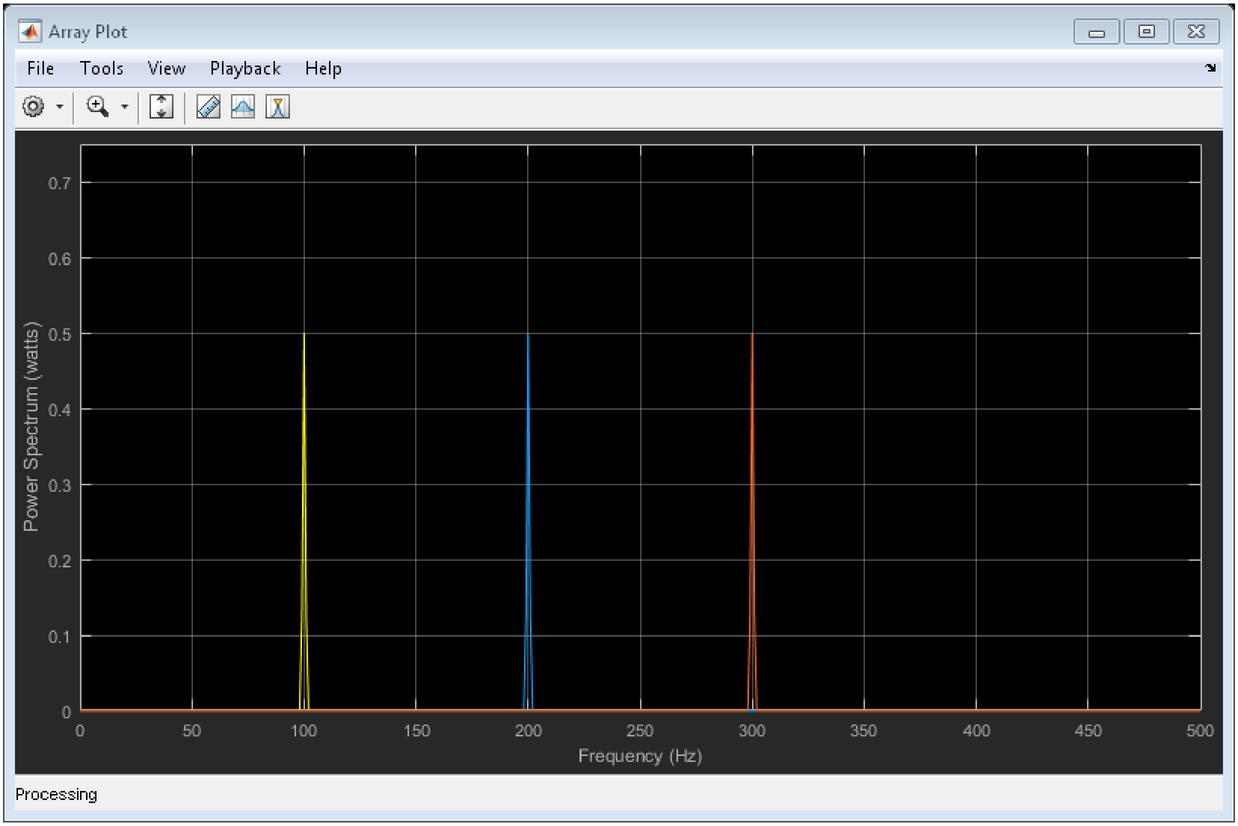
```
SINE = dsp.SineWave('SamplesPerFrame',1000,'SampleRate',1000, ...
    'Frequency',[100 200 300],'PhaseOffset',[0 pi/2 pi/4]);
```

Estimate and plot the one-sided spectrum of the signal. Use the Spectrum Estimator for the computation and the Array Plot for the plotting.

```
SE = dsp.SpectrumEstimator('FrequencyRange','onesided');
PLOTTER = dsp.ArrayPlot('PlotType','Line','YLimits',[0 0.75], ...
    'YLabel','Power Spectrum (watts)','XLabel','Frequency (Hz)');
```

Step through to obtain the data streams and display the spectra of the three channels.

```
y = step(SINE);
pxx = step(SE,y);
step(PLOTTER,pxx)
```


Array Plot
File Tools View Playback Help
Power Spectrum (watts)
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 50 100 150 200 250 300 350 400 450 500
Frequency (Hz)
Processing

dsp.StandardDeviation System object

Package: dsp

Standard deviation of input or sequence of inputs

Description

The `dsp.StandardDeviation` object computes the standard deviation for an input or sequence of inputs.

To compute the standard deviation for an input or sequence of inputs:

1 Define and set up your standard deviation object. See "Construction" on page 4-1665.
2 Call step to compute the standard deviation according to the properties of `dsp.StandardDeviation`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Note: The `Running` mode in the `dsp.StandardDeviation` System object will be removed in a future release. To compute the running standard deviation in MATLAB, use the dsp.MovingStandardDeviation System object instead.

Construction

`std = dsp.StandardDeviation` returns a standard deviation System object, `std`, that computes the standard deviation for the columns of input.

`std = dsp.StandardDeviation('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a standard deviation System object, `std`, with each specified property set to the specified value.

Properties

RunningStandardDeviation

Enable calculation over successive calls to the `step` method

Set this property to `true` to enable the calculation of standard deviation over successive calls to the `step` method. The default is `false`.

ResetInputPort

Enable resetting in running standard deviation mode

Set this property to `true` to enable resetting for the running standard deviation. When the property is set to `true`, you must specify a reset input to the `step` method to reset the running standard deviation. This property applies only when you set the `RunningStandardDeviation` on page 4-1666 property to `true`. The default is `false`.

ResetCondition

Reset condition for running standard deviation mode

Specify event to reset the running standard deviation as one of `| Rising edge | Falling edge | Either edge | Non-zero |`. This property applies only when you set the `ResetInputPort on page 4-1666` property to `true`. The default is `Non-zero`.

Dimension

Dimension to operate along

Specify how the standard deviation calculation is performed over the data as one of `| All | Row | Column | Custom |`. This property applies only when you set the `RunningStandardDeviation on page 4-1666` property to `false`. The default is `Column`.

CustomDimension

Numerical dimension to operate along

Specify the dimension (one-based value) of the input signal, over which the object computes the standard deviation. The cannot exceed the number of dimensions for the input signal. This property applies when you set the `Dimension on page 4-1666` property to `Custom`. The default is `1`.

Methods

reset	Reset states for running standard deviation computation
step	Calculate standard deviation of input

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Running Standard Deviation

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Compute the running standard deviation of a signal using `dsp.StandardDeviation` object. To activate this mode, set the `RunningStandardDeviation` property to `true`.

```
std2 = dsp.StandardDeviation;
std2.RunningStandardDeviation = true;
x = randn(100,1);
y = std2(x);
```

y(i) is the standard deviation of the ith input sample with respect to all the past input samples.

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Standard Deviation` block reference page. The object properties correspond to the block parameters, except:

- The **Reset port** block parameter corresponds to the `ResetInputPort` and `ResetCondition` object properties.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

See Also

System Objects

dsp.MovingStandardDeviation | dsp.MovingRMS | dsp.RMS | dsp.MovingVariance | dsp.Variance | dsp.Minimum

Blocks

`Moving RMS` | `Moving Standard Deviation` | `Moving Variance` | `RMS` | `Standard Deviation` | `Variance`

Introduced in R2012a

reset

System object: dsp.StandardDeviation
Package: dsp

Reset states for running standard deviation computation

Syntax

`reset(std)`

Description

`reset(std)` sets the internal states of the `StandardDeviation` object `std` to their initial values when computing the running standard deviation.

step

System object: dsp.StandardDeviation
Package: dsp

Calculate standard deviation of input

Syntax

```
Y = step(std,X)
Y = step(std,X,R)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(std,X)` computes the standard deviation,`Y`, of input `X`. The object computes the standard deviation over successive calls to the `step` method when the `RunningStandardDeviation` property is `true`.

`Y = step(std,X,R)` resets its state based on the value of reset signal `R` and the `ResetCondition` property. You can use this option only when the `RunningStandardDeviation` property is `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.StateLevels System object

Package: dsp

State-level estimation for bilevel rectangular waveform

Description

The `StateLevels` object estimates the state levels of a bilevel rectangular waveform.

To estimate the state levels of a bilevel waveform:

1. Define and set up your state-level estimation. See "Construction" on page 4-1671.
2. Call step to estimate the state levels for an input vector according to the properties of `dsp.StateLevels`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`sl = dsp.StateLevels` creates a state-level estimation System object, `sl`, that estimates state levels in a bilevel rectangular waveform using the histogram method with 100 bins.

`sl = dsp.StateLevels('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an `StateLevels` System object, `sl`, with each specified property set to the specified value.

Properties

`HistogramBounds`

Minimum and maximum levels of the histogram. Specify the range of the histogram as a 2-element real–valued row vector. Signal values outside the range defined by this property are ignored. This property applies when you set the `Method` property to`'Histogram mode'` or `'Histogram mean'`, and either `RunningStateLevels` is `true`, or the `HistogramBoundsSource` property is set to `'Property'`.

Default: `[0 5]`

HistogramBoundsSource

Source of histogram bounds. Specify how to determine the histogram bounds as one of `'Auto'` or `'Property'`. When you set this property to `'Auto'`, the histogram bounds are determined by the minimum and maximum input values. When you set this property to `'Property'`, the histogram bounds are determined by the value of the `HistogramBounds` property. This property applies when you set the `Method` property to `'Histogram mode'` or `'Histogram mean'`, and the `RunningStateLevels` property is `false`.

Default: `'Auto'`

HistogramNumBins

Number of bins in the histogram. Specify the number of bins in the histogram. This property applies when you set the `Method` property to `'Histogram mode'` or `'Histogram mean'`.

Default: `100`

HistogramOutputPort

Enable histogram output. Set this property to `true` to output the histogram used in the computation of the state levels. This property applies when you set the `Method` property to `'Histogram mode'` or `'Histogram mean'`.

Default: `false`

Method

Algorithm used to compute state levels. Specify the method used to compute state levels as one of `'Histogram mean'`, `'Histogram mode'`, or `'Peak to peak'`.

Default: `'Histogram mode'`

RunningStateLevels

Calculation over successive calls to step. Set this property to `true` to enable computation of the state levels over successive calls to the `step`. Otherwise, compute the state levels of the current input. When you set the `RunningStateLevels` property to `false` and you are using a histogram to compute your state levels, you must set the `HistogramBoundsSource` property to `'Property'`.

State

A particular level, which can be associated with an upper and lower state boundary. States are ordered from the most negative to the most positive. In a bilevel waveform, the most negative state is the low state. The most positive state is the high state.

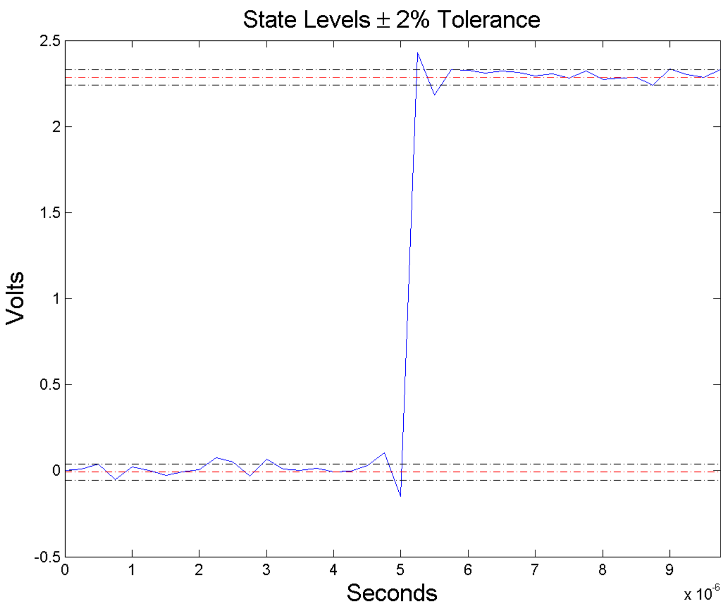
State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state. To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the α% tolerance region for the low state is defined as

$$S_1 \pm \frac{\alpha}{100}(S_2 - S_1)$$

where S_1 is the low-state level and S_2 is the high-state level. Replace the first term in the equation with S_2 to obtain the α% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The estimated state levels are indicated by a dashed red line.



Methods

plot	Plot signal, state levels, and histogram
reset	Reset internal states of state levels object
step	Estimate state levels for bilevel rectangular waveform

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

State Levels of 2.3 V Underdamped Noisy Clock

Compute and plot the state levels of a 2.3 V underdamped noisy clock. Load the clock data in the variable, `x`, and the sampling instants in the variable `t`.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

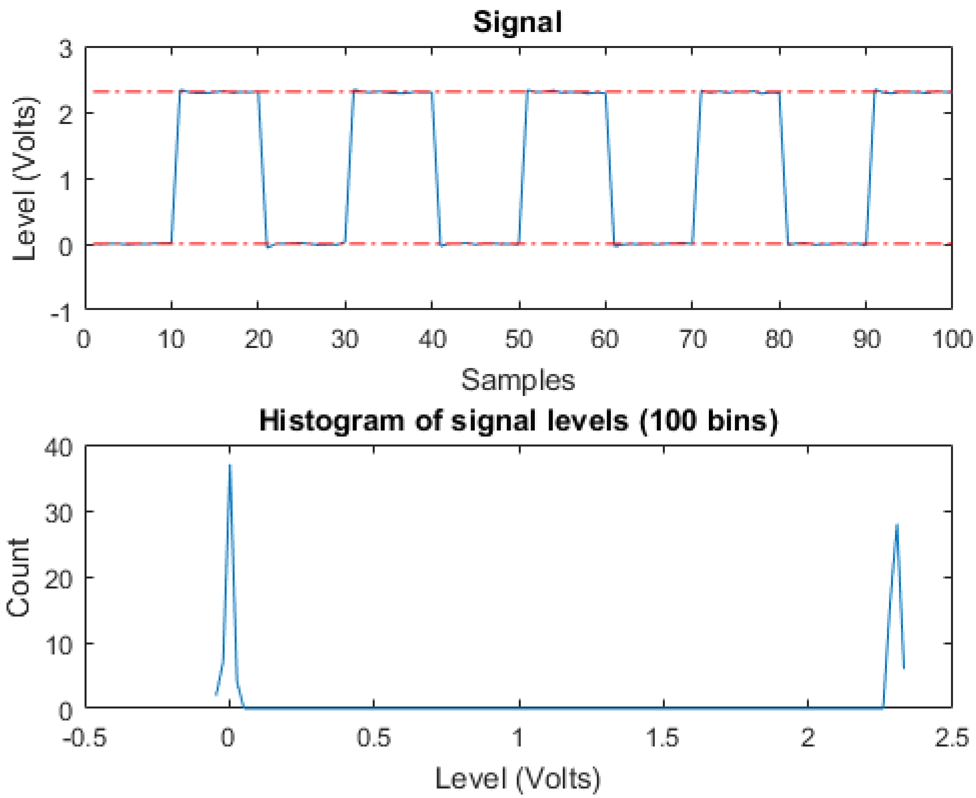
```
load('clockex.mat', 'x', 't');
```

Estimate the state levels.

```
sl = dsp.StateLevels;
levels = sl(x);
```

Plot the clock data along with the estimated state levels and histograms.

```
plot(sl)
```



Algorithms

The `StateLevels` System object uses the histogram method to estimate the states of a bilevel waveform. The histogram method is described in [1]. To summarize the method:

1. Determine the maximum and minimum amplitudes and amplitude range of the data.
2. For the specified number of histogram bins, determine the bin width as the ratio of the amplitude range to the number of bins.
3. Sort the data values into the histogram bins.
4. Identify the lowest-indexed histogram bin, i_{low}, and highest-indexed histogram bin, i_{high}, with nonzero counts.

5 Divide the histogram into two subhistograms. The lower-histogram bins are $i_{low} \leq i \leq 1/2(i_{high} — i_{low})$.

The upper-histogram bins are $i_{low} + 1/2(i_{high} - i_{low}) \leq i \leq i_{high}$.

6 Compute the state levels by determining the mode or mean of the lower and upper histograms.

References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15–17.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

dsp.PulseMetrics | dsp.TransitionMetrics

Introduced in R2012a

plot

System object: dsp.StateLevels
Package: dsp

Plot signal, state levels, and histogram

Syntax

`plot(sl)`

Description

`plot(sl)` plots the signal and state levels computed in the last call to the `step` method. If the `Method` property of the state levels object is set to `'Histogram mode'` or `'Histogram Mean'`, the histogram is plotted in a subplot below the signal.

reset

System object: dsp.StateLevels
Package: dsp

Reset internal states of state levels object

Syntax

`reset(sl)`

Description

`reset(sl)` resets the state levels object, `sl`, to clear information accumulated from previous calls to `step`.

step

System object: dsp.StateLevels
Package: dsp

Estimate state levels for bilevel rectangular waveform

Syntax

```
LEVELS = step(sl,X)
[LEVELS,HISTOGRAM] = step(sl,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`LEVELS = step(sl,X)` returns a 2-element row vector, `LEVELS`, containing the estimated state levels for `X`. `X` is a real-valued column vector.

`[LEVELS,HISTOGRAM] = step(sl,X)` returns a double-precision column vector, `HISTOGRAM`, containing the histogram of the sample values in `X`. You can obtain this output only when you set the `Method` property to either `'Histogram mean'` or `'Histogram mode'`, and you set the `HistogramOutputPort` property to `true`.

dsp.SubbandAnalysisFilter System object

Package: dsp

Decompose signal into high-frequency and low-frequency subbands

Description

The `SubbandAnalysisFilter` object decomposes a signal into high-frequency and low-frequency subbands.

To decompose a signal into high-frequency and low-frequency subbands:

1. Define and set up your two-channel subband analysis filter. See “Construction” on page 4-1681.
2. Call step to decompose the signal according to the properties of `dsp.SubbandAnalysisFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`subAna = dsp.SubbandAnalysisFilter` returns a two-channel subband analysis filter, `subAna`, that decomposes the input signal into a high-frequency subband and a low-frequency subband, each with half the bandwidth of the input.

`subAna = dsp.SubbandAnalysisFilter('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a two-channel subband analysis filter, `subAna`, with each specified property set to the specified value.

`subAna = dsp.SubbandAnalysisFilter(lpc,hpc,` *'PropertyName'*,*PropertyValue*,...) returns a two-channel subband analysis filter, subAna, with the LowpassCoefficients on page 4-1682 property set to lpc, the HighpassCoefficients on page 4-1682 property set to hpc, and other specified properties set to the specified values.

Properties

LowpassCoefficients

Lowpass FIR filter coefficients

Specify a vector of lowpass FIR filter coefficients, in descending powers of z. For the lowpass filter, use a half-band filter that passes the frequency band stopped by the filter specified in the HighpassCoefficients on page 4-1682 property. The default values of this property specify a filter based on a third-order Daubechies wavelet.

HighpassCoefficients

Highpass FIR filter coefficient

Specify a vector of highpass FIR filter coefficients, in descending powers of z. For the highpass filter, use a half-band filter that passes the frequency band stopped by the filter specified in the LowpassCoefficients on page 4-1682 property. The default values of this property specify a filter based on a third-order Daubechies wavelet.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set FullPrecisionOverride to true, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set FullPrecisionOverride to false, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor. This property applies only if the object is not in full precision mode.

OverflowAction

Action to take when integer input is out of range

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap. This property applies only if the object is not in full precision mode.

CoefficientsDataType

Data type of the coefficients

Specify the FIR filter coefficients fixed-point data type as one of | Same word length as input | Custom |. The default is Same word length as input.

CustomCoefficientsDataType

Coefficients word and fraction lengths

Specify the FIR filter coefficients fixed-point type as a numerictype object with a Signedness of Auto. This property applies only when you set the CoefficientsDataType on page 4-1683 property to Custom. The default is numerictype([],16,15).

ProductDataType

Data type of product

Specify the product data type as one of | Full precision | Same as input | Custom |. The default is Full precision.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `ProductDataType` on page 4-1683 property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Data type of accumulator

Specify the accumulator data type as one of `Full precision` | `Same as input` | `Same as product` | `Custom` |. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `AccumulatorDataType` on page 4-1684 property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Data type of output

Specify the output data type as one of | `Same as accumulator` | `Same as product` | `Same as input` | `Custom` |. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` on page 4-1684 property to `Custom`. The default is `numerictype([],16,14)`.

Methods

reset	Reset filter states of subband analysis filter object

step	Decompose signal into high- and low-frequency subbands

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Decompose and Reconstruct a Signal

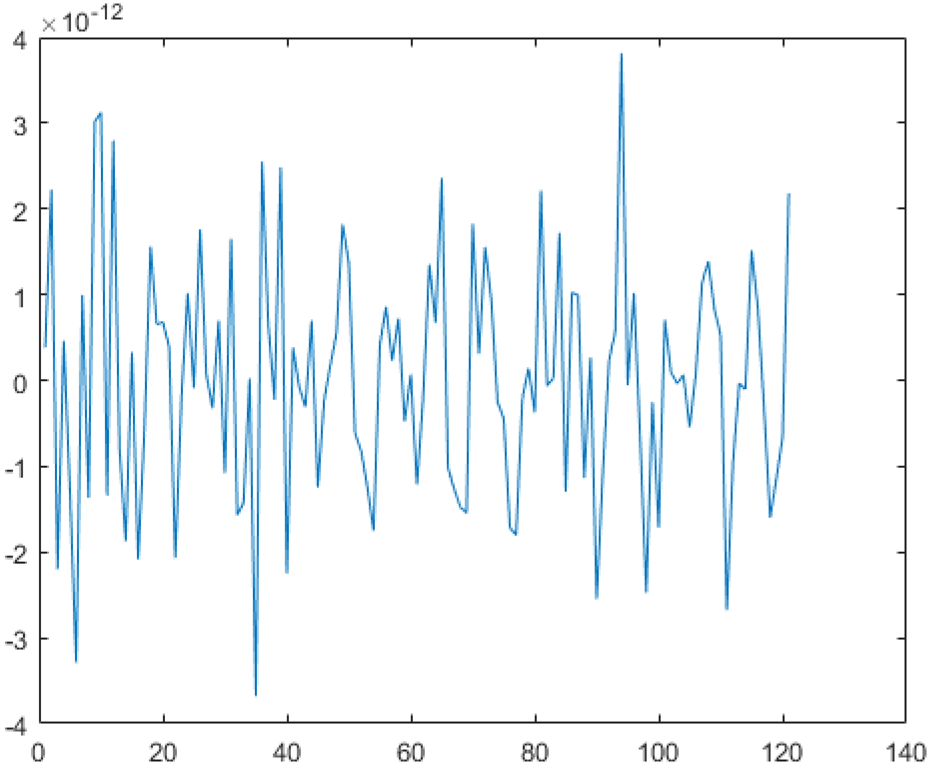
Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Decompose a signal into low frequency and high frequency subbands using the subband analysis filter. Reconstruct the signal using the subband synthesis filter.

```
load dspwlets; % load the filter coefficients lod, hid, lor and hir
subAna = dsp.SubbandAnalysisFilter(lod, hid);
subSynth = dsp.SubbandSynthesisFilter(lor, hir);
u = randn(128,1);

[hi, lo] = subAna(u); % Two channel analysis
y = subSynth(hi, lo); % Two channel synthesis
```

Plot difference between original and reconstructed signals with filter latency compensated.

```
plot(u(1:end-7)-y(8:end));
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Two-Channel Analysis Subband Filter` block reference page. The object properties correspond to the block parameters, except:

- The `SubbandAnalysisFilter` object does not have a property that corresponds to the **Input processing** parameter of the Two-Channel Analysis Subband Filter block. The object assumes the input is frame based and always maintains the input frame rate.

- The **Rate options** block parameter is not supported by the `dsp.SubbandAnalysisFilter` object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.SubbandSynthesisFilter | dsp.DyadicAnalysisFilterBank

Introduced in R2012a

reset

System object: dsp.SubbandAnalysisFilter
Package: dsp

Reset filter states of subband analysis filter object

Syntax

`reset(subAna)`

Description

`reset(subAna)` sets the filter states of the `SubbandAnalysisFilter` object `subAna` to `0`.

step

System object: dsp.SubbandAnalysisFilter
Package: dsp

Decompose signal into high- and low-frequency subbands

Syntax

`[HI,LO] = step(subAna,SIG)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[HI,LO] = step(subAna,SIG)` decomposes the input signal, `SIG`, into a high-frequency subband, `HI`, and a low-frequency subband, `LO`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.SubbandSynthesisFilter System object

Package: dsp

Reconstruct signal from high-frequency and low-frequency subbands

Description

The `SubbandSynthesisFilter` object reconstructs a signal from high-frequency and low-frequency subbands.

To reconstruct a signal from high-frequency and low-frequency subbands:

1. Define and set up your two-channel subband synthesis filter. See "Construction" on page 4-1690.
2. Call step to reconstruct the signal according to the properties of `dsp.SubbandSynthesisFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`subSyn = dsp.SubbandSynthesisFilter` returns a two-channel subband synthesis filter, `subSyn`, that reconstructs a signal from its high-frequency subband and low-frequency subband. Each subband contains half the bandwidth of the original signal.

`subSyn = dsp.SubbandSynthesisFilter('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a two-channel subband synthesis filter, `subSyn`, with each specified property set to the specified value.

`subSyn = dsp.SubbandSynthesisFilter(lpc,hpc,` `'PropertyName',PropertyValue,...)` returns a two-channel subband synthesis filter, `subSyn`. The object has the `LowpassCoefficients` on page 4-1691 property set to `lpc`, the `HighpassCoefficients` on page 4-1691 property set to `hpc`, and other specified properties set to the specified values.

Properties

LowpassCoefficients

Lowpass FIR filter coefficients

Specify a vector of lowpass FIR filter coefficients, in descending powers of z. For the lowpass filter, use a half-band filter that passes the frequency band stopped by the filter specified in the `HighpassCoefficients` on page 4-1691 property. The default values of this property specify a filter based on a third-order Daubechies wavelet.

HighpassCoefficients

Highpass FIR filter coefficients

Specify a vector of highpass FIR filter coefficients, in descending powers of z. For the highpass filter, use a half-band filter that passes the frequency band stopped by the filter specified in the `LowpassCoefficients` on page 4-1691 property. The default values of this property specify a filter based on a third-order Daubechies wavelet.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero` |. The default is `Floor`. This property applies only if the object is not in full precision mode.

OverflowAction

Action to take when integer input is out of range

Specify the overflow action as one of | `Wrap` | `Saturate` |. The default is `Wrap`. This property applies only if the object is not in full precision mode.

CoefficientsDataType

Data type of the coefficients

Specify the FIR filter coefficients fixed-point data type as one of | `Same word length as input` | `Custom` |. The default is `Same word length as input`.

CustomCoefficientsDataType

Coefficient word and fraction lengths

Specify the FIR filter coefficients fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `CoefficientsDataType` on page 4-1692 property to `Custom`. The default is `numerictype([],16,15)`.

ProductDataType

Data type of product

Specify the product data type as one of | `Full precision` | `Same as input` | `Custom` |. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `ProductDataType` on page 4-1692 property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Data type of accumulator

Specify the accumulator data type as one of | `Full precision` | `Same as input` | `Same as product` | `Custom` |. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `AccumulatorDataType` on page 4-1693 property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Data type of output

Specify the output data type as one of | `Same as accumulator` | `Same as product` | `Same as input` | `Custom` |. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` on page 4-1693 property to `Custom`. The default is `numerictype([],16,14)`.

Methods

reset	Reset internal states of subband synthesis filter object

step	Reconstruct signal from high- and low-frequency subbands

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Decompose and Reconstruct a Signal

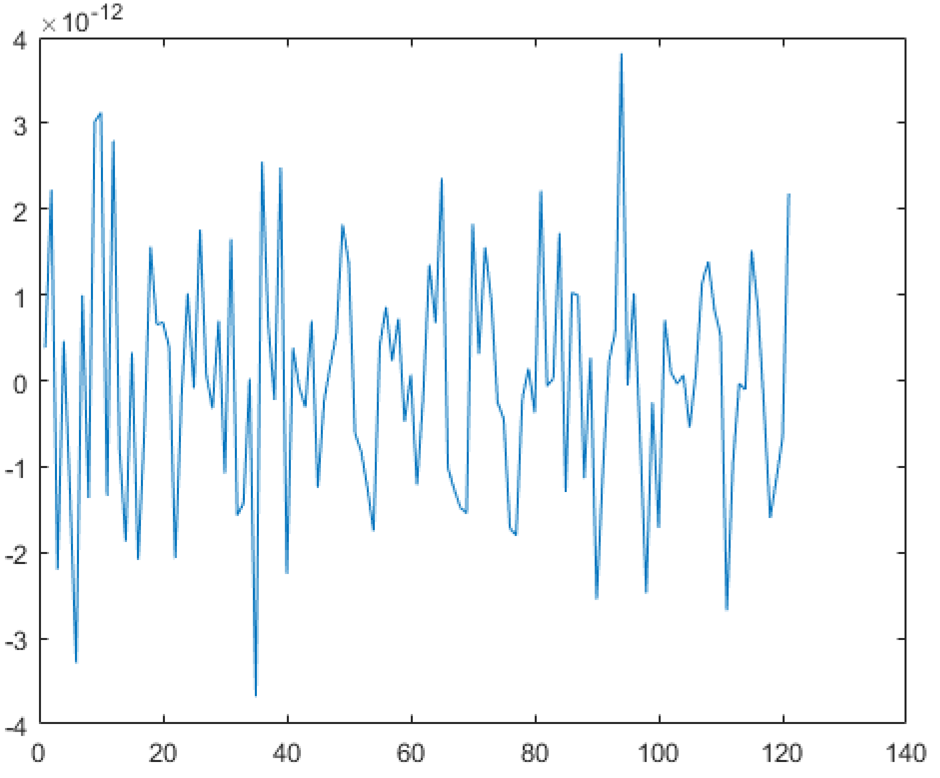
Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Decompose a signal into low frequency and high frequency subbands using the subband analysis filter. Reconstruct the signal using the subband synthesis filter.

```
load dspwlets; % load the filter coefficients lod, hid, lor and hir
subAna = dsp.SubbandAnalysisFilter(lod, hid);
subSynth = dsp.SubbandSynthesisFilter(lor, hir);
u = randn(128,1);

[hi, lo] = subAna(u); % Two channel analysis
y = subSynth(hi, lo); % Two channel synthesis
```

Plot difference between original and reconstructed signals with filter latency compensated.

```
plot(u(1:end-7)-y(8:end));
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Two-Channel Synthesis Subband Filter` block reference page. The object properties correspond to the block parameters, except:

- The `SubbandSynthesisFilter` object does not have a property that corresponds to the **Input processing** parameter of the Two-Channel Synthesis Subband Filter block. The object only performs sample-based processing and always maintains the input frame rate.

- The **Rate options** block parameter is not supported by the `SubbandSynthesisFilter` object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.SubbandAnalysisFilter | dsp.DyadicSynthesisFilterBank

Introduced in R2012a

reset

System object: dsp.SubbandSynthesisFilter
Package: dsp

Reset internal states of subband synthesis filter object

Syntax

`reset(subSyn)`

Description

`reset(subSyn)` sets the internal states of the `SubbandSynthesisFilter` object `subSyn` to their initial values.

step

System object: dsp.SubbandSynthesisFilter
Package: dsp

Reconstruct signal from high- and low-frequency subbands

Syntax

`Y = step(subSyn,HI,LO)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(subSyn,HI,LO)` reconstructs a signal from a high-frequency subband, `HI`, and a low-frequency subband, `LO`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.TimeScope System object

Package: dsp

Time domain signal display

Description

The `TimeScope` object displays time-domain signals.

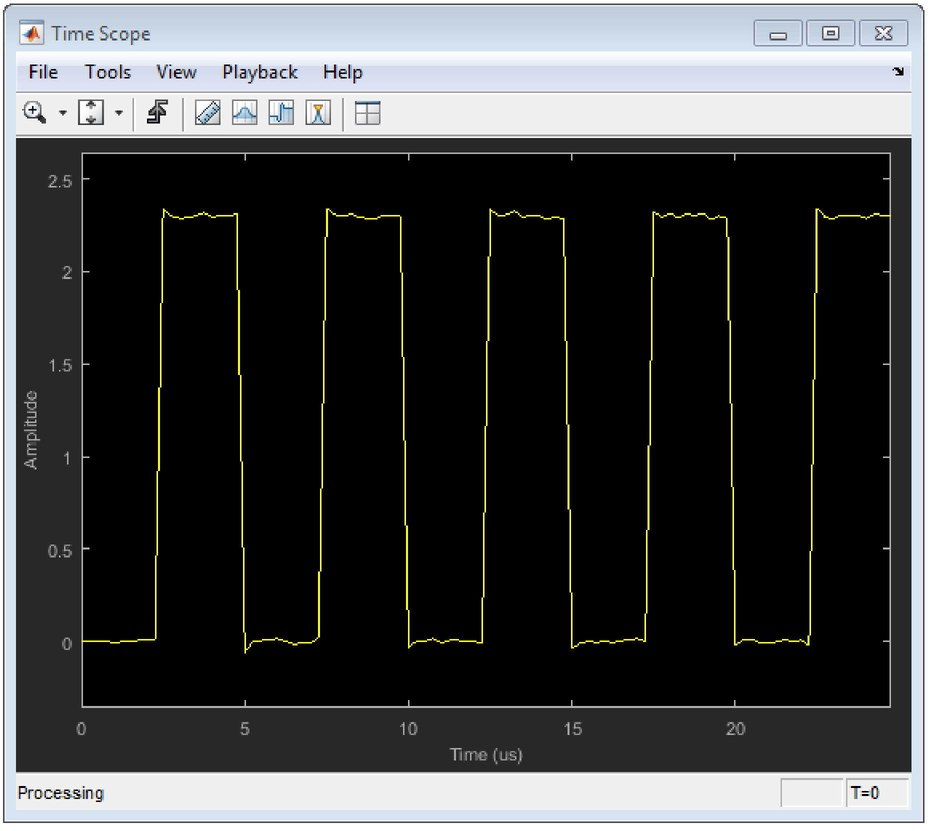
To display time-domain signals in the Time Scope:

1. Define and set up your Time Scope. See "Construction" on page 4-1701.
2. Call step to display the signal in the Time Scope figure. The behavior of `step` is specific to each object in the toolbox.

Use the MATLAB `clear` function to close the Time Scope figure window.

Use the MATLAB `mcc` function to compile code containing a Time Scope.

Note: You cannot open scope configuration dialogs if you have more than one compiled component in your application.



See the following sections for information on the Time Scope System object:

- “Construction” on page 4-1701
- “Properties” on page 4-1702
- “Methods” on page 4-1710

See the following sections for information on the Time Scope Graphical User Interface:

- “Displaying Multiple Signals” on page 4-1710
- “Signal Display” on page 4-1713
- “Measurements Panels” on page 4-1716
- “Style Dialog Box” on page 2-1698

- “Visuals — Time Domain Properties” on page 4-1750
- “Tools — Axes Scaling Properties” on page 4-1758
- “Sources — Streaming Properties” on page 4-1760

For examples on how to use the Time Scope System object, see “Examples” on page 4-1761.

Note: For information about the Time Scope block, see `Time Scope`.

Note: If you own the MATLAB Coder product, you can generate C or C++ code from MATLAB code in which an instance of this system object is created. When you do so, the scope system object is automatically declared as an *extrinsic* variable. In this manner, you are able to see the scope display in the same way that you would see a figure using the `plot` function, without directly generating code from it. For the full list of system objects supporting code generation, see “DSP System Toolbox” (MATLAB Coder).

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`scope = dsp.TimeScope` returns a Time Scope System object, `scope`. This object displays real- and complex-valued floating and fixed-point signals in the time domain.

`scope = dsp.TimeScope('`*`PropertyName`*`',`*`PropertyValue`*`,...)` creates a Time Scope System object, `scope`, with each specified property *`PropertyName`* set to the specified value.

`scope = dsp.TimeScope(NUMINPUTS,SAMPLERATE,'`*`PropertyName`*`',`*`PropertyValue`*`,...)` creates a Time Scope System object, `scope`. This object has the `NumInputPorts` property set to *`NUMINPUTS`*, the `SampleRate` property set to *`SAMPLERATE`*, and other specified properties set to the specified values. *`NUMINPUTS`* and *`SAMPLERATE`* are value-only arguments. You can specify `PropertyName`–`PropertyValue` arguments in any order.

Properties

`ActiveDisplay`

Active display for display-specific properties

Specify the active display as an integer to get and set relevant properties. The number of a display corresponds to its column-wise placement index. Set this property to control which display has its axes colors, line properties, marker properties, and visibility changed. Tunable (Simulink)

Setting this property controls which display is used for `ShowGrid`, `ShowLegend`, `Title`, `PlotAsMagnitudePhase`, `YLabel`, and `YLimits`.

Default: `1`

`AxesScaling`

Enable axes scaling mode

Specify when the scope automatically scales the axes. Valid values are

- `'Auto'` — The scope scales the axes as needed to fit the data, both during and after simulation.
- `'Updates'` — The scope scales the axes after 10 updates.
- `'Manual'` — The scope does not scale the axes automatically.
- `'OnceAtStop'` — The scope scales the axes when the simulation stops.

`BufferLength`

Number of data points in buffer

Specify the size of the buffer that the scope holds in its memory cache. Memory is limited by available memory on your system. If your signal has M rows of data and N data points in each row, M x N is the number of data points per time step. Multiply this result by the number of time steps for your model to obtain the required buffer length. For example, if you have 10 rows of data with each row having 100 data points and your run will be 10 time steps, you should enter 10,000 (which is 10 x 100 x 10) as the buffer length.

Default: 5000

ChannelNames

Names of input signals to display in scope legend

Specify the names of the input signals to display in the scope legend. Enter the names in a cell array of character vectors. If you do not specify any names, they default to `Channel 1`, `Channel 2`, etc.

FrameBasedProcessing

Process input in frames or as samples

When you set this property to `true`, you enable frame-based processing. When you set this property to `false`, you enable sample-based processing.

Default: `true`

LayoutDimensions

Layout grid dimensions

Specify the layout grid dimensions as a 2-element vector: `[numberOfRows, numberOfColumns]`. You can use no more than 16 rows and 16 columns. This property is Tunable (Simulink).

Default: `[1,1]`

MaximizeAxes

Maximize axes control

Specify whether to display the scope in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- `Auto` — In this mode, the axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- `On` — In this mode, the axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.

- `Off` — In this mode, none of the axes appear maximized.

This property is Tunable (Simulink).

Default: `'Auto'`

`Name`

Caption to display on the Time Scope window

Specify as a character vector the caption to display on the scope window. This property is Tunable (Simulink).

Default: `'Time Scope'`

`NumInputPorts`

Number of input signals

Specify the number of input signals to display on the scope as a positive integer. You must invoke the `step` method with the same number of inputs as the value of this property.

This property does not apply to Floating Scopes and Scope Viewers.

Default: `1`

`PlotAsMagnitudePhase`

Plot signal magnitude and phase

When you set this property to `true`, the scope plots the magnitude and phase of the input signal on two separate axes within the same active display. When you set this property to `false`, the scope plots the real and imaginary parts of the input signal on two separate axes within the same active display. This property is particularly useful for complex-valued input signals. Selecting this check box affects the phase for real-valued input signals. When the amplitude of the input signal is nonnegative, the phase is 0 degrees. When the amplitude of the input signal is negative, the phase is 180 degrees. This property is Tunable (Simulink).

When set, `ActiveDisplay` controls which displays are updated. The active display shows the magnitude of the input signal on the top axes and its phase, in degrees, on the bottom axes.

Default: `false`

`PlotType`

Option to control the type of plot

Specify the type of plot to use. The default setting is `Line`.

- `Line` — line graph, similar to the `line` or `plot` function.
- `Stairs` — *stair-step* graph, similar to the `stairs` function. Stair-step graphs are useful for drawing time history graphs of digitally sampled data.

This property is Tunable (Simulink).

Default: `'Line'`

`Position`

Scope or Time Scope window position in pixels

Specify, in pixels, the size and location of the scope window as a 4-element double vector of the form [left bottom width height]. You can place the scope window in a specific position on your screen by modifying the values to this property. This property is Tunable (Simulink).

Default: The default depends on your screen resolution. By default, a scope window appears in the center of your screen with a width of 410 pixels and height of 300 pixels.

`ReduceUpdates`

Reduce updates to improve performance

When you set this property to `true`, the scope logs data for later use and updates the window periodically. When you set this property to `false`, the scope updates every time the `step` method is called. The simulation speed is faster when this property is set to `true`. This property is Tunable (Simulink).

You can also modify this property from the Time Scope GUI. Opening the **Simulation** menu and clearing the **Reduce Updates to Improve Performance** check box is the same as setting this property to `false`.

Default: `true`

SampleRate

Sample rate of inputs

Specify the sample rate, in hertz, of the input signals.

You can specify a scalar or as a numeric vector with length equal to the value of NumInputPorts. The inverse of the sample rate determines the spacing between points on the time axis in the displayed signal. When you set SampleRate to a scalar value and NumInputPorts is greater than 1, the object uses the same sample rate for all inputs.

Default: 1

ShowGrid

Option to enable or disable grid display

When you set this property to true, the grid appears. When you set this property to false, the grid is hidden. This property is Tunable (Simulink).

When set, ActiveDisplay controls which display is updated.

Default: false

ShowLegend

When you set this property to true, the scope displays a legend with input channel names. When you set this property to false, the scope does not display a legend. This property is Tunable (Simulink)

See FrameBasedProcessing on page 4-1703 for information on input channels.

When set, ActiveDisplay controls which display is updated.

Default: false

ShowTimeAxisLabel

When you set this property to true, the scope displays the time-axis label. When you set this property to false, the scope does not display the time-axis label, but still displays tick marks and other time-axis items. This property applies only if the TimeAxisLabels property is All or Bottom. This property is Tunable (Simulink)

`TimeAxisLabels`

Time-axis labels

Specify how time-axis labels should appear in the scope displays as one of `'All'`, `'Bottom`, or `'None'`.

- When you set this property to `'All'`, time-axis labels appear in all displays.
- When you set this property to `'Bottom'`, time-axis labels appear in the bottom display of each column.
- When you set this property to `'None'`, there are no labels in any displays.

This property is Tunable (Simulink).

Default: `'All'`

`TimeDisplayOffset`

Time display offset

Specify the offset, in seconds, to apply to the time axis. This property can be either a numeric scalar or a vector of length equal to the number of input channels. If you specify this property as a scalar, then that value is the time display offset for all channels. If you specify a vector, each vector element is the time offset for the corresponding channel. For vectors with length less than the number of input channels, the time display offsets for the remaining channels are set to 0. If a vector has a length greater than the number of input channels, the extra vector elements are ignored. This property is Tunable (Simulink).

See FrameBasedProcessing on page 4-1703 for information on input channels. See `TimeSpan` on page 4-1707 and `TimeSpanSource` on page 4-1708 for information on the *x*-axis limits and time span settings.

Default: `0`

`TimeSpan`

Time span

Specify the time span, in seconds, as a positive, numeric scalar value. This property applies when `FrameBasedProcessing` is `false`. This property also applies when

FrameBasedProcessing is true and TimeSpanSource is Property. The time-axis limits are calculated as follows.

- Minimum time-axis limit = min(TimeDisplayOffset)
- Maximum time-axis limit = max(TimeDisplayOffset) + TimeSpan

where TimeDisplayOffset and TimeSpan are the values of their respective properties. This property is Tunable (Simulink).

Default: 10

TimeSpanOverrunAction

Wrap or scroll when the TimeSpan value is overrun

Specify how the scope displays new data beyond the visible time span. You can select one of the following options:

- Wrap — In this mode, the scope displays new data until the data reaches the maximum time-axis limit. When the data reaches the maximum time-axis limit of the scope window, the scope clears the display. The scope then updates the time offset value and begins displaying subsequent data points starting from the minimum time-axis limit.
- Scroll — In this mode, the scope scrolls old data to the left to make room for new data on the right side of the scope display. This mode is graphically intensive and can affect run-time performance. However, it is beneficial for debugging and monitoring time-varying signals.

This property is Tunable (Simulink).

Default: 'Wrap'

TimeSpanSource

Source of time span

Specify the source of the time span for frame-based input signals as one of 'Auto' or 'Property'. This property applies when FrameBasedProcessing is set to true. When you set this property to 'Property', the object derives the *x*-axis limits from the TimeDisplayOffset and TimeSpan properties. When you set this property to Auto, the time-axis limits are the following:

- Minimum time-axis limit = min(TimeDisplayOffset)

- Maximum time-axis limit = max(`TimeDisplayOffset`) + max(1/`SampleRate`.**FrameSize*)

where `TimeDisplayOffset` and `SampleRate` are the values of their respective properties. *FrameSize* is a vector equal to the number of rows in each input signal. This property is Tunable (Simulink).

Default: `'Property'`

`TimeUnits`

Units of the time axis

Specify the units used to describe the time axis. You can select one of the following options:

- `Metric` — In this mode, the scope converts the times on the time axis to the most appropriate measurement units. These units include milliseconds, microseconds, nanoseconds, minutes, days, etc. The scope chooses the appropriate measurement units based on the minimum time-axis limit and the maximum time-axis limit of the scope window.
- `Seconds` — In this mode, the scope always displays the units on the time axis as seconds.
- `None` — In this mode, the scope does not display any units on the time axis. The scope only shows the word `Time` on the time axis.

This property is Tunable (Simulink).

Default: `'Metric'`

`Title`

Display title

Specify the display title as a character vector. When set, `ActiveDisplay` controls which display is updated.

Default: `''`

`YLabel`

The label for the *y*-axis

Specify the text for the scope to display to the left of the *y*-axis. Tunable (Simulink)

This property applies only when `PlotAsMagnitudePhase` is `false`. When `PlotAsMagnitudePhase` is `true`, the two *y*-axis labels are read-only values. The *y*-axis labels are set to `'Magnitude'` and `'Phase'` for the magnitude plot and the phase plot, respectively. When set, `ActiveDisplay` controls which display is updated.

Default: `'Amplitude'` if `PlotAsMagnitudePhase` is `false`

YLimits

The limits for the *y*-axis

Specify the *y*-axis limits as a 2-element numeric vector, `[ymin ymax]`. This property is Tunable (Simulink).

When `PlotAsMagnitudePhase` is `true`, this property specifies the *y*-axis limits of only the magnitude plot. The *y*-axis limits of the phase plot are always `[-180, 180]`. When set, `ActiveDisplay` controls which display is updated.

Default: `[-10, 10]`, if `PlotAsMagnitudePhase` is `false`, or `[0, 10]`, if `PlotAsMagnitudePhase` is `true`.

Methods

hide	Hide time scope window
reset	Reset internal states of time scope object
show	Make time scope window visible
step	Display signal in time scope figure

Displaying Multiple Signals

Multiple Signal Input

You can configure the Time Scope to show multiple signals within the same display or on separate displays. By default, the signals appear as different-colored lines on the same display. The signals can have different dimensions, sample rates, and data types. Each signal can be either real or complex valued. You can set the number of input ports on the Time Scope in the following ways:

- Set the `NumInputPorts` on page 4-1704 property. This property is nontunable, so you should set it before you run the step method.
- Run the show method to open the scope window. In the scope menu, select **File** > **Number of Input Ports**.
- Run the show method to open the scope window. In the scope menu, select **View** > **Configuration Properties** and set the **Number of input ports** on the **Main** tab.

An input signal may contain multiple channels, depending on its dimensions. Multiple channels of data always appear as different-colored lines on the same display.

Multiple Signal Names and Colors

By default, if the input signal has multiple channels, the scope uses an index number to identify each channel of that signal. For example, a 2-channel signal would have the following default names in the channel legend: `Channel 1`, `Channel 2`. To show the legend, select **View** > **Configuration Properties**, click the **Display** tab, and select the **Show Legend** check box. If there are a total of 7 input channels, the following legend appears in the display.

By default, the scope has a black axes background and chooses line colors for each channel in a manner similar to the Simulink Scope block. When the scope axes background is black, it assigns each channel of each input signal a line color in the order shown in the above figure.

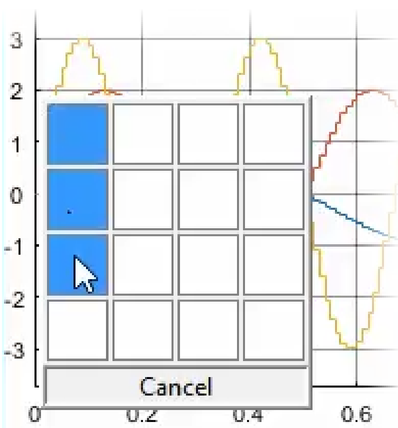
If there are more than 7 channels, then the scope repeats this order to assign line colors to the remaining channels. To choose line colors for each channel, change the axes background color to any color except black. To change the axes background color to white, select **View** > **Style**, click the Axes background color button (), and select white from the color palette. Run the simulation again. The following legend appears in the display. This is the color order when the background is not black.

Multiple Displays

You can display multiple channels of data on different displays in the scope window. In the scope toolbar, select **View** > **Layout**, or select the Layout button ( ).

Note: The **Layout** menu item and button are not available when the scope is in snapshot mode.

You can tile the window into multiple displays. For example, if there are three inputs to the scope, you can display the signals in three separate displays. The layout grid shows a 4 by 4 grid, but you can select up to 16 by 16 by clicking and dragging within the layout grid.



When you use the Layout option to tile the window into multiple displays, the display highlighted in blue is referred to as the *active display*. The scope dialog boxes reference the active display.

Signal Display

Time Scope uses the **Time span** and **Time display offset** parameters to determine the time range. To change the signal display settings, select **View** > **Configuration Properties** to bring up the Visuals—Time Domain Properties dialog box. Then, modify the values for the **Time span** and **Time display offset** parameters on the **Main** tab. For example, if you set the **Time span** to 25 seconds, the scope displays 25 seconds' worth of simulation data at a time. If you also set the **Time display offset** to 5 seconds, the scope displays values on the time axis from 5 to 30 seconds. The values on the time axis of the Time Scope display remain the same throughout simulation.

To communicate the simulation time that corresponds to the current display, the scope uses the *Time units*, *Time offset*, and *Simulation time* indicators on the scope window. The following figure highlights these and other important aspects of the Time Scope window.

Time Indicators

- *Minimum time-axis limit* — The Time Scope sets the minimum time-axis limit using the value of the **Time display offset** parameter on the **Main** tab of the Visuals—Time Domain Properties dialog box. If you specify a vector of values for the **Time display offset** parameter, the scope uses the smallest of those values to set the minimum time-axis limit.
- *Maximum time-axis limit* — The Time Scope sets the maximum time-axis limit by summing the value of **Time display offset** parameter with the value of the **Time span** parameter. If you specify a vector of values for the **Time display offset**

parameter, the scope sets the maximum time-axis limit by summing the largest of those values with the value of the **Time span** parameter.

- *Time units* — The units used to describe the time-axis. The Time Scope sets the time units using the value of the **Time Units** parameter on the **Time** tab of the Configuration Properties dialog box. By default, this parameter is set to `Metric (based on Time Span)` and displays in metric units such as milliseconds, microseconds, minutes, days, etc. You can change it to `Seconds` to always display the time-axis values in units of seconds. You can change it to `None` to not display any units on the time axis. When you set this parameter to `None`, then Time Scope shows only the word `Time` on the time axis.

  To hide both the word `Time` and the values on the time axis, set the **Show time-axis labels** parameter to `None`. To hide both the word `Time` and the values on the time axis in all displays, except the bottom ones in each column of displays, set this parameter to `Bottom Displays Only`. This behavior differs from the Simulink Scope block, which always shows the values but never shows a label on the *x*-axis.

For more information, see "Visuals — Time Domain Properties" on page 4-1750.

Simulation Indicators

- *Simulation status* — Provides the current status of the model simulation. The status can be either of the following conditions:

  - `Processing` — Occurs after you run the `step` method and before you run the `release` method.
  - `Stopped` — Occurs after you construct the scope object and before you first run the `step` method. This status also occurs after you run the `release` method.

  The *Simulation status* is part of the *Status Bar* in the scope window. You can choose to hide or display the entire *Status Bar*. From the scope menu, select **View > Status Bar**.
- *Time offset* — The `Offset` value helps you determine the simulation times for which the scope is displaying data. The value is always in the range `0`≤ `Offset`≤ *Simulation time*. If the time offset is 0, the Scope does not display the `Offset` status field. Add the Time offset to the fixed time span values on the time-axis to get the overall simulation time.

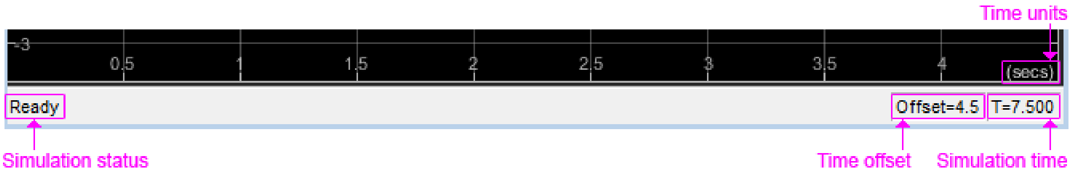
  For example, if you set the **Time span** to `20` seconds, and you see an `Offset` of `0 (secs)` on the scope window. This value indicates that the scope is displaying data for the first `0` to `20` seconds of simulation time. If the **Offset** changes to `20 (secs)`, the scope displays data for simulation times from `20` seconds to `40` seconds. The scope continues to update the `Offset` value until the simulation is complete.

- *Simulation time* — The amount of time that the Time Scope has spent processing the input. Every time you call the step method, the simulation time increases by the number of rows in the input signal divided by the sample rate, as given by the following formula: $t_{sim} = t_{sim} + \frac{\text{length(0:length(xsine))-1}}{\text{SampleRate}}$. You can set the sample rate using the `SampleRate` on page 4-1706 property. For frame-based inputs, the displayed Simulation time is the time at the beginning of the frame.

  The *Simulation time* is part of the *Status Bar* in the Time Scope window. You can choose to hide or display the entire *Status Bar*. From the Time Scope menu, select **View** > **Status Bar** .

Axes Maximization

When the scope is in maximized axes mode, the following figure highlights the important indicators on the scope window.



To toggle this mode, in the scope menu, select **View** > **Configuration Properties**. In the **Main** pane, locate the **Maximize axes** parameter.

Specify whether to display the scope in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- `Auto` — In this mode, the axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- `On` — In this mode, the axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.

- `Off` — In this mode, none of the axes appear maximized.

This property is Tunable (Simulink).

The default setting is `Auto`.

Reduce Updates to Improve Performance

By default, the scope updates the displays periodically at a rate not exceeding 20 hertz. If you would like the scope to update on every simulation time step, you can disable the **Reduce Updates to Improve Performance** option. However, as a recommended practice, leave this option enabled because doing so can significantly improve the speed of the simulation.

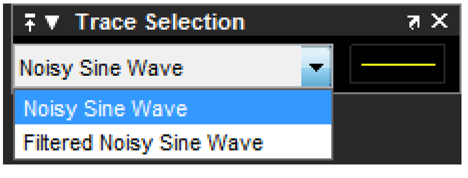
In the Time Scope menu, select **Playback** > **Reduce Updates to Improve Performance** to clear the check box. Alternatively, use the **Ctrl+R** shortcut to toggle this setting. You can also set the `ReduceUpdates` on page 4-1705 property to `false` to disable this option.

Measurements Panels

The Measurements panels are the five panels that appear to the right side of the Time Scope GUI. .

Trace Selection Panel

When you use the scope to view multiple signals, the Trace Selection panel appears if you have more than one signal displayed and you click any of the other Measurements panels. The Measurements panels display information about only the signal chosen in this panel. Choose the signal name for which you would like to display time domain measurements. See the following figure.



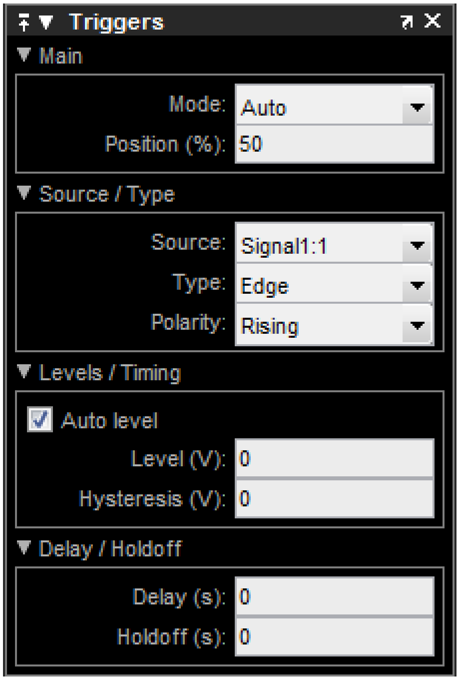
You can choose to hide or display the **Trace Selection** panel. In the Scope menu, select **Tools** > **Measurements** > **Trace Selection**.

Triggers Panel

The **Triggers** panel allows you to pause the display only when certain events occur. You can use the Triggers panel when you want to align or search for interesting events. You can configure triggers to both select and align specific regions of interest in the display area of the scope. Triggers work across multiple displays. You can also choose to hide or display the **Triggers** panel.

Note: If triggers are on and the model is

In the scope toolbar, click the Triggers button ( ). Alternatively, in the scope menu, select **Tools** > **Triggers**.



When the **Triggers** panel is displayed, triangle pointers appear at the top and right side of the axes on each display. These markers indicate the time position ( ) and level ( ) at the event. The color of the markers corresponds to the color of the source signal.

Note: The scope does not display an event until at least a full-time span is completely viewable inside the display. To prevent data from being shown twice in the display, the scope suppresses the alignment of recurring events until a full-time span has elapsed since the previous update.

Main Pane

The **Main** pane lets you choose how often the display updates and in what position the trigger indicator appears.

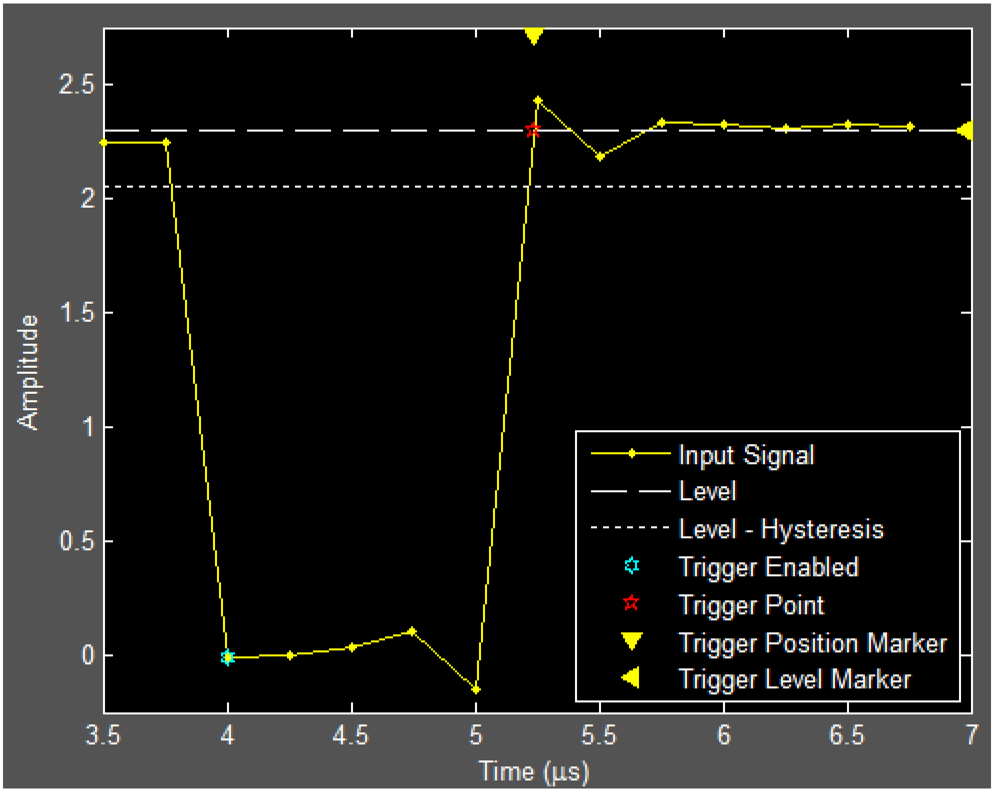
- **Mode** — Define how often the display should update.
  - `Auto` — The scope aligns and displays data from the latest trigger event. Data between trigger events is displayed. If no trigger event is found after an entire time span has elapsed, the scope displays that time span of data. Use this mode to see your data and have it align whenever a trigger event occurs.
  - `Normal` — The scope aligns and displays data only from the last trigger event in the time span and freezes the display. Data between trigger events is not displayed. Use this mode to search for infrequently occurring events in your data.
  - `Once` — The scope aligns and displays data on the first encountered trigger event and freezes the display. The scope ignores subsequent data until you press the **Rearm** button. If no trigger event is found, no data is shown in the display.
  - `Off` — Triggering is disabled. This setting is equivalent to hiding the **Triggers** panel. You can use panning only if **Mode** is set to `Off`.

  If mode is set to either `Normal` or `Once` and no trigger event occurs, the display remains blank. Set **Mode** to `Auto` to display all signal data, in addition to trigger events.
- **Position (%)** — Specify, as a percentage of the total time span within the active display, the horizontal position in which the trigger indicator appears. A position value of `0` corresponds to the minimum time-axis value at the far-left side of the display. A position value of `100` corresponds to the maximum time-axis value at the far-right side of the display. Drag the trigger position indicator to the left or right to adjust its position.

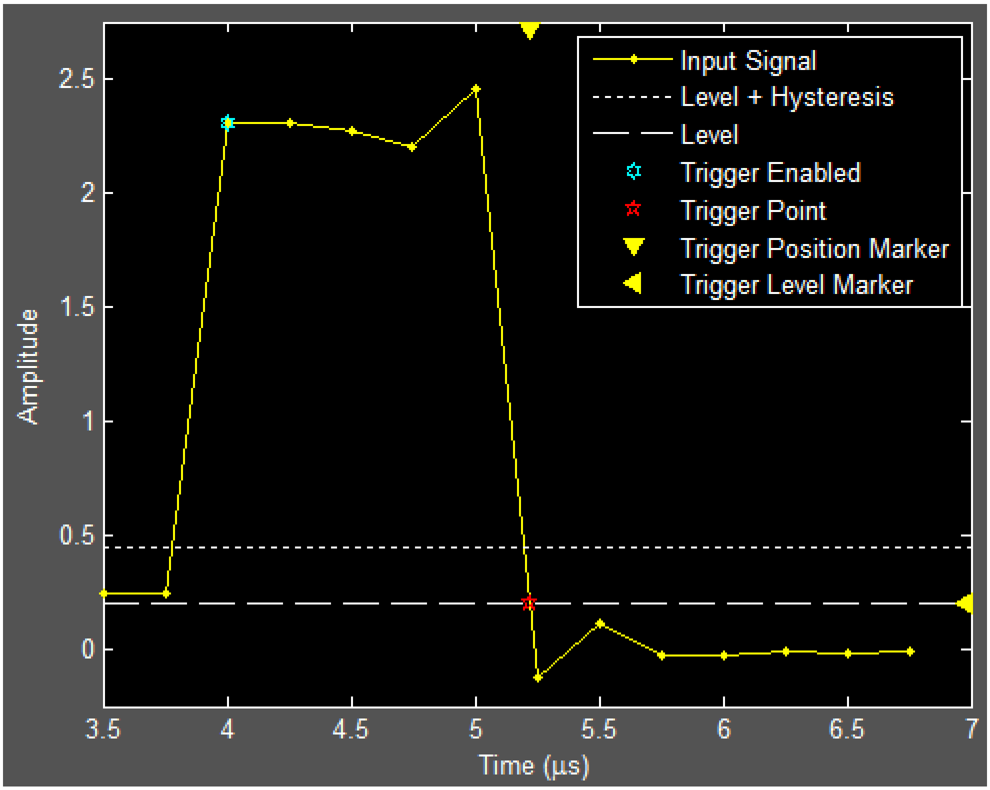
Source / Type Pane

The **Source / Type** pane lets you choose the source of the trigger and the type of events on which to stop.

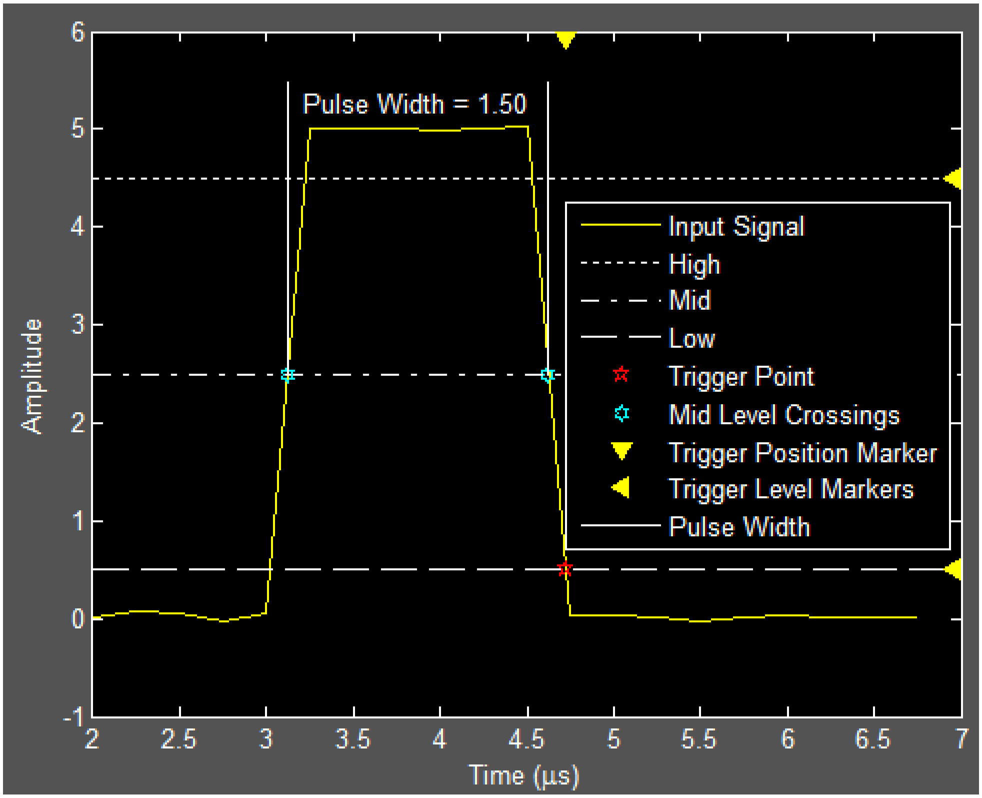
- **Source** — Assign the trigger source to a particular channel. If you are viewing a magnitude/phase plot, you can trigger off the magnitude or the phase. If you are not viewing the magnitude/phase plot, you can trigger off the real or imaginary data. If the input signal has multiple channels, the scope assigns an index number to identify each channel of that signal. For more information, see "Display Multiple Signals in the Time Scope".
- **Type** — Select the type of trigger to use.
  - `Edge` — Trigger when the scope crosses a level threshold. In the case of a rising edge, the scope enables the trigger event when the signal value becomes less than the level threshold minus hysteresis. The scope disables the trigger event when the signal becomes greater than the level threshold for the first time. The scope uses linear interpolation to generate a trigger event at the time when the signal crosses the level threshold, as shown in the following figure.



In the case of a falling edge, the scope enables the trigger event when the signal value becomes greater than the level threshold plus hysteresis. The scope disables the trigger event when the signal becomes less than the level threshold for the first time. The scope uses linear interpolation to generate a trigger event at the time when the signal crosses the level threshold, as shown in the following figure.

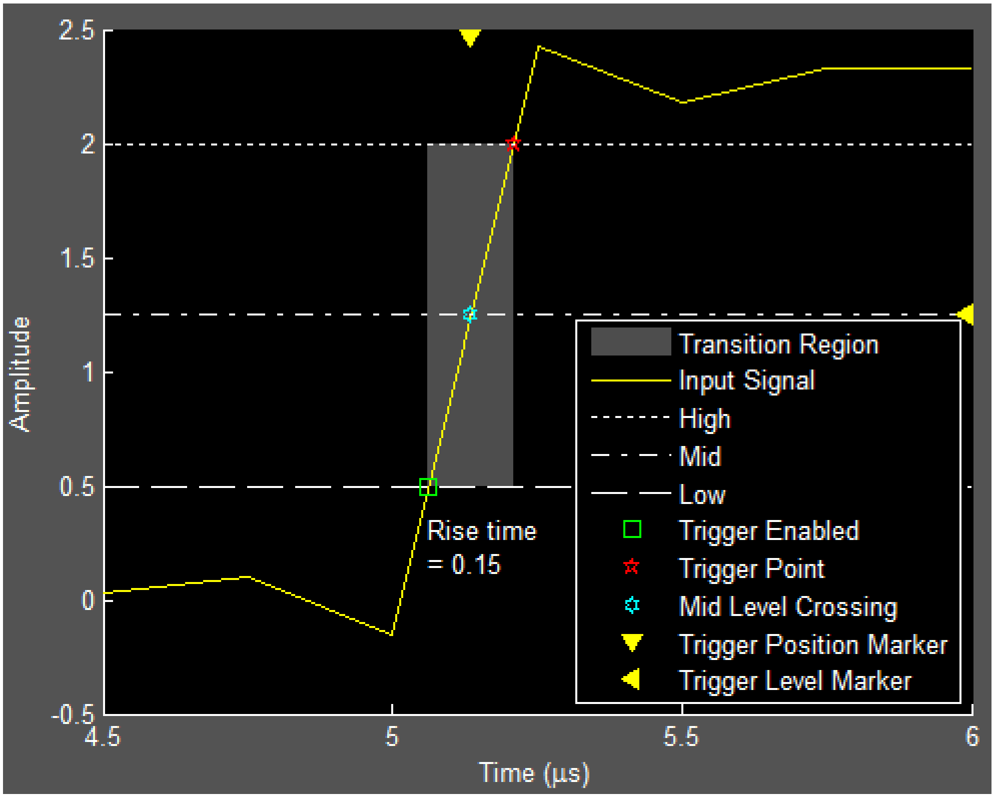


- `Pulse Width` — Trigger when the scope encounters a pulse whose width falls inside or outside specified time limits. You specify the range of valid time limits in the **Levels / Timing** pane. In the case of a positive-polarity pulse, the scope encounters a trigger event when the signal crosses the low threshold for the second time. The scope measures the pulse width as the time between the first and second crossings of the middle threshold, located halfway between the high and low thresholds, as shown in the following figure.

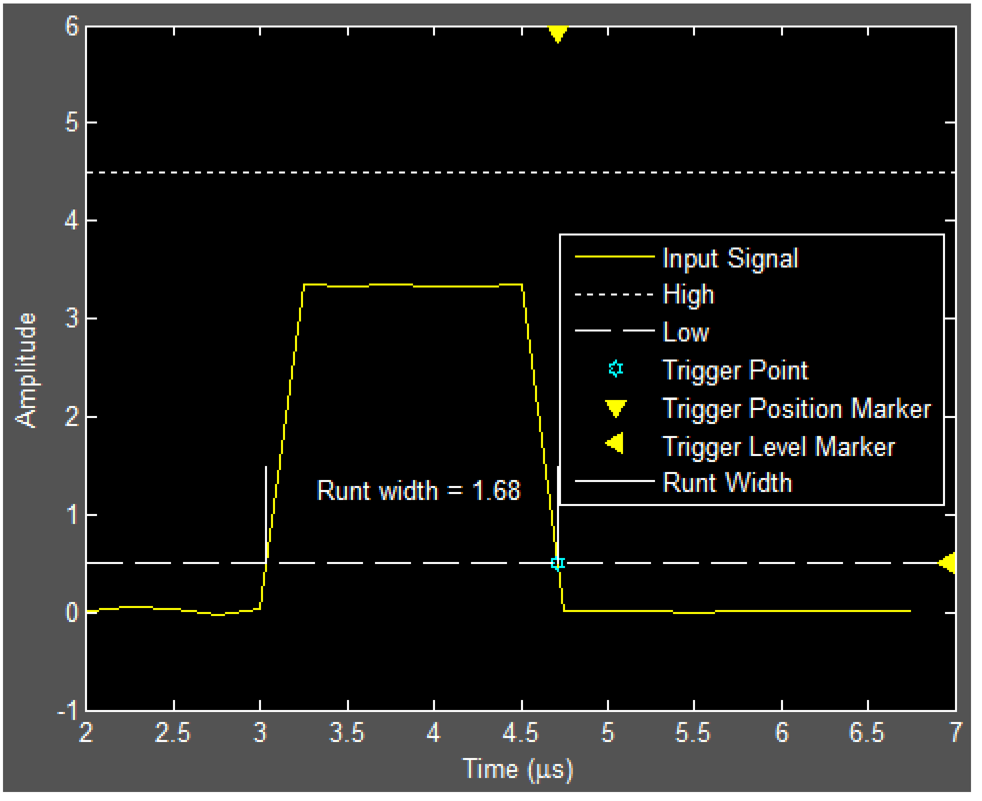


Note: A *Glitch*-type trigger looks for a pulse or spike whose duration is less than a specified amount. You can implement a *Glitch* type trigger by using a `Pulse Width` type trigger and manually setting the **Max Width** parameter.

- `Transition` — Trigger on a rising or falling edge that crosses two levels, high and low, inside or outside a specified time interval. You specify the range of valid transition times in the **Levels / Timing** pane. In the case of a rising transition, the scope encounters the trigger event when the signal crosses the high threshold. The transition time is when the signal crosses the middle threshold, located halfway between the high and low thresholds, as shown in the following figure.

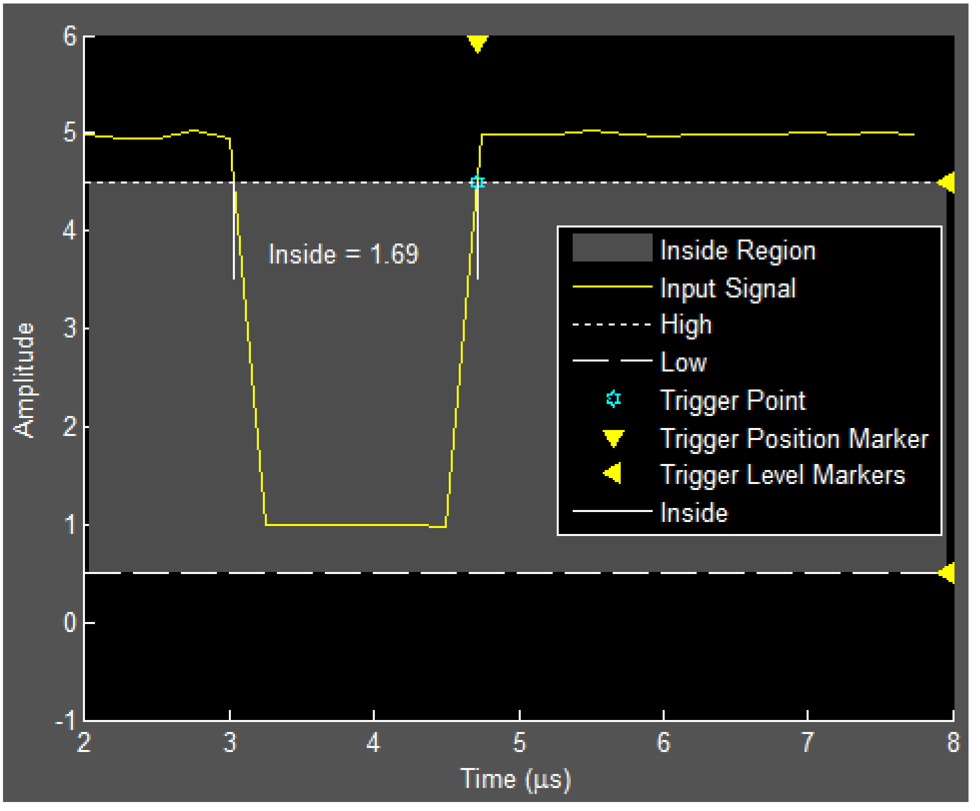


- `Runt` — Trigger on a runt pulse, which crosses one threshold, high or low, but not both. In the case of a positive-polarity runt pulse, the scope encounters a trigger event when the signal crosses the low threshold the second time, without ever crossing the high threshold. The scope measures the runt width as the time between the first and second crossings of the low threshold, as shown in the following figure. The runt width is the **Max Width** – **Min Width**. Any runt pulse width that is less than the minimum width or greater than the maximum width will not generate a trigger event.

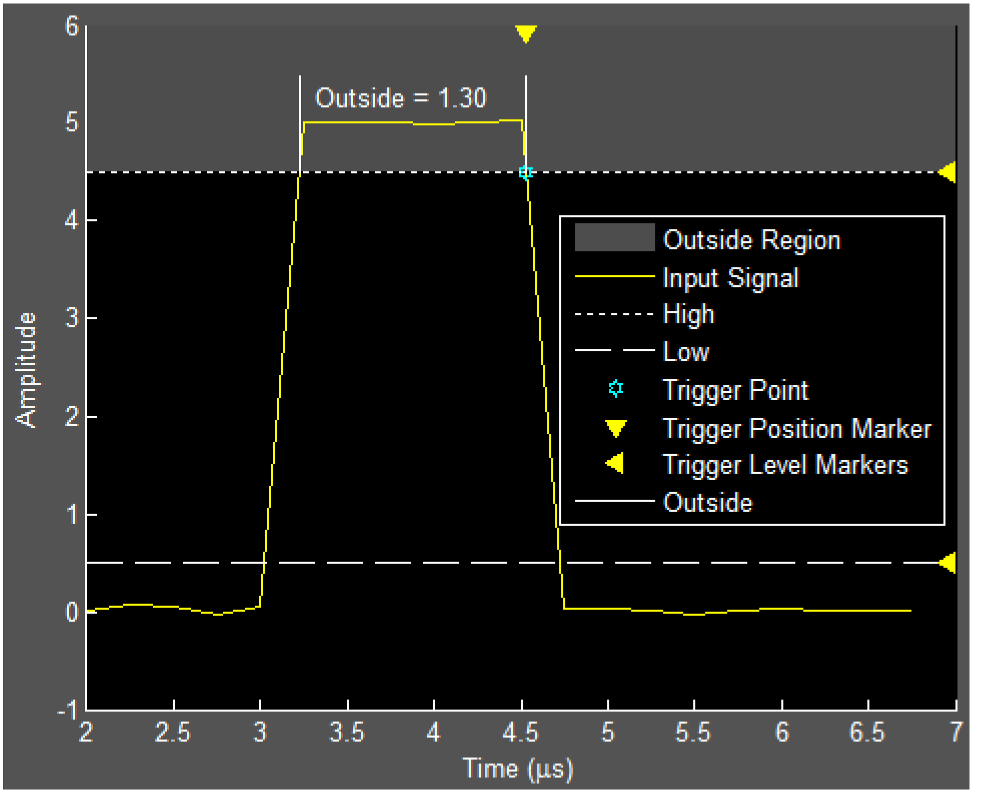


Note: You can also replicate a runt-type trigger by using a window-type trigger and setting **Polarity** to `Inside`.

- `Window` — Trigger when the input signal stays within or outside the region defined by the high and low thresholds for a period of time. In the case of an inside window, the scope encounters a trigger event when the signal enters and exits the inside region, as shown in the following figure.

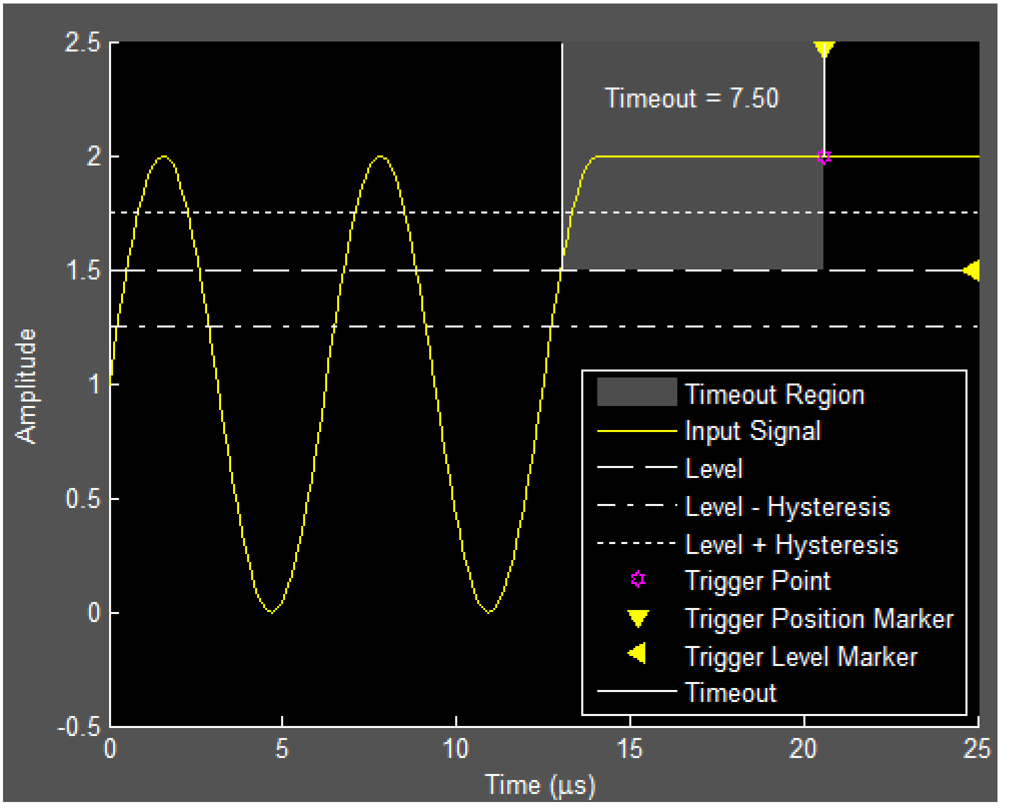


In the case of an outside window, the scope encounters a trigger event when the signal enters and exits the outside region, as shown in the following figure.

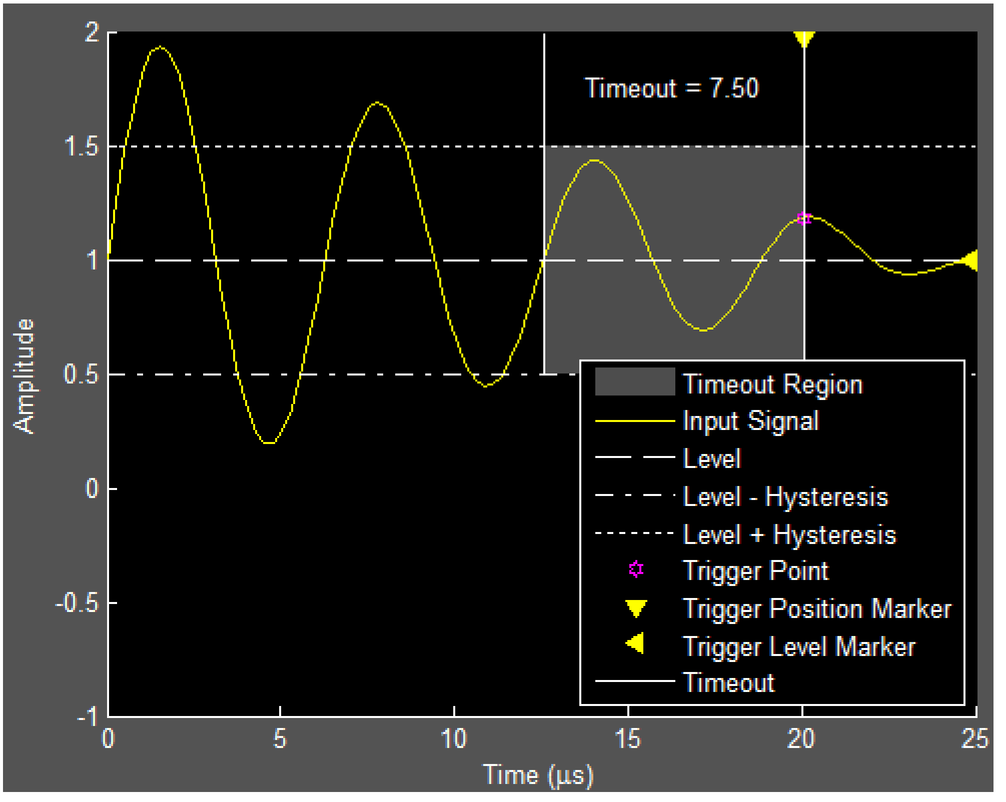


The scope encounters a trigger event when the signal crosses either the high or low threshold the second time.

- `Timeout` — Trigger when the input signal stays above or below a voltage threshold longer than a specified time. In the case of a timeout trigger with polarity set to `Either` and a timeout duration of 7.50 seconds, the scope can encounter the trigger event 7.50 seconds after the signal crosses the level threshold the last time, as shown in the following figure.



Alternatively, the scope can encounter the trigger event when the signal stays within the boundaries defined by the hysteresis for 7.50 seconds after the signal crosses the level threshold, as shown in the following figure.



- **Polarity** — Select the polarity of the trigger type. The option you choose for **Type** directly affects the options available for **Polarity**, as shown in the following table.

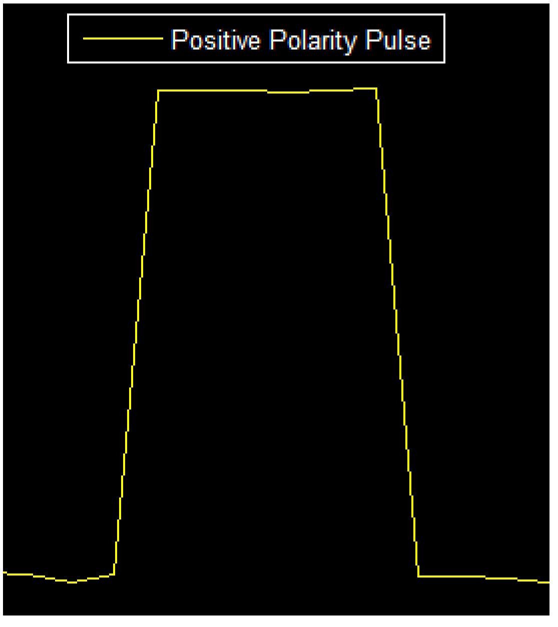
Trigger Type	Polarity Options
`Edge`	`Rising`, `Falling`, `Either`
`Pulse Width`	`Positive`, `Negative`, `Either`
`Transition`	`Rise Time`, `Fall Time`, `Either`
`Runt`	`Positive`, `Negative`, `Either`
`Window`	`Inside`, `Outside`, `Either`
`Timeout`	`Rising`, `Falling`, `Either`

When you set **Type** to `Edge`, the polarity options are:

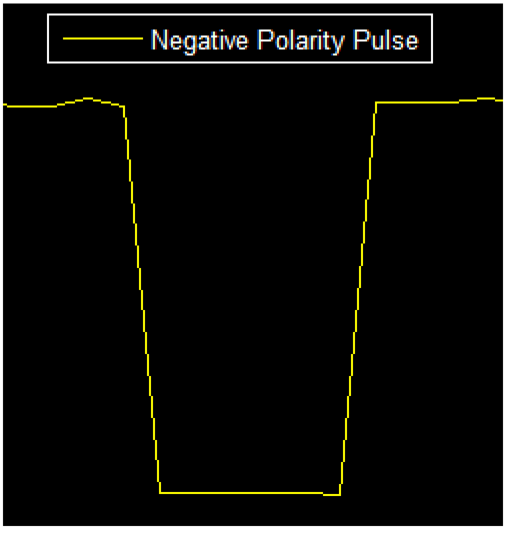
- `Rising` — Trigger on a *rising edge*, a transition from a low-state level to a high-state level.
- `Falling` — Trigger on a *falling edge*, transition from a high-state level to a low-state level.
- `Either` — Trigger on both rising edges and falling edges.

When you set **Type** to `Pulse Width` or `Runt`, the polarity options are:

- `Positive` — Trigger on a positive-polarity pulse, as shown in the following figure.



- `Negative` — Trigger on a negative-polarity pulse, as shown in the following figure.



- `Either` — Trigger on both positive-polarity and negative-polarity pulses.

When you set **Type** to `Transition`, the polarity options are:

- `Rise Time` — Trigger based on how long the signal takes to transition from the low threshold to the high threshold.
- `Fall Time` — Trigger based on how long the signal takes to transition from the high threshold to the low threshold.
- `Either` — Trigger based on how long it takes to make either a rising or falling transition.

When you set **Type** to `Window`, the polarity options are:

- `Inside` — Trigger when the signal stays within the low and high levels for a specified time duration.
- `Outside` — Trigger when the signal stays outside of the low and high levels for a specified time duration.
- `Either` — Trigger on both inside and outside windows.

When you set **Type** to `Timeout`, the polarity options are:

- `Rising` — Trigger when the signal does not cross the reference level from below.

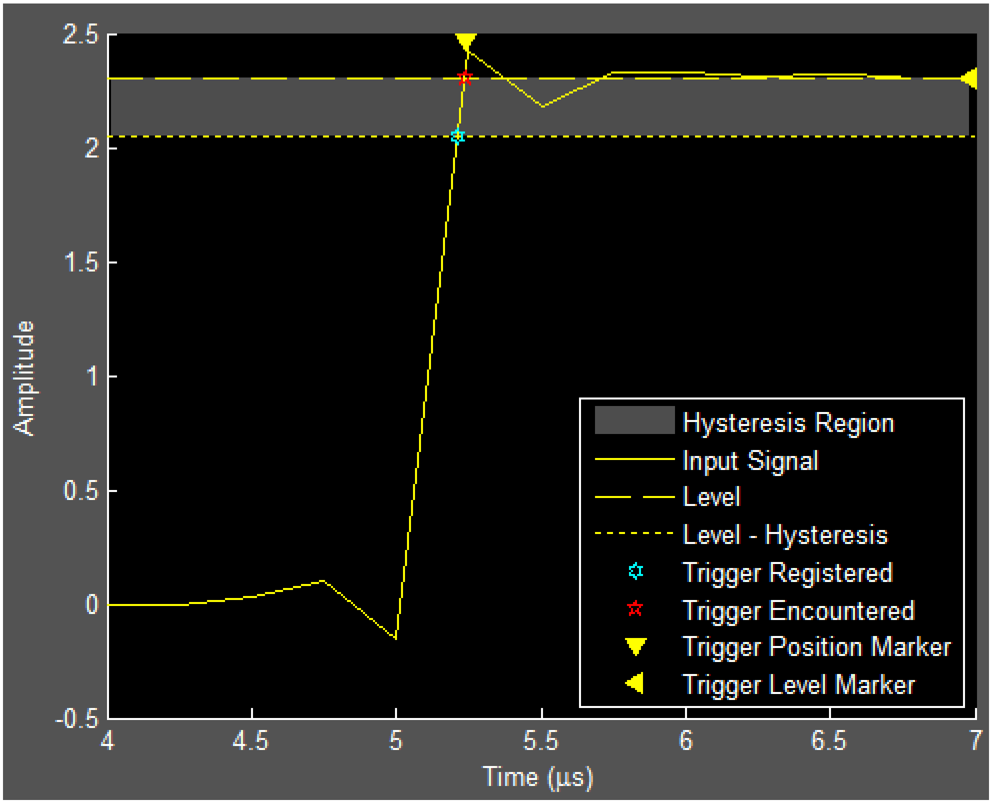
- `Falling` — Trigger when the signal does not cross the reference level from above.
- `Either` — Trigger when the signal does not cross the reference level from either direction.

Levels / Timing Pane

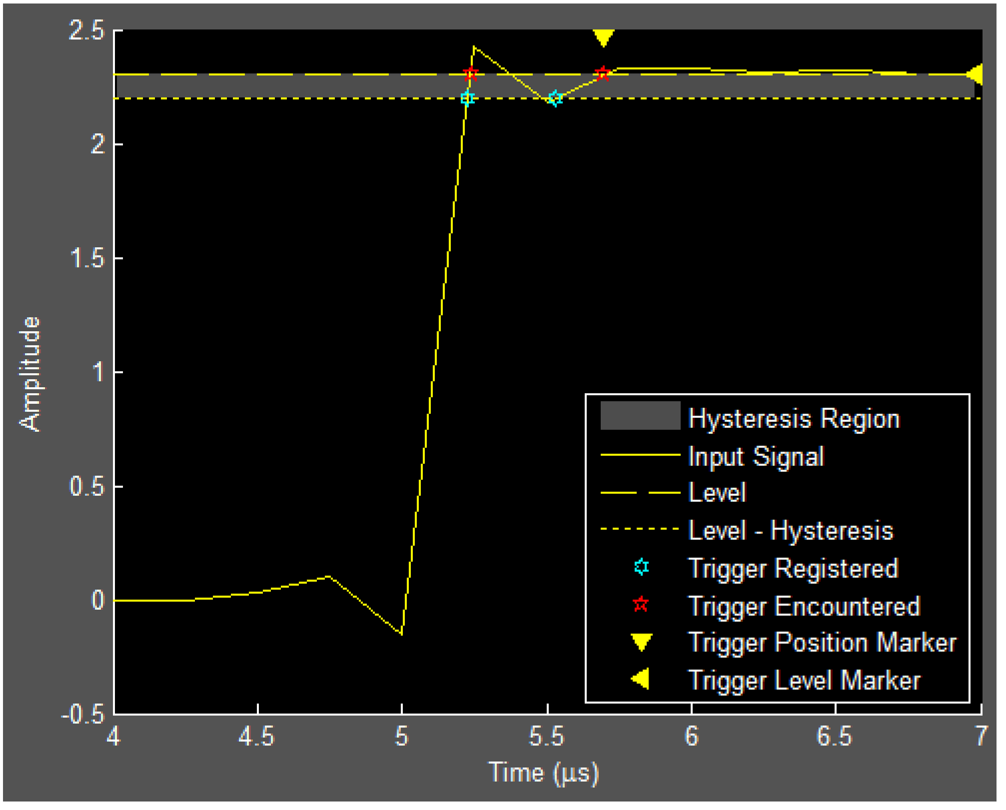
The **Levels / Timing** pane enables you to set the trigger level and hysteresis value. The option you choose for **Type** directly affects which level and timing parameters are available, as shown in the following table.

Trigger Type	Level Parameters	Auto-Level Setting	Timing Parameters
`Edge`	Level, Hysteresis	Level = 50%	n/a
`Pulse Width`	High, Low	High = 90%, Low = 10%	Min Width, Max Width
`Transition`	High, Low	High = 90%, Low = 10%	Min Time, Max Time
`Runt`	High, Low	High = 90%, Low = 10%	Min Width, Max Width
`Window`	High, Low	High = 90%, Low = 10%	Min Time, Max Time
`Timeout`	Level, Hysteresis	n/a	Timeout

- **Auto level** — Enable the Triggers panel to automatically choose the level parameters. If you set the trigger **Type** to `Edge`, this option sets the **Level** parameter to 50% of the range of the source signal. If you set the trigger **Type** to `Timeout`, the Triggers panel does not show this option. Setting the trigger **Type** to other menu choices results in **High** and **Low** parameter adjustment. **Auto level** sets the **High** parameter to 90% of the range of the source signal and the **Low** parameter to 10% of the range of the source signal.
- **Level (V)** — Specify, in volts, the trigger level. This parameter is visible when you set **Type** to `Edge` or `Timeout`.
- **Hysteresis (V)** — Specify, in volts, the hysteresis or noise reject value. This parameter is visible when you set **Type** to `Edge` or `Timeout`. If the signal jitters inside this range and briefly crosses the trigger level, the scope does not register an event. In the case of an edge trigger with rising polarity, the scope ignores any times that the signal crosses the trigger level within the hysteresis region, as shown in the following figure.



You can reduce the hysteresis region size by decreasing the hysteresis value. If you set the hysteresis value to 0.07 in this example, then the scope also considers the second rising edge to be a trigger event, as shown in the following figure.



- **High (V)** — Specify, in volts, the value that denotes a positive polarity, or high-state level. This parameter is visible when you set **Type** to `Pulse Width`, `Transition`, `Runt`, or `Window`.
- **Low (V)** — Specify, in volts, the value that denotes a negative polarity, or low-state level. This parameter is visible when you set **Type** to `Pulse Width`, `Transition`, `Runt`, or `Window`.
- **Min Width (s)** — Specify, in seconds, the minimum pulse width. This parameter is visible when you set **Type** to `Pulse Width` or `Runt`.
- **Max Width (s)** — Specify, in seconds, the maximum pulse width. This parameter is visible when you set **Type** to `Pulse Width` or `Runt`.
- **Min Time (s)** — Specify, in seconds, the minimum duration. This parameter is visible when you set **Type** to `Transition` or `Window`.

- **Max Time (s)** — Specify, in seconds, the maximum duration. This parameter is visible when you set **Type** to `Transition` or `Window`.
- **Timeout (s)** — Specify, in seconds, the timeout duration. This parameter is visible when you set **Type** to `Timeout`.

Delay / Holdoff Pane

The **Delay / Holdoff** pane enables you to offset the trigger position by a fixed delay or set the minimum possible time between trigger events.

- **Delay (s)** — Specify, in seconds, the fixed delay time by which to offset the trigger position. This parameter controls the amount of time the scope waits after a trigger event occurs before displaying a signal.
- **Holdoff (s)** — Specify, in seconds, the minimum possible time between trigger events. This amount of time is used to suppress data acquisition after a valid trigger event is encountered. A trigger holdoff prevents repeated occurrences of a trigger from occurring during the portion of a burst that is of interest.

Cursor Measurements Panel

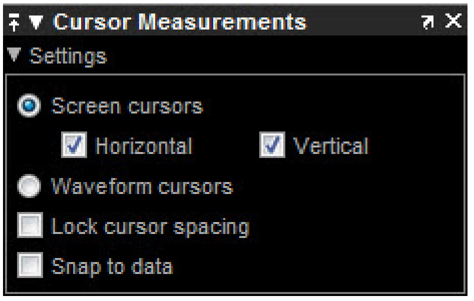
The **Cursor Measurements** panel displays screen cursors. The panel provides two types of cursors for measuring signals. Waveform cursors are vertical cursors that track along the signal. Screen cursors are both horizontal and vertical cursors that you can place anywhere in the display.

Note: If a data point in your signal has more than one value, the cursor measurement at that point is undefined and no cursor value is displayed.

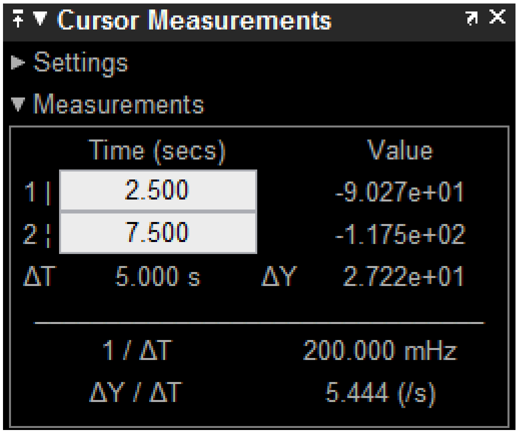
In the Scope menu, select **Tools** > **Measurements** > **Cursor Measurements**.

Alternatively, in the Scope toolbar, click the Cursor Measurements  button.

The **Settings** pane enables you to modify the type of screen cursors used for calculating measurements. When more than one signal is displayed, you can assign cursors to each trace individually.



- **Screen Cursors** — Shows screen cursors.
- **Horizontal** — Shows horizontal screen cursors.
- **Vertical** — Shows vertical screen cursors.
- **Waveform Cursors** — Shows cursors that attach to the input signals.
- **Lock Cursor Spacing** — Locks the frequency difference between the two cursors.
- **Snap to Data** — Positions the cursors on signal data points.



You can use the mouse or the left and right arrow keys to move vertical or waveform cursors and the up and down arrow keys for horizontal cursors.

The **Measurements** pane shows the time and value measurements.

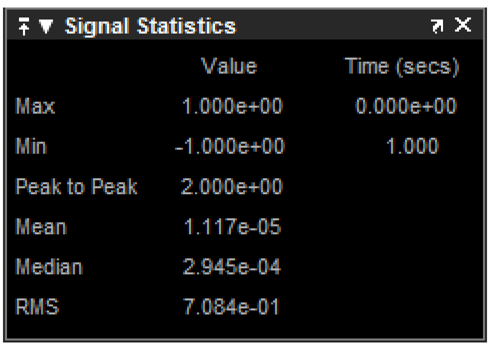
- **1 |**— Shows or enables you to modify the time or value at cursor number one, or both.
- **2 :**— Shows or enables you to modify the time or value at cursor number two, or both.

- **Δt**— Shows the absolute value of the difference in the times between cursor number one and cursor number two.
- **ΔV**— Shows the absolute value of the difference in signal amplitudes between cursor number one and cursor number two.
- **1/Δt**— Shows the rate, the reciprocal of the absolute value of the difference in the times between cursor number one and cursor number two.
- **ΔV/Δt**— Shows the scope, the ratio of the absolute value of the difference in signal amplitudes between cursors to the absolute value of the difference in the times between cursors.

Signal Statistics Panel

Note: The **Signal Statistics** panel requires a DSP System Toolbox or Simscape license.

The **Signal Statistics** panel displays the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal. It also shows the *x*-axis indices at which the maximum and minimum values occur. In the Scope menu, select **Tools** > **Measurements** > **Signal Statistics**. Alternatively, in the scope toolbar, click the Signal Statistics  button.



The statistics shown are:

- **Max** — The maximum or largest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `max` function reference.

- **Min** — The minimum or smallest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `min` function reference.
- **Peak to Peak** — The difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `peak2peak` function reference.
- **Mean** —The average or mean of all the values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `mean` function reference.
- **Median** — The median value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `median` function reference.
- **RMS** — Shows the difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `rms` function reference.

When you use the zoom options in the Scope, the Signal Statistics measurements automatically adjust to the time range shown in the display. For example, you can zoom in on one pulse to make the **Signal Statistics** panel display information about only that particular pulse.

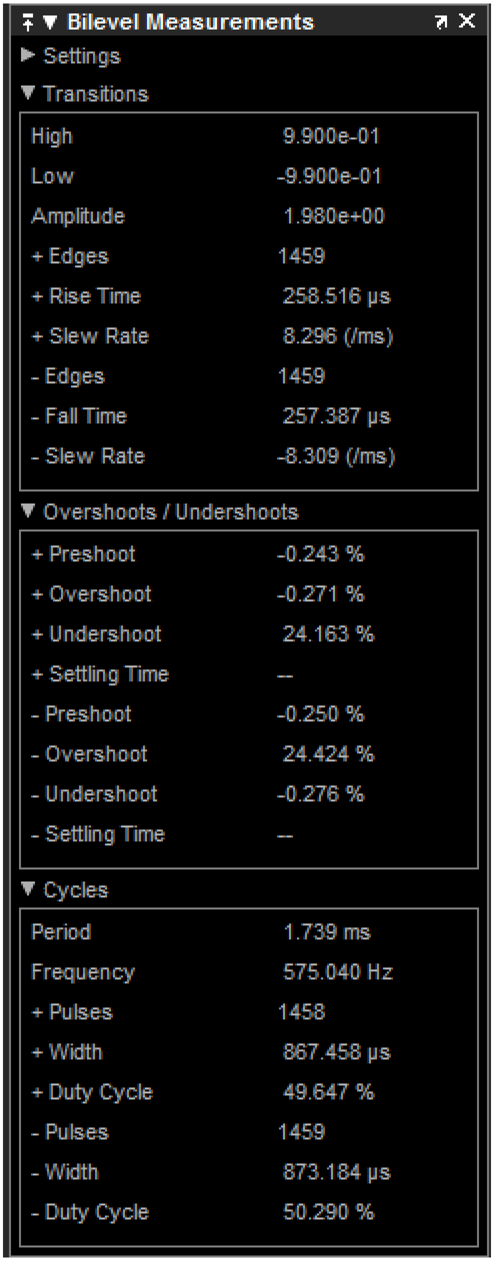
The Signal Statistics measurements are valid for any units of the input signal. The letter after the value associated with each measurement represents the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Bilevel Measurements Panel

Note: The **Bilevel Measurements** panel requires a DSP System Toolbox or Simscape license.

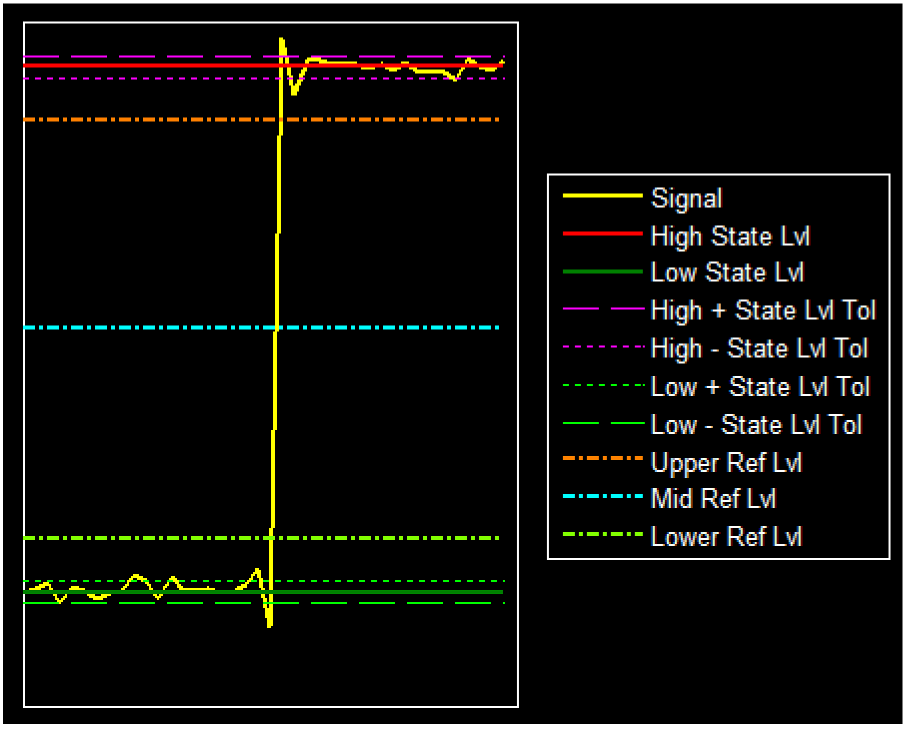
The **Bilevel Measurements** panel shows information about transitions, overshoots, undershoots, and cycles for a selected signal. You can choose to hide or display the **Bilevel Measurements** panel. In the scope menu, select **Tools** > **Measurements** > **Bilevel Measurements**. Alternatively, in the scope toolbar, you can select the Bilevel Measurements  button.

When you use the zoom options in the Scope, the bilevel measurements automatically adjust to the time range shown in the display. For example, you can zoom in on one rising edge to make the **Bilevel Measurements** panel display information about only that particular rising edge. This feature does not apply to the **High** and **Low** measurements.


Bilevel Measurements
Settings
Transitions
High 9.900e-01
Low -9.900e-01
Amplitude 1.980e+00
+ Edges 1459
+ Rise Time 258.516 μs
+ Slew Rate 8.296 (/ms)
- Edges 1459
- Fall Time 257.387 μs
- Slew Rate -8.309 (/ms)
Overshoots / Undershoots
+ Preshoot -0.243 %
+ Overshoot -0.271 %
+ Undershoot 24.163 %
+ Settling Time --
- Preshoot -0.250 %
- Overshoot 24.424 %
- Undershoot -0.276 %
- Settling Time --
Cycles
Period 1.739 ms
Frequency 575.040 Hz
+ Pulses 1458
+ Width 867.458 μs
+ Duty Cycle 49.647 %
- Pulses 1459
- Width 873.184 μs
- Duty Cycle 50.290 %

The **Bilevel Measurements** panel is separated into four panes, labeled **Settings**, **Transitions**, **Overshoots/Undershoots**, and **Cycles**. You can expand each pane to see the available options.

Bilevel Measurements Plot



Note: Opening the **Overshoots/Undershoots** pane may reduce the number of available edges by 1 if the last edge of the pulse is not within the **Settle Seek** time of the end of the plot. This reduction in the number of edges changes the values of **Edges**, **Fall Time**, and **Slew Rate** in the **Transitions** pane.

The **Settings** pane enables you to modify the properties used to calculate various measurements involving transitions, overshoots, undershoots, and cycles. You can modify the high-state level, low-state level, state-level tolerance, upper-reference level, mid-reference level, and lower-reference level, as shown in the following figure.

- **Auto State Level** — When this check box is selected, the Bilevel measurements panel autodetects the high- and low- state levels of a bilevel waveform. For more information on the algorithm this option uses, see the Signal Processing Toolbox `statelevels` function reference. When this check box is cleared, you can enter in values for the high- and low- state levels manually.
  - **High** — Manually specify the value for a positive polarity or high-state level.
  - **Low** — Manually specify the value for a negative polarity or low-state level.
- **State Level Tolerance** — Tolerance within which the initial and final levels of each transition must be within their respective state levels. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Upper Ref Level** — Used to compute the end of the rise-time measurement or the start of the fall time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Mid Ref Level** — Used to determine when a transition occurs. This value is expressed as a percentage of the difference between the high- and low- state levels. The mid-reference level is shown as a horizontal line, and its corresponding mid-reference level instant is shown as a vertical line.
- **Lower Ref Level** — Used to compute the end of the fall-time measurement or the start of the rise-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Settle Seek** — The duration after the mid-reference level instant when each transition occurs used for computing a valid settling time. This value is equivalent to the input parameter, `D`, which you can set when you run the `settlingtime` function. The settling time is displayed in the **Overshoots/Undershoots** pane.

The **Transitions** pane displays calculated measurements associated with the input signal changing between its two possible state level values, high and low. The Transition measurements assume that the amplitude of the input signal is in units of volts. Convert all input signals to volts for the Transition measurements to be valid.

A positive-going transition, or *rising edge*, in a bilevel waveform is a transition from the low-state level to the high-state level. A positive-going transition has a slope value greater than zero. Whenever there is a plus sign (+) next to a text label, this symbol refers to measurement associated with a rising edge, a transition from a low-state level to a high-state level.

A negative-going transition, or falling edge, in a bilevel waveform is a transition from the high-state level to the low-state level. A negative-going transition has a slope value less than zero. Whenever there is a minus sign (–) next to a text label, this symbol refers to measurement associated with a falling edge, a transition from a high-state level to a low-state level.

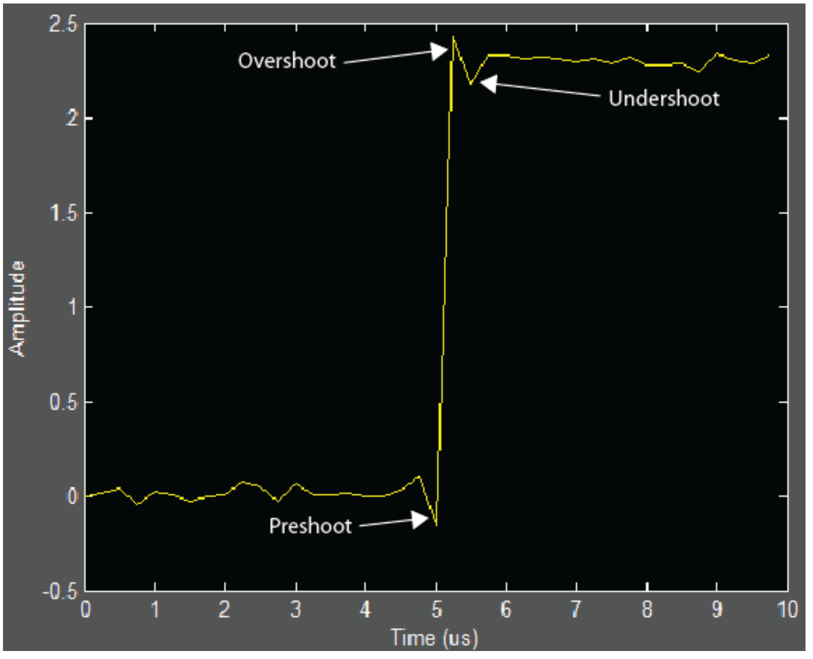
- **High** — The high-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** or**Time** tab of the Visuals—Time Domain Properties dialog box. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `statelevels` function reference.
- **Low** — The low-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** or **Time** tab of the Visuals—Time Domain Properties dialog box. The tab in which **Time Span** appears depends on whether you launched the scope from a from a MATLAB System object or from a Simulink block, respectively. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `statelevels` function reference.
- **Amplitude** — Difference in amplitude between the high-state level and the low-state level.
- **+ Edges** — Total number of positive-polarity, or rising, edges counted within the displayed portion of the input signal.
- **+ Rise Time** — Average amount of time required for each rising edge to cross from the lower-reference level to the upper-reference level. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `risetime` function reference.
- **+ Slew Rate** — Average slope of each rising-edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. The region in which the slew rate is calculated appears in gray.

  For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `slewrate` function reference.
- **– Edges** — Total number of negative-polarity or falling edges counted within the displayed portion of the input signal.
- **– Fall Time** — Average amount of time required for each falling edge to cross from the upper-reference level to the lower-reference level. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `falltime` function reference.
- **– Slew Rate** — Average slope of each falling edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. For

more information on the algorithm this measurement uses, see the Signal Processing Toolbox `slewrate` function reference.

The **Overshoots/Undershoots** pane displays calculated measurements involving the distortion and damping of the input signal. *Overshoot* and *undershoot* refer to the amount that a signal, respectively, exceeds and falls below its final steady-state value. *Preshoot* refers to the amount before a transition that a signal varies from its initial steady-state value. This figure shows preshoot, overshoot, and undershoot for a rising-edge transition.

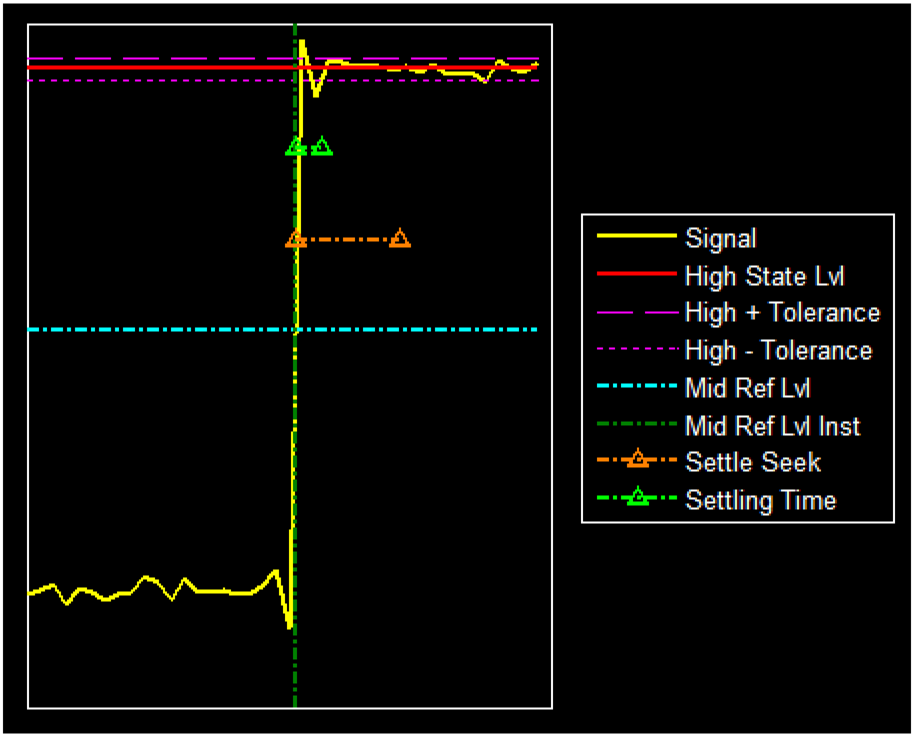
Overshoot/Undershoot Plot



- + *Preshoot* — Average lowest aberration in the region immediately preceding each rising transition.
- + *Overshoot* — Average highest aberration in the region immediately following each rising transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `overshoot` function reference.
- + *Undershoot* — Average lowest aberration in the region immediately following each rising transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `undershoot` function reference.

- + *Settling Time* — Average time required for each rising edge to enter and remain within the tolerance of the high-state level for the remainder of the settle seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the high-state level. This crossing is illustrated in the following figure.

Settling Time Plot



You can modify the settle seek duration parameter in the **Settings** pane. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `settlingtime` function reference.

- – *Preshoot* — Average highest aberration in the region immediately preceding each falling transition.

- – *Overshoot* — Average highest aberration in the region immediately following each falling transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `overshoot` function reference.
- – *Undershoot* — Average lowest aberration in the region immediately following each falling transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `undershoot` function reference.
- – *Settling Time* — Average time required for each falling edge to enter and remain within the tolerance of the low-state level for the remainder of the settle seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the low-state level. You can modify the settle seek duration parameter in the **Settings** pane. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `settlingtime` function reference.

The **Cycles** pane displays calculated measurements of to repetitions or trends in the displayed portion of the input signal.

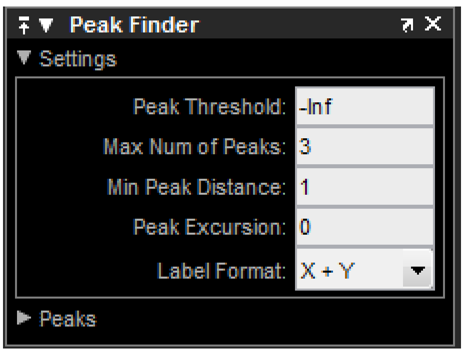
- **Period** — Average duration between adjacent edges of identical polarity within the displayed portion of the input signal. The Bilevel measurements panel calculates period as follows. It takes the difference between the mid-reference level instants of the initial transition of each positive-polarity pulse and the next positive-going transition. These mid-reference level instants appear as red dots.

  For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulseperiod` function reference.
- **Frequency** — Reciprocal of the average period. Whereas period is typically measured in some metric form of seconds, or seconds per cycle, frequency is typically measured in hertz or cycles per second.
- **+ Pulses** — Number of positive-polarity pulses counted.
- **+ Width** — Average duration between rising and falling edges of each positive-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulsewidth` function reference.
- **+ Duty Cycle** — Average ratio of pulse width to pulse period for each positive-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `dutycycle` function reference.
- **– Pulses** — Number of negative-polarity pulses counted.

- – **Width** — Average duration between rising and falling edges of each negative-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulsewidth` function reference.
- – **Duty Cycle** — Average ratio of pulse width to pulse period for each negative-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `dutycycle` function reference.

Peak Finder Panel

The **Peak Finder** panel displays the maxima, showing the *x*-axis values at which they occur. Peaks are defined as a local maximum where lower values are present on both sides of a peak. Endpoints are not considered to be peaks. This panel allows you to modify the settings for peak threshold, maximum number of peaks, and peak excursion. You can choose to hide or display the **Peak Finder** panel. In the scope menu, select **Tools** > **Measurements** > **Peak Finder**. Alternatively, in the scope toolbar, select the Peak Finder  button.

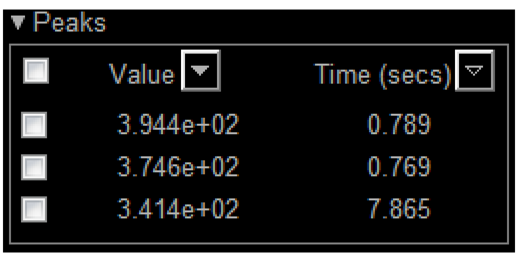


The **Peak finder** panel is separated into two panes, labeled **Settings** and **Peaks**. You can expand each pane to see the available options.

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input signal. For more information on the algorithms this pane uses, see the Signal Processing Toolbox `findpeaks` function reference.

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the `MINPEAKHEIGHT` parameter, which you can set when you run the `findpeaks` function.
- **Max Num of Peaks** — The maximum number of peaks to show. The value you enter must be a scalar integer from 1 through 99. This setting is equivalent to the `NPEAKS` parameter, which you can set when you run the `findpeaks` function.
- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the `MINPEAKDISTANCE` parameter, which you can set when you run the `findpeaks` function.
- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples The peak excursion setting is equivalent to the `THRESHOLD` parameter, which you can set when you run the `findpeaks` function.
- **Label Format** — The coordinates to display next to the calculated peak values on the plot. To see peak values, expand the **Peaks** pane and select the check boxes associated with individual peaks of interest. By default, both *x*-axis and *y*-axis values are displayed on the plot. Select which axes values you want to display next to each peak symbol on the display.
  - `X+Y` — Display both *x*-axis and *y*-axis values.
  - `X` — Display only *x*-axis values.
  - `Y` — Display only *y*-axis values.

The **Peaks** pane displays all of the largest calculated peak values. It also shows the coordinates at which the peaks occur, using the parameters you define in the **Settings** pane. You set the **Max Num of Peaks** parameter to specify the number of peaks shown in the list.



The numerical values displayed in the **Value** column are equivalent to the `pks` output argument returned when you run the `findpeaks` function. The numerical values

displayed in the second column are similar to the `locs` output argument returned when you run the `findpeaks` function.

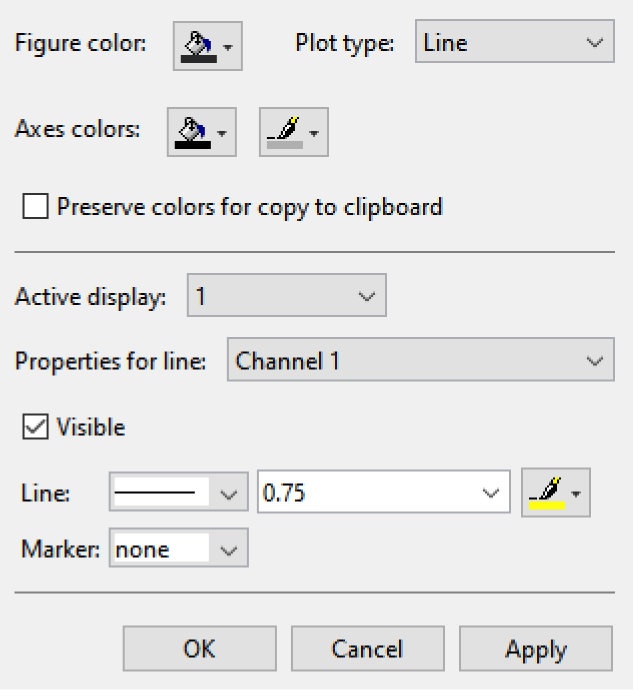
The Peak Finder displays the peak values in the **Peaks** pane. By default, the **Peak Finder** panel displays the largest calculated peak values in the **Peaks** pane in decreasing order of peak height. Use the sort descending button ( ) to rearrange the category and order by which Peak Finder displays peak values. Click this button again to sort the peaks in ascending order instead. When you do so, the arrow changes direction to become the sort ascending button ( ). A filled sort button indicates that the peak values are currently sorted in the direction of the button arrow. If the sort button is not filled ( ), then the peak values are sorted in the opposite direction of the button arrow. The **Max Num of Peaks** parameter still controls the number of peaks listed.

Use the check boxes to control which peak values are shown on the display. By default, all check boxes are cleared and the **Peak Finder** panel hides all the peak values. To show all the peak values on the display, select the check box in the top-left corner of the **Peaks** pane. To hide all the peak values on the display, clear this check box. To show an individual peak, select the check box directly to the left of its **Value** listing. To hide an individual peak, clear the check box directly to the left of its **Value** listing.

The Peaks are valid for any units of the input signal. The letter after the value associated with each measurement indicates the abbreviation for the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Style Dialog Box

Select **View > Style** or the Style button ( ) in the dropdown below the Configuration Properties button to open the Style dialog box. In this dialog box, you can change the figure colors, background axes colors, foreground axes colors, and properties of lines in a display.



Properties

The **Style** dialog box allows you to modify the following properties of the scope figure:

Figure color

Specify the color that you want to apply to the background of the scope figure. By default, the figure color is gray.

Plot type

Specify the type of plot to use. The default setting is `Line`. Valid values for **Plot type** are:

- `Line` — line graph, similar to the `line` or `plot` function.
- `Stairs` — *stair-step* graph, similar to the `stairs` function. Stair-step graphs are useful for drawing time history graphs of digitally sampled data.

This property is Tunable (Simulink).

Active display

Specify the active display as an integer to get and set relevant properties. The number of a display corresponds to its column-wise placement index. Set this property to control which display has its axes colors, line properties, marker properties, and visibility changed. Tunable (Simulink)

When you use the Layout option to tile the window into multiple displays, the display highlighted in blue is referred to as the *active display*. The default setting is 1.

Axes colors

Specify the color that you want to apply to the background of the axes for the active display.

Preserve colors for copy to clipboard

Specify whether or not to use the displayed color of the scope when copying.

When you select **File** > **Copy to Clipboard**, the software changes the color of the scope to be printer friendly (white background, visible lines). If you want to copy and paste the scope with the colors displayed, select this check box.

Default: Off

Properties for line

Specify the signal for which you want to modify the visibility, line properties, and marker properties.

Visible

Specify whether the selected signal on the active display is visible. If you clear this check box, the line disappears.

Line

Specify the line style, line width, and line color for the selected signal on the active display.

Marker

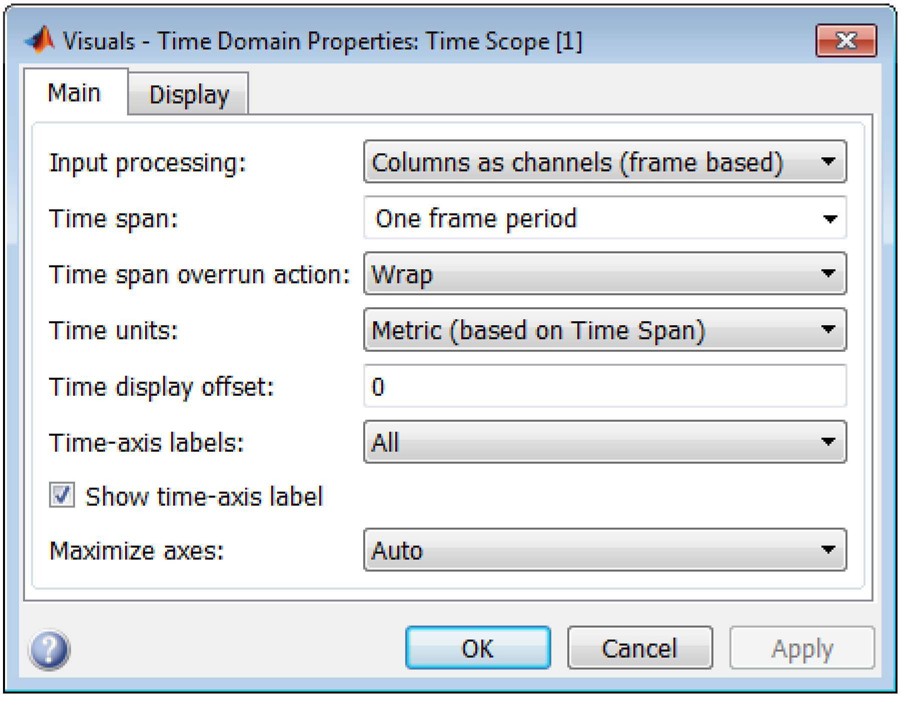
Specify marks for the selected signal on the active display to show at data points. This property is similar to the `Marker` property for plot objects. Choose any of the marker symbols from the dropdown list.

Visuals — Time Domain Properties

The Visuals — Time Domain Properties dialog box controls the visual configuration settings of the Time Scope displays. From the Time Scope menu, select **View** > **Configuration Properties** to open this dialog box.

Main Pane

The **Main** pane of the Time Scope Visuals—Time Domain Properties dialog box appears as follows.



Input processing

Specify whether the Time Scope treats the input signal as `Columns as channels (frame based)` or `Elements as channels (sample based)`.

Frame-based processing is only available for discrete input signals. For more information about frame-based input channels, see “What Is Frame-Based Processing?”. For an example that uses the Time Scope block and frame-based input signals, see “Display Time-Domain Data”.

Time span

Specify the time span, either by selecting a predefined option or by entering a numeric value in seconds. You can select one of the following options:

- `Auto` — In this mode, the Time Scope automatically calculates the appropriate value for time span from the difference between the simulation “Start time” (Simulink) and “Stop time” (Simulink) properties. This option is available only for the Time Scope block but not for the `dsp.TimeScope` System object.
- `One frame period` — In this mode, the Time Scope uses the frame period of the input signal to the Time Scope block. This option is only available when the **Input processing** parameter is set to `Columns as channels (frame based)`. This option is not available when you set the **Input processing** parameter to `Elements as channels (sample based)`.
- `<user defined>` — In this mode, you specify the time span by replacing the text `<user defined>` with a numeric value in seconds.

The scope sets the time-axis limits using the value of this property and the value of the **Time display offset** property. For example, if you set the **Time display offset** to `5e-6` and the **Time span** to `25e-6`, the scope sets the time-axis limits to a minimum value of 5 μs and a maximum value of 30 μs.

This property is Tunable (Simulink).

Time span overrun action

Specify how the scope displays new data beyond the visible time span. You can select one of the following options:

- `Wrap` — In this mode, the scope displays new data until the data reaches the maximum time-axis limit. When the data reaches the maximum time-axis limit of the scope window, the scope clears the display. The scope then updates the time offset value and begins displaying subsequent data points starting from the minimum time-axis limit.

- `Scroll` — In this mode, the scope scrolls old data to the left to make room for new data on the right side of the scope display. This mode is graphically intensive and can affect run-time performance. However, it is beneficial for debugging and monitoring time-varying signals.

This property is Tunable (Simulink).

The default setting is `Wrap`.

Time units

Specify the units used to describe the time axis. The default setting is `Metric`. You can select one of the following options:

- `Metric` — In this mode, Time Scope converts the times on the time axis to some metric units such as milliseconds, microseconds, days, etc. Time Scope chooses the appropriate metric units, based on the minimum time-axis limit and the maximum time-axis limit of the scope window.
- `Seconds` — In this mode, Time Scope always displays the units on the time axis as seconds.
- `None` — In this mode, Time Scope displays no units on the time axis. Time Scope shows only the word `Time` on the time axis.

This property is Tunable (Simulink).

Time display offset

This property allows you to offset the values displayed on the time axis by a specified number of seconds. When you specify a scalar value, the scope offsets all channels equally. When you specify a vector of offset values, the scope offsets each channel independently. Tunable (Simulink).

When you specify a **Time display offset** vector of length *N*, the scope offsets the input channels as follows:

- When *N* is equal to the number of input channels, the scope offsets each channel according to its corresponding value in the offset vector.

- When N is less than the number of input channels, the scope applies the values you specify in the offset vector to the first N input channels. The scope does not offset the remaining channels.
- When N is greater than the number of input channels, the scope offsets each input channel according to the corresponding value in the offset vector. The scope ignores all values in the offset vector that do not correspond to a channel of the input.

The scope computes the time-axis range using the values of the **Time display offset** and **Time span** properties. For example, if you set the **Time display offset** to `5e-6` and the **Time span** to `25e-6`, the scope sets the time-axis limits to a minimum value of 5 μs and a maximum value of 30 μs.

Similarly, when you specify a vector of values, the scope sets the minimum time-axis limit using the smallest value in the vector. To set the maximum time-axis limit, the scope sums the largest value in the vector with the value of the **Time span** property. For more information, see "Signal Display" on page 4-1713.

Time-axis labels

Specify how to display the time units used to describe the time axis. The default setting is `All`. You can select one of the following options:

- `All` — In this mode, the time-axis labels appear in all displays.
- `None` — In this mode, the time-axis labels do not appear in the displays.
- `Bottom Displays Only` — In this mode, the time-axis labels appear in only the bottom row of the displays.

Tunable (Simulink).

Show time-axis label

Select this check box to show the time-axis label on the scope display. This check box is not available if **Time-axis labels** is `None`.

Tunable (Simulink).

Maximize axes

Specify whether to display the scope in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

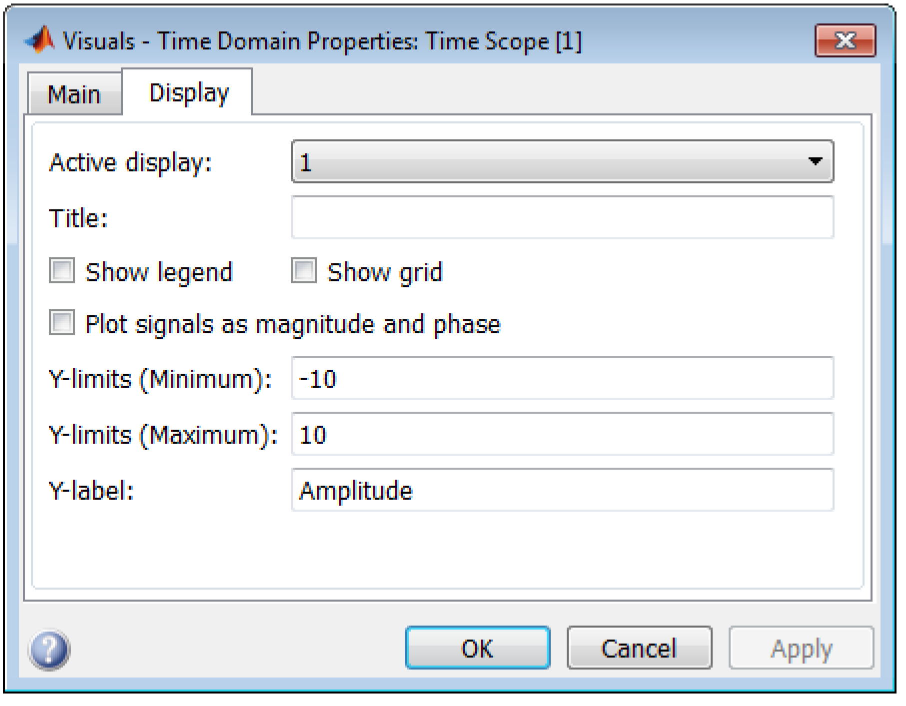
- `Auto` — In this mode, the axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- `On` — In this mode, the axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.
- `Off` — In this mode, none of the axes appear maximized.

This property is Tunable (Simulink).

The default setting is `Auto`.

Display Pane

The **Display** pane of the Visuals—Time Domain Properties dialog box appears as follows.



Active display

Specify the active display as an integer to get and set relevant properties. The number of a display corresponds to its column-wise placement index. Set this property to control which display has its axes colors, line properties, marker properties, and visibility changed. Tunable (Simulink)

When you use the Layout option to tile the window into multiple displays, the display highlighted in blue is referred to as the *active display*. The default setting is 1.

Title

Specify the active display title as a character vector. By default, the active display has no title. Tunable (Simulink).

Show legend

Select this check box to show the legend in the display. The channel legend displays a name for each channel of each input signal. When the legend appears, you can place it anywhere inside of the scope window. To turn off the legend, clear the **Show legend** check box.Tunable (Simulink)

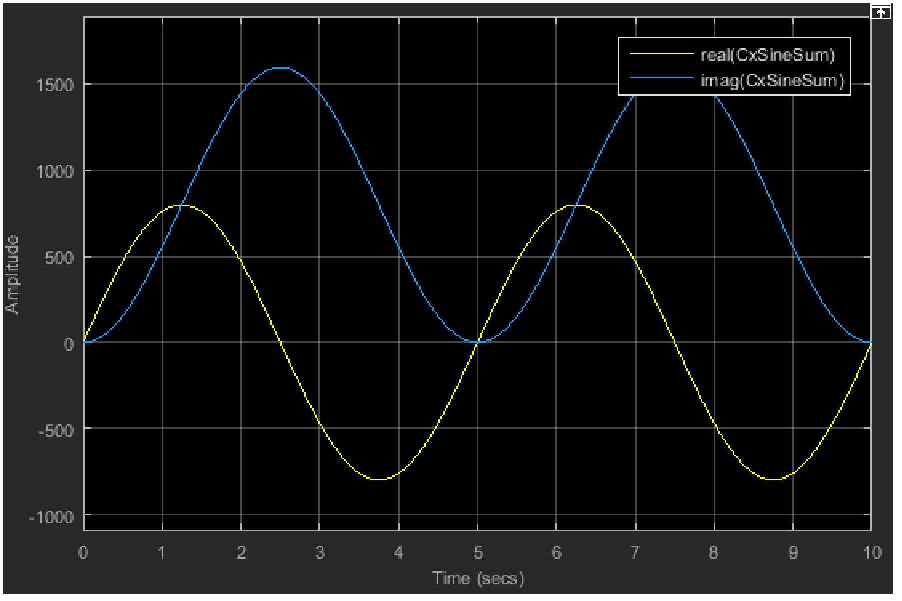
You can edit the name of any channel in the legend by double-clicking the current name and entering a new channel name. By default, if the signal has multiple channels, the scope uses an index number to identify each channel of that signal. To change the appearance of any channel of any input signal in the scope window, from the scope menu, select **View** > **Style**. The legend lets you modify what signals are shown. To show only one signal, click the signal name. To toggle a signal on/off, right-click the signal name.

Show grid

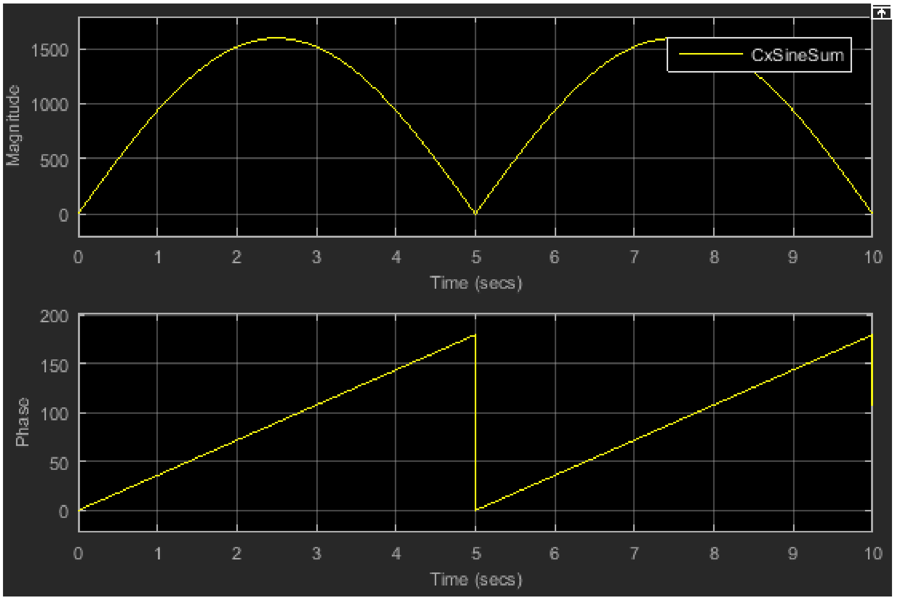
When you select this check box, a grid appears in the display of the scope figure. To hide the grid, clear this check box. Tunable (Simulink)

Plot signals as magnitude and phase

When you select this check box, the scope splits the display into a magnitude plot and a phase plot. By default, this check box is cleared. If the input signal has complex values, the scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes, as shown in the following figure.



Selecting this check box and clicking the **Apply** or **OK** button changes the display. The magnitude of the input signal appears on the top axes and its phase, in degrees, appears on the bottom axes. See the following figure.



This feature is useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box returns the absolute value of the signal for the magnitude. The phase is 0 degrees for nonnegative input and 180 degrees for negative input. Tunable (Simulink)

Y-limits (Minimum)

Specify the minimum value of the *y*-axis. Tunable (Simulink)

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this property always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a minimum value of -180 degrees.

Y-limits (Maximum)

Specify the maximum value of the *y*-axis. Tunable (Simulink)

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this property always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a maximum value of 180 degrees.

Y-label

Specify the text for the scope to display to the left of the *y*-axis. Tunable (Simulink)

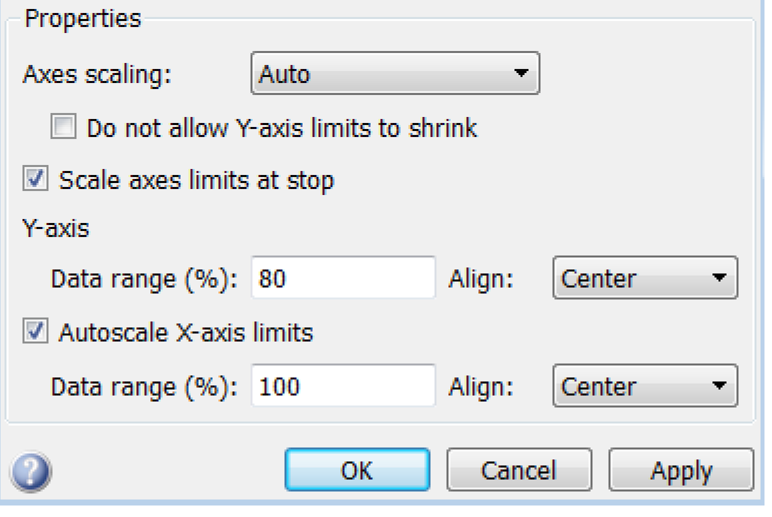
This property becomes invisible when you select the **Plot signal(s) as magnitude and phase** check box. When you enable that property, the *y*-axis label always appears as `Magnitude` on the top axes and `Phase` on the bottom axes.

Tools — Axes Scaling Properties

The Axes Scaling Properties: Time Scope dialog box provides you with the ability to automatically zoom in on and zoom out of your data, and to scale the axes of the Time Scope. In the Time Scope menu, select **Tools** > **Axes Scaling** > **Axes Scaling Properties** to open this dialog box.

Properties

The Tools—Axes Scaling Properties dialog box appears as follows.



Axes scaling

Specify when the scope automatically scales the axes. You can select one of the following options:

- `Manual` — When you select this option, the scope does not automatically scale the axes. You can manually scale the axes in any of the following ways:
  - Select **Tools** > **Axes Scaling Properties**.
  - Press one of the **Scale Axis Limits** toolbar buttons.
  - When the scope figure is the active window, press **Ctrl** and **A** simultaneously.
- `Auto` — When you select this option, the scope scales the axes as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** check box.
- `After N Updates` — Selecting this option causes the scope to scale the axes after a specified number of updates. This option is useful and more efficient when your scope display starts with one axis scale, but quickly reaches a different steady state axis scale. Selecting this option shows the **Number of updates** edit box.

By default, this property is set to `Auto`. This property is Tunable (Simulink).

Do not allow Y-axis limits to shrink

When you select this property, the *y*-axis is allowed only to grow during axes scaling operations. If you clear this check box, the *y*-axis or color limits may shrink during axes scaling operations.

This property appears only when you select `Auto` for the **Axis scaling** property. When you set the **Axes scaling** property to `Manual` or `After N Updates`, the *y*-axis or color limits are allowed to shrink. Tunable (Simulink).

Number of updates

Specify as a positive integer the number of updates after which to scale the axes. This property appears only when you select `After N Updates` for the **Axes scaling** property. Tunable (Simulink).

Scale axes limits at stop

Select this check box to scale the axes when the simulation stops. The *y*-axis is always scaled. The *x*-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

Y-axis Data range (%)

Set the percentage of the *y*-axis that the scope uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the Scope scales the *y*-axis limits such that your data uses the entire *y*-axis range. If you then set this property to `30`, the scope increases the *y*-axis range such that your data uses only 30% of the *y*-axis range. Tunable (Simulink).

Y-axis Align

Specify where the scope aligns your data along the *y*-axis when it scales the axes. You can select `Top`, `Center`, or `Bottom`. Tunable (Simulink).

Autoscale X-axis limits

Check this box to allow the scope to scale the *x*-axis limits when it scales the axes. If **Axes scaling** is set to `Auto`, checking **Autoscale X-axis limits** only scales the data currently within the axes, not the entire signal in the data buffer. If **Autoscale X-axis limits** is on and the resulting axis is greater than the span of the scope, trigger position markers will not be displayed. Triggers are controlled using the Trigger Measurements panel. Tunable (Simulink).

X-axis Data range (%)

Set the percentage of the *x*-axis that the scope uses to display the data when scaling the axes. Valid values are from 1 through 100. For example, if you set this property to `100`, the scope scales the *x*-axis limits such that your data uses the entire*x*-axis range. If you then set this property to `30`, the scope increases the *x*-axis range such that your data uses only 30% of the *x*-axis range. Use the *x*-axis **Align** property to specify data placement along the *x*-axis.

This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

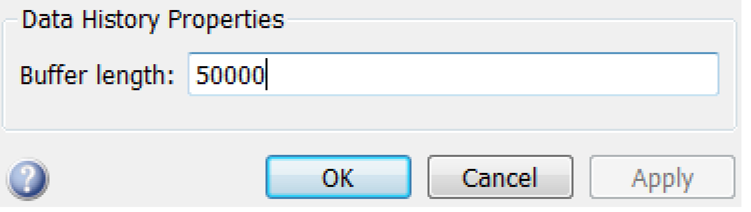
X-axis Align

Specify how the scope aligns your data along the *x*-axis: `Left`, `Center`, or `Right`. This property appears only when you select the **Scale X-axis limits** check box. Tunable (Simulink).

Sources — Streaming Properties

The Sources – Streaming Properties dialog box lets you control the number of input signal samples that Time Scope holds in memory. In the Time Scope menu, select **View** > **Data History Properties** to open this dialog box.

Data History Properties



Buffer length

Specify the size of the buffer that the scope holds in its memory cache. Memory is limited by available memory on your system. If your signal has *M* rows of data and *N* data points in each row, *M* x *N* is the number of data points per time step. Multiply this result by the number of time steps for your model to obtain the required buffer length. For example, if you have 10 rows of data with each row having 100 data points and your run will be 10 time steps, you should enter 10,000 (which is 10 x 100 x 10) as the buffer length.

The default setting is `50000`.

Examples

The first four examples show how to use the Time Scope object to view a variety of input signals in the time domain. The remaining two examples show how to use the Measurements panels in the Time Scope UI to glean information about the input signals.

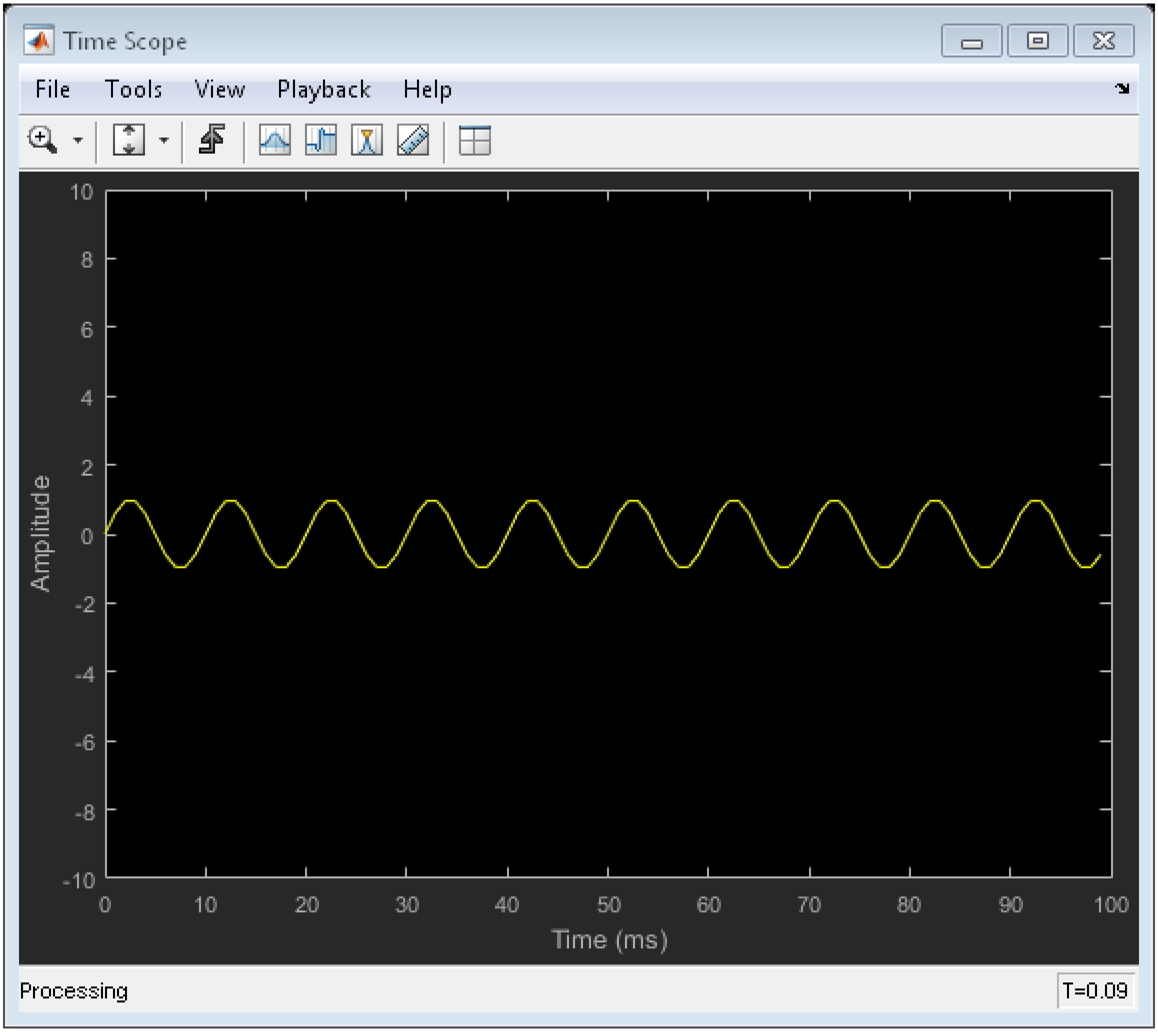
- "Display Simple Sine Wave Input Signal" on page 4-1761
- "View Sine Wave Input Signals at Different Sample Rates and Offsets" on page 4-1764
- "Display Complex-Valued Input Signal" on page 4-1765
- "Display Input Signal of Changing Size" on page 4-1769
- "Use Bilevel Measurements Panel with Clock Input Signal" on page 4-1772
- "Find Heart Rate Using Peak Finder Panel with ECG Input Signal" on page 4-1775

Display Simple Sine Wave Input Signal

Display a sine wave on a time scope.

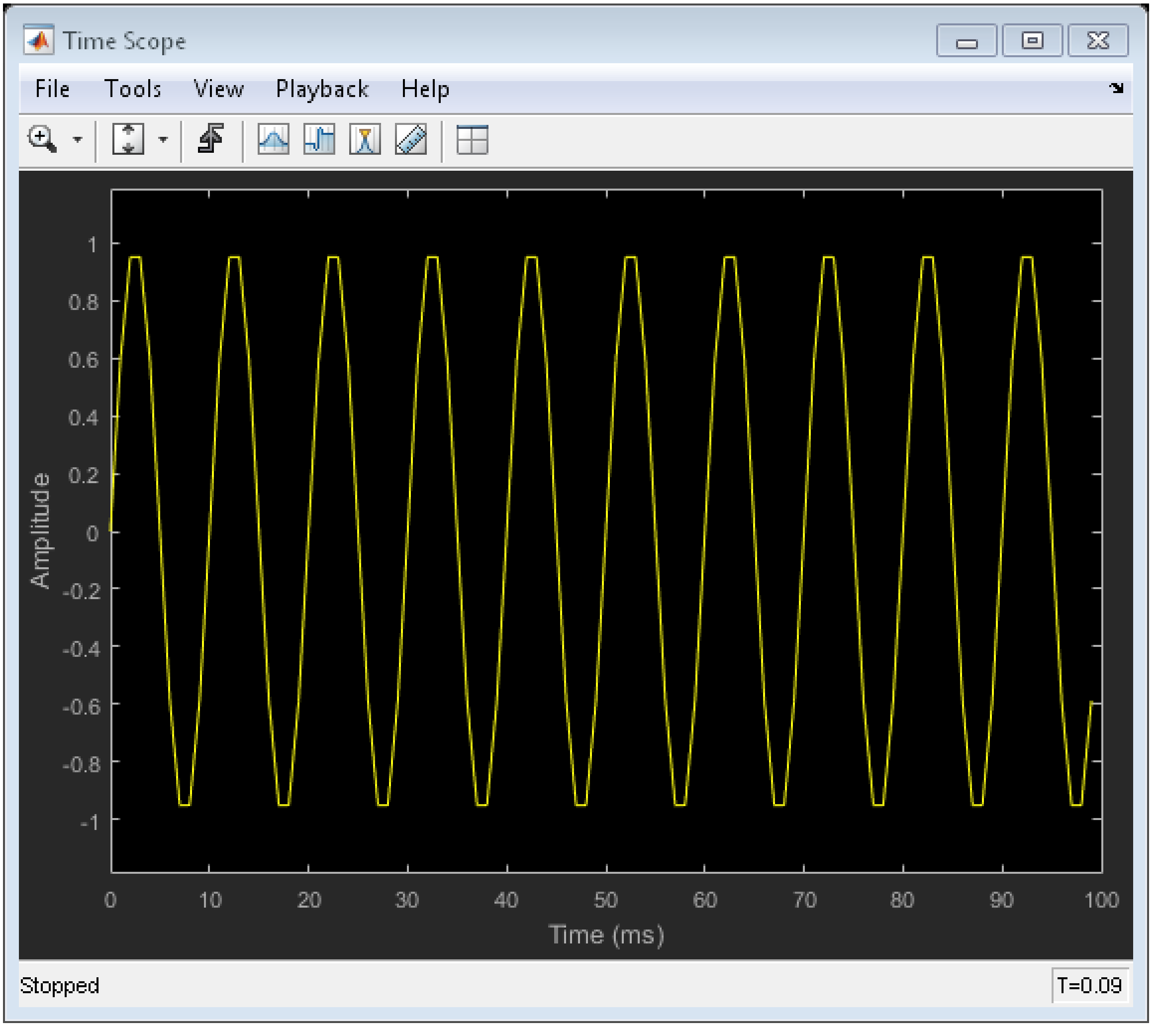
Create `dsp.SineWave` and `dsp.TimeScope` objects. Run the scope to display the signal

```
sine = dsp.SineWave('Frequency',100,'SampleRate',1000);
sine.SamplesPerFrame = 10;
scope = dsp.TimeScope('SampleRate',sine.SampleRate,'TimeSpan',0.1);
for ii = 1:10
     x = sine();
     scope(x);
end
```



Run the `release` method to allow changes to property values and input characteristics. The scope automatically scales the axes.

```
release(scope)
```



Close the Time Scope window and clear the variables.

```
clear scope sine x;
```

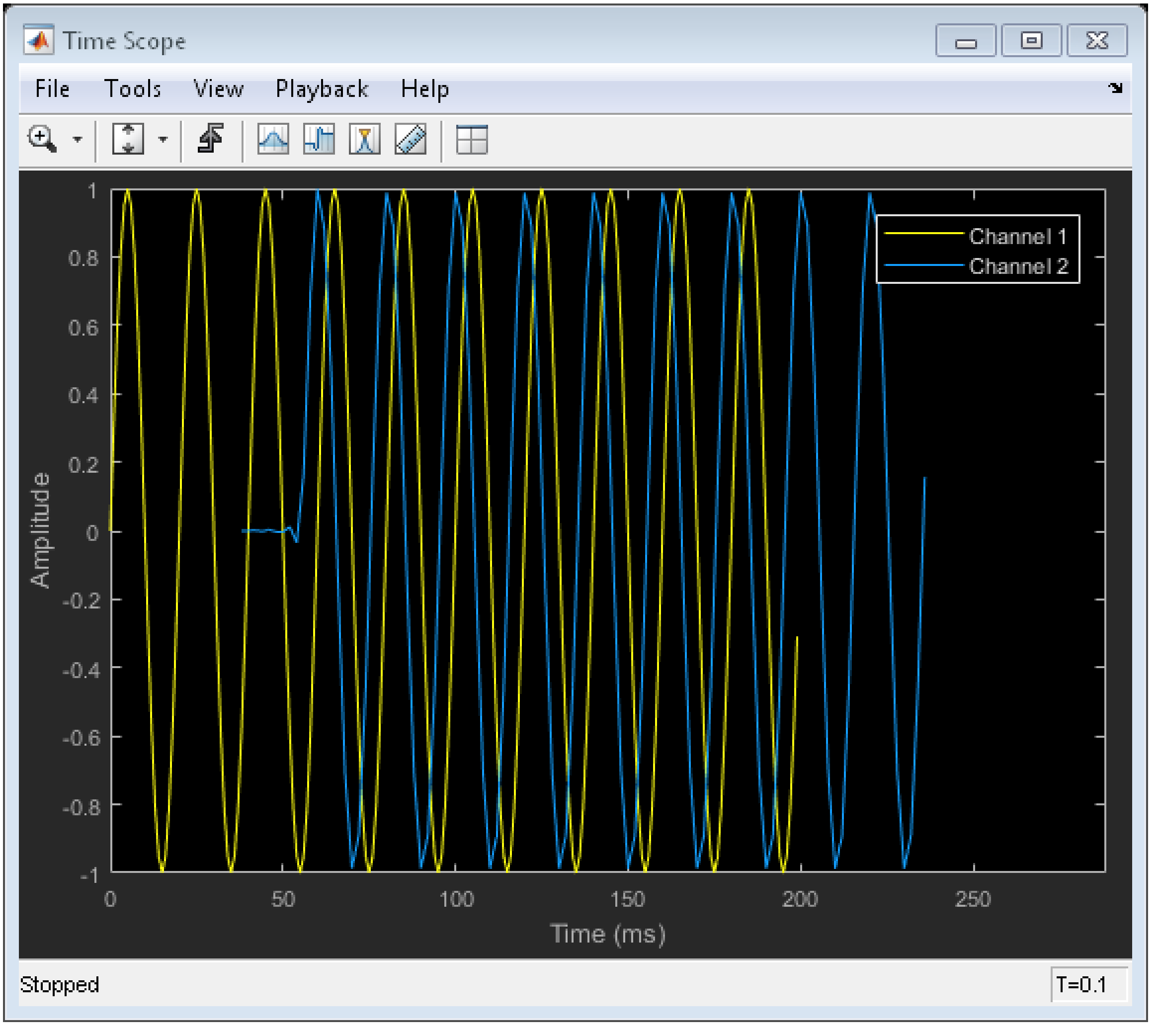
View Sine Wave Input Signals at Different Sample Rates and Offsets

Create a `dsp.SineWave` with a 1000 Hz sampling frequency. Create a `dsp.FIRDecimator` object to decimate the sine wave by 2. Create a `dsp.TimeScope` object with two input ports.

```
Fs = 1000;  % Sampling frequency
sine = dsp.SineWave('Frequency',50,...
   'SampleRate',Fs, ...
   'SamplesPerFrame',100);
decimate = dsp.FIRDecimator; % To decimate sine by 2
scope = dsp.TimeScope(2,[Fs Fs/2], ...
   'TimeDisplayOffset',[0 38/Fs], ...
   'TimeSpan',0.25, ...
   'YLimits',[-1 1], ...
   'ShowLegend', true);
```

Call the `dsp.SineWave` object to create a sine wave signal. Use the `dsp.FIRDecimator` object to create a second signal that equals the original signal, but decimated by a factor of 2. Display the signals by calling the `dsp.TimeScope` object.

```
for ii = 1:2
     xsine = sine();
     xdec = decimate(xsine);
     scope(xsine,xdec)
end
release(scope)
```



Close the Time Scope window and clear the variables.

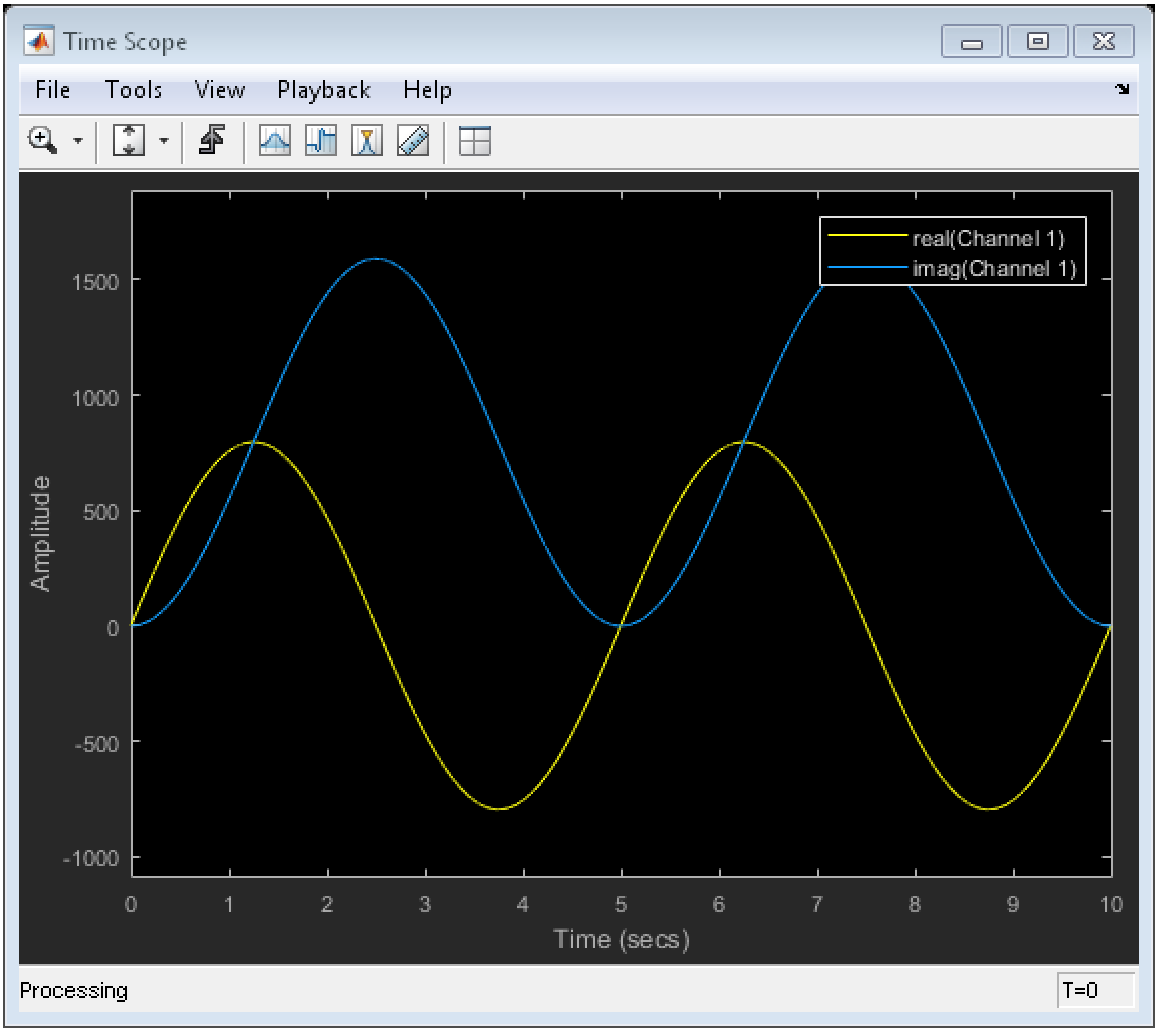
```
clear scope Fs sine decimate ii xsine xdec
```

Display Complex-Valued Input Signal

Use the time scope to show complex signals.

Create a vector representing a complex-valued sinusoidal signal, and create a `dsp.TimeScope` object. Call the scope to display the signal.

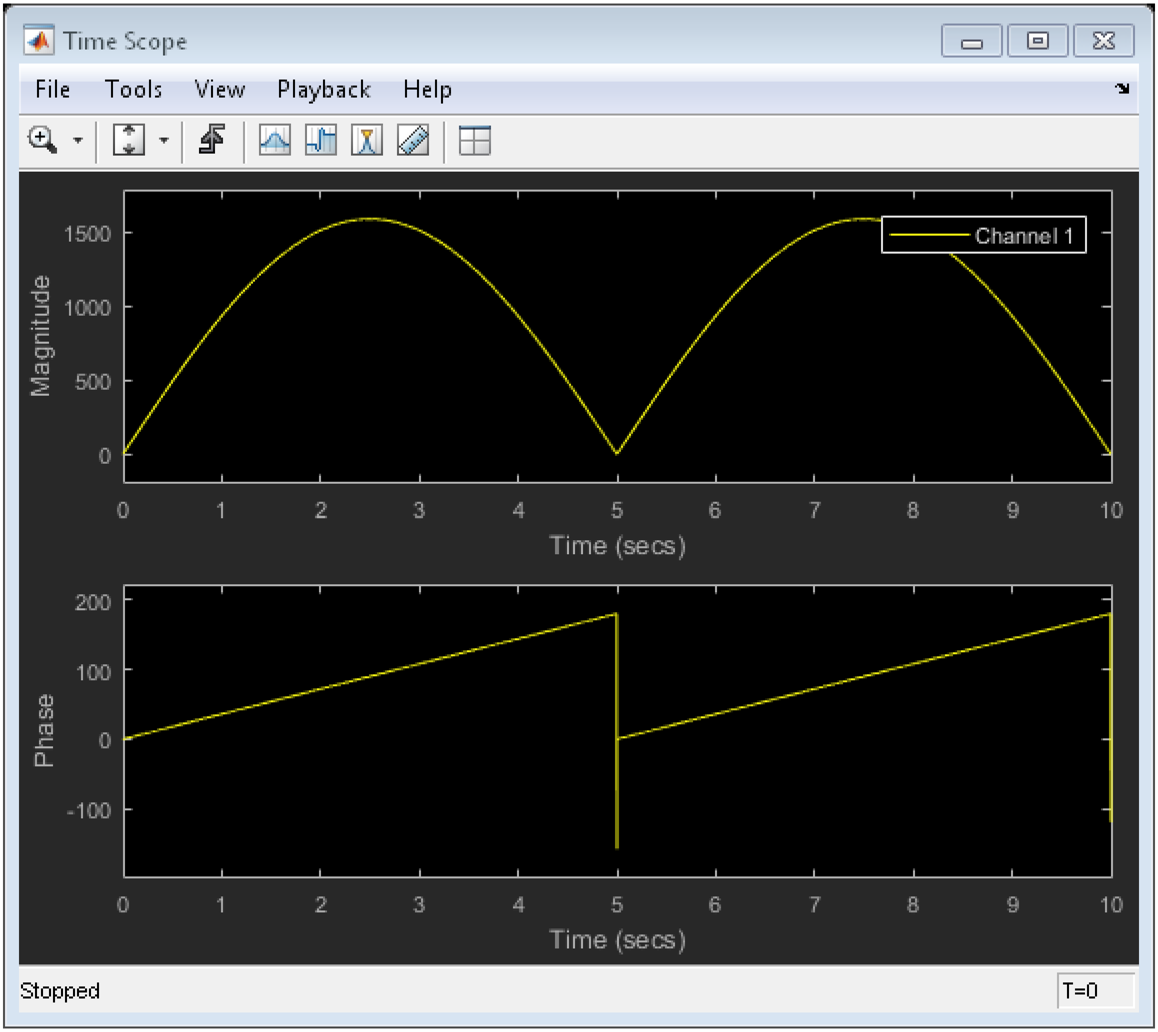
```
fs = 1000;
t = (0:1/fs:10)';
CxSine = cos(2*pi*0.2*t) + 1i*sin(2*pi*0.2*t);
CxSineSum = cumsum(CxSine);
scope = dsp.TimeScope(1,fs,'TimeSpanSource','Auto','ShowLegend',1);
scope(CxSineSum);
scope.AxesScaling = 'Auto';
```



By default, when the input is a complex-valued signal, Time Scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes within the same active display.

Change the `PlotAsMagnitudePhase` property to `true` and call `release`.

```
scope.PlotAsMagnitudePhase = true;
release(scope)
```



Time Scope now plots the magnitude and phase of the input signal on two separate axes within the same active display. The top axes display magnitude and the bottom axes display the phase, in degrees.

Close the Time Scope window and remove the variables you created.

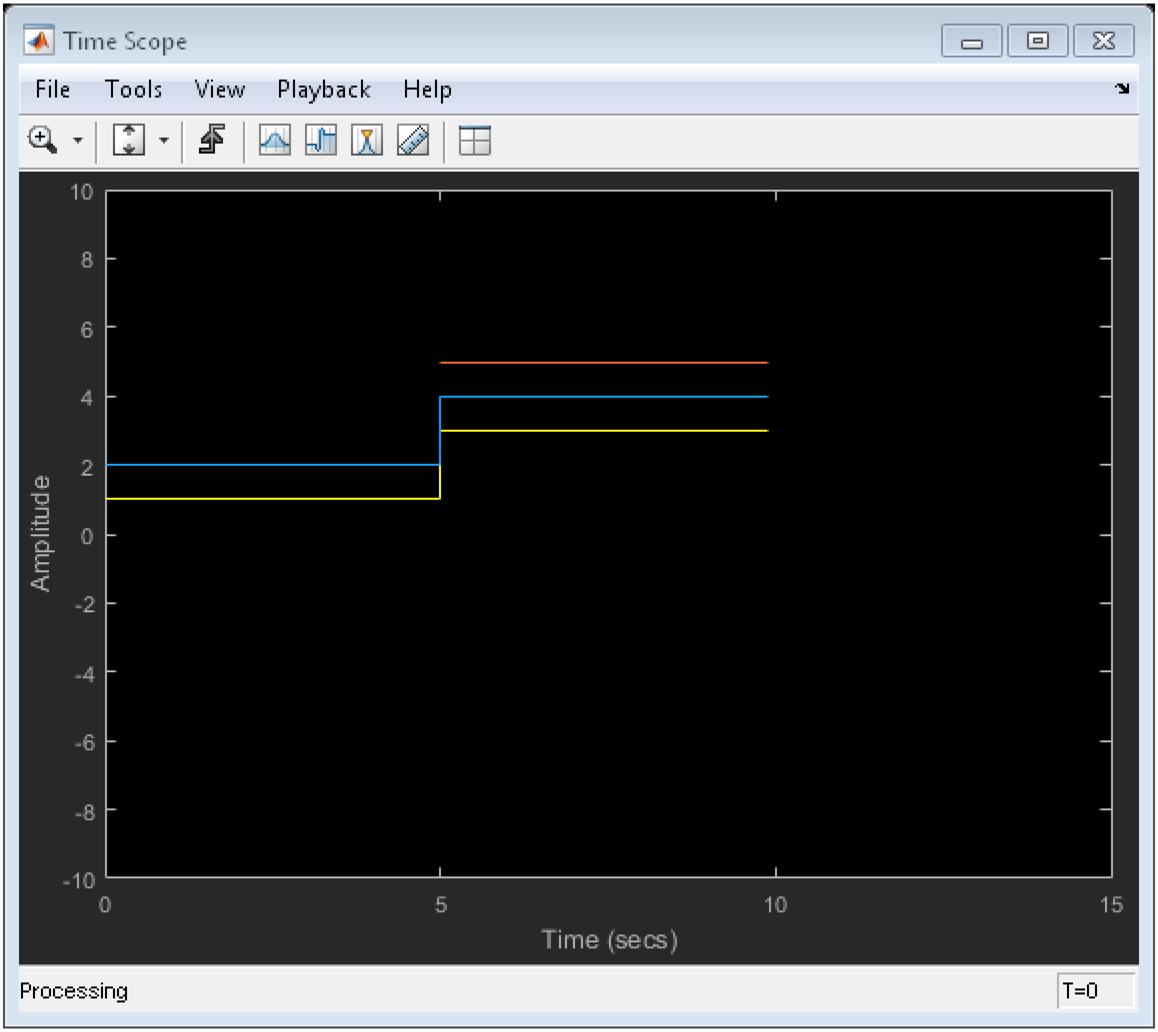
```
clear scope fs t CxSine CxSineSum
```

Display Input Signal of Changing Size

Use a Time Scope to display a variable-size signal.

Create a vector that represents a two-channel constant signal. Create another vector that represents a three-channel constant signal. Construct a `dsp.TimeScope` object with two input ports. Call the scope to display the signal.

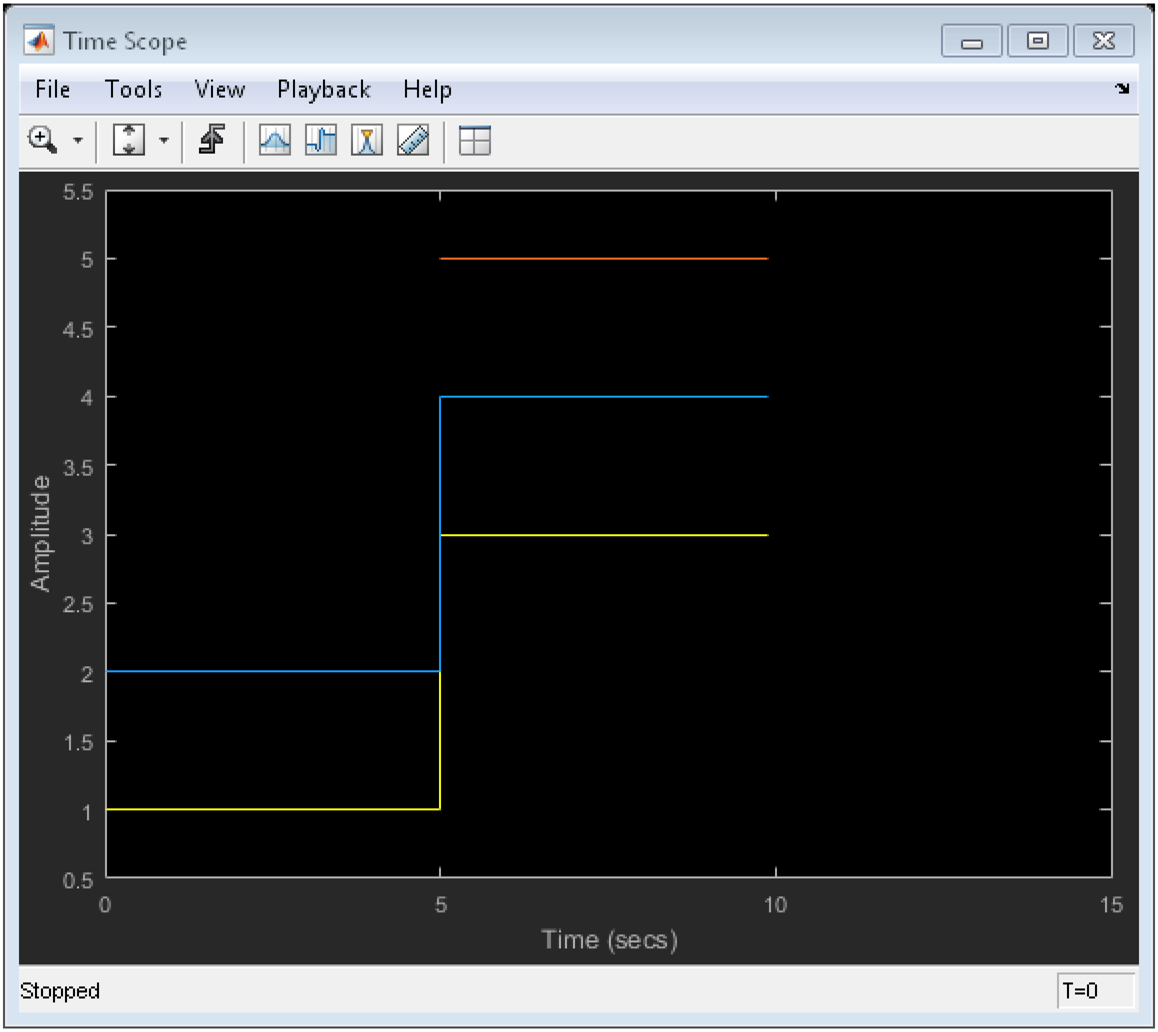
```
fs = 10;
sigdim2 = [ones(5*fs,1) 1+ones(5*fs,1)];                    % 2-dim 0-5 s
sigdim3 = [2+ones(5*fs,1) 3+ones(5*fs,1) 4+ones(5*fs,1)];  % 3-dim 5-10 s
scope = dsp.TimeScope(2,fs,'TimeSpanSource','Property');
scope.PlotType = 'Stairs';
scope.TimeSpanOverrunAction = 'Scroll';
scope.TimeDisplayOffset = [0 0 5];
scope([sigdim2; sigdim3(:,1:2)], sigdim3(:,3));
```



In this example, the size of the input signal to the Time Scope block changes as the simulation progresses. When the simulation time is less than 5 seconds, Time Scope plots only the two-channel signal, `sigdim2`. After 5 seconds, Time Scope also plots the three-channel signal, `sigdim3`.

Run the `release` method to enable changes to property values and input characteristics. The scope automatically scales the axes.

```
release(scope)
```



Close the Time Scope window and remove the variables you created from the workspace.

```
clear scope fs sigdim2 sigdim3
```

Use Bilevel Measurements Panel with Clock Input Signal

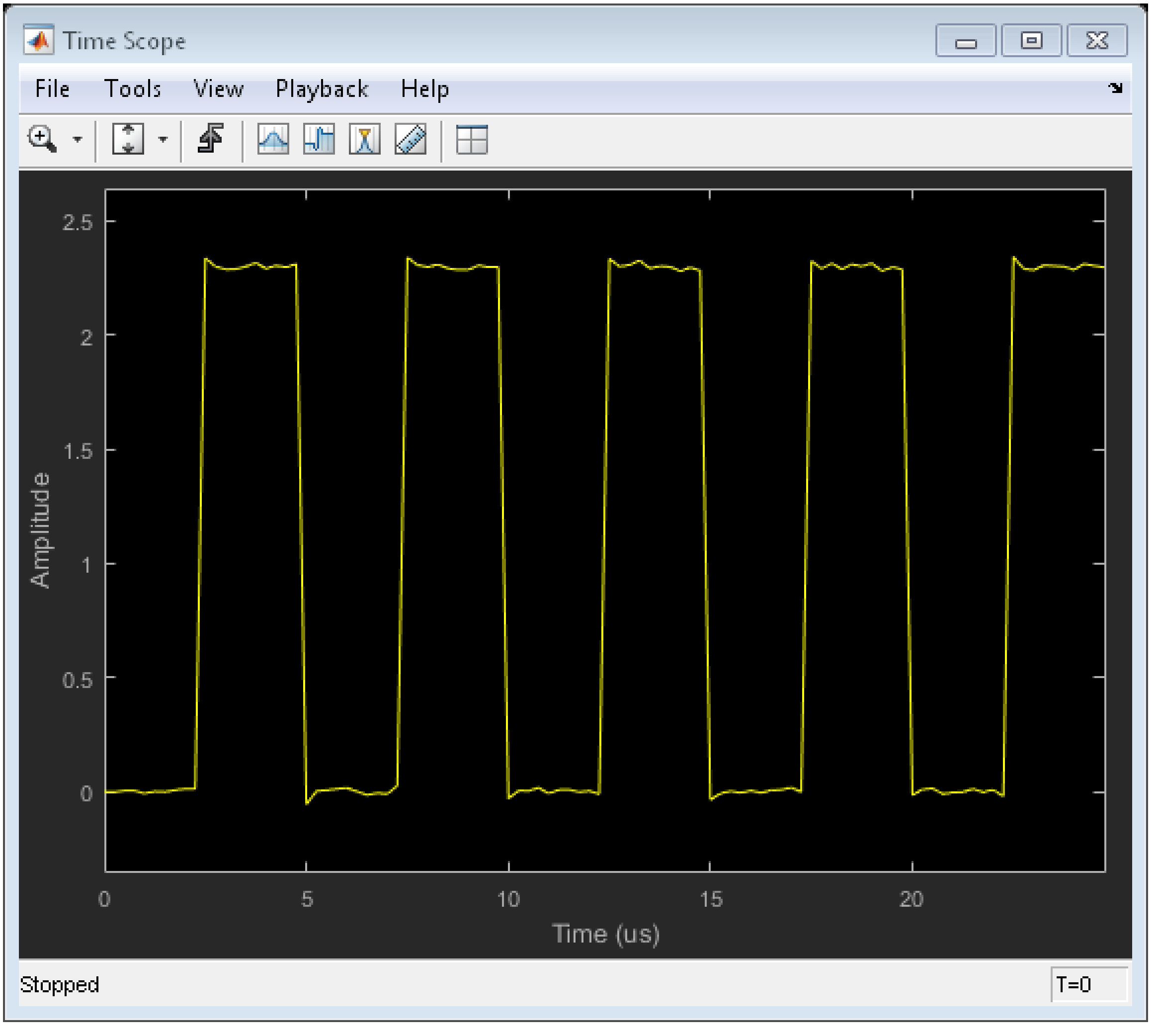
Create and Display Clock Input Signal

Load the clock data, `x` and `t`. Find the sample time, `ts`.

```
load clockex
ts = t(2)-t(1);
```

Create a `dsp.TimeScope` object and call the object to display the signal. To autoscale the axes and enable changes to property values and input characteristics, call `release`.

```
scope = dsp.TimeScope(1,1/ts,'TimeSpanSource','Auto');
scope(x);
release(scope)
```

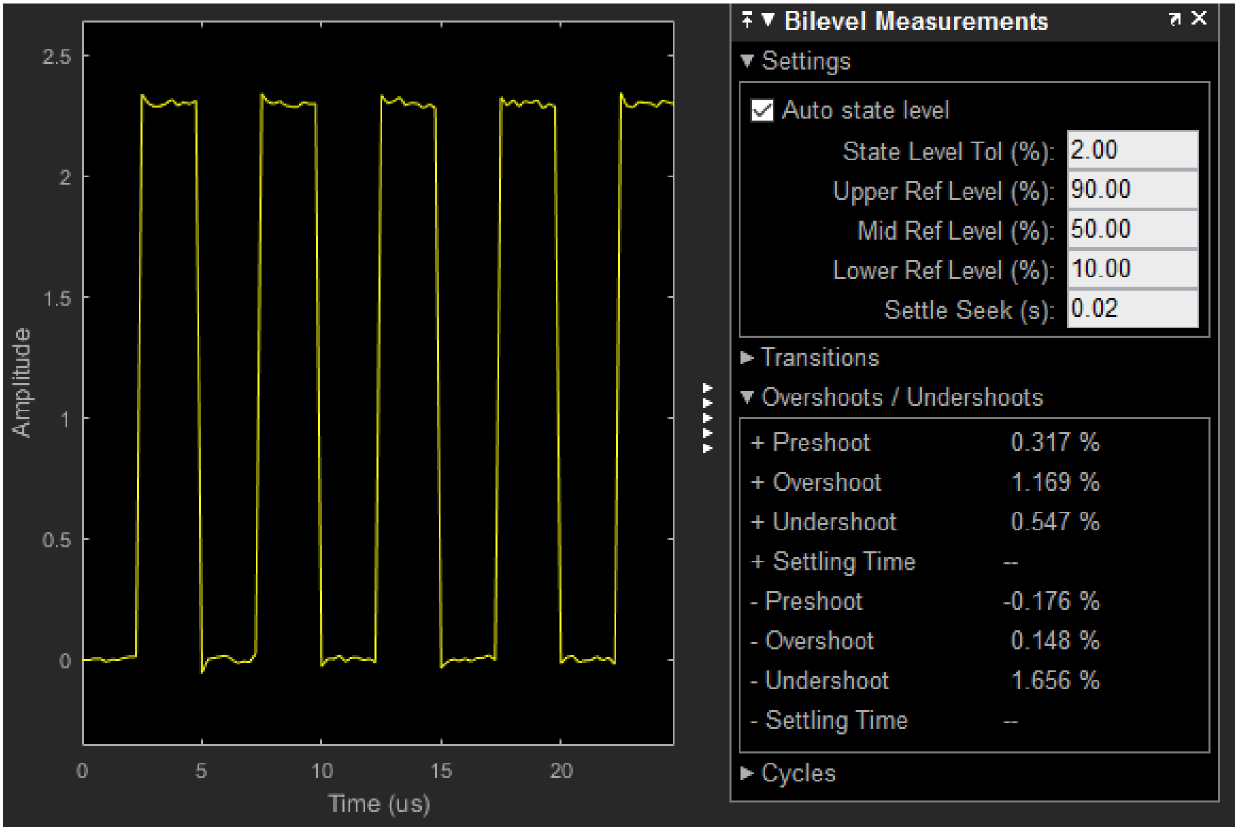


Use Bilevel Measurements Panel to Find Settling Time

1. From the Time Scope menu, select **Tools > Measurements > Bilevel Measurements**.

2. Expand the **Settings** pane and the **Overshoots / Undershoots** pane.

Initially, the Time Scope does not display the rising edge **Settling Time** parameter. This absence occurs because the default value of the **Settle Seek** parameter is longer than the entire simulation duration.



3. In the **Settle Seek** box, enter `2e-6` and press **Enter**.

Time Scope now displays a rising edge **Settling Time** value of `118.392` ns.

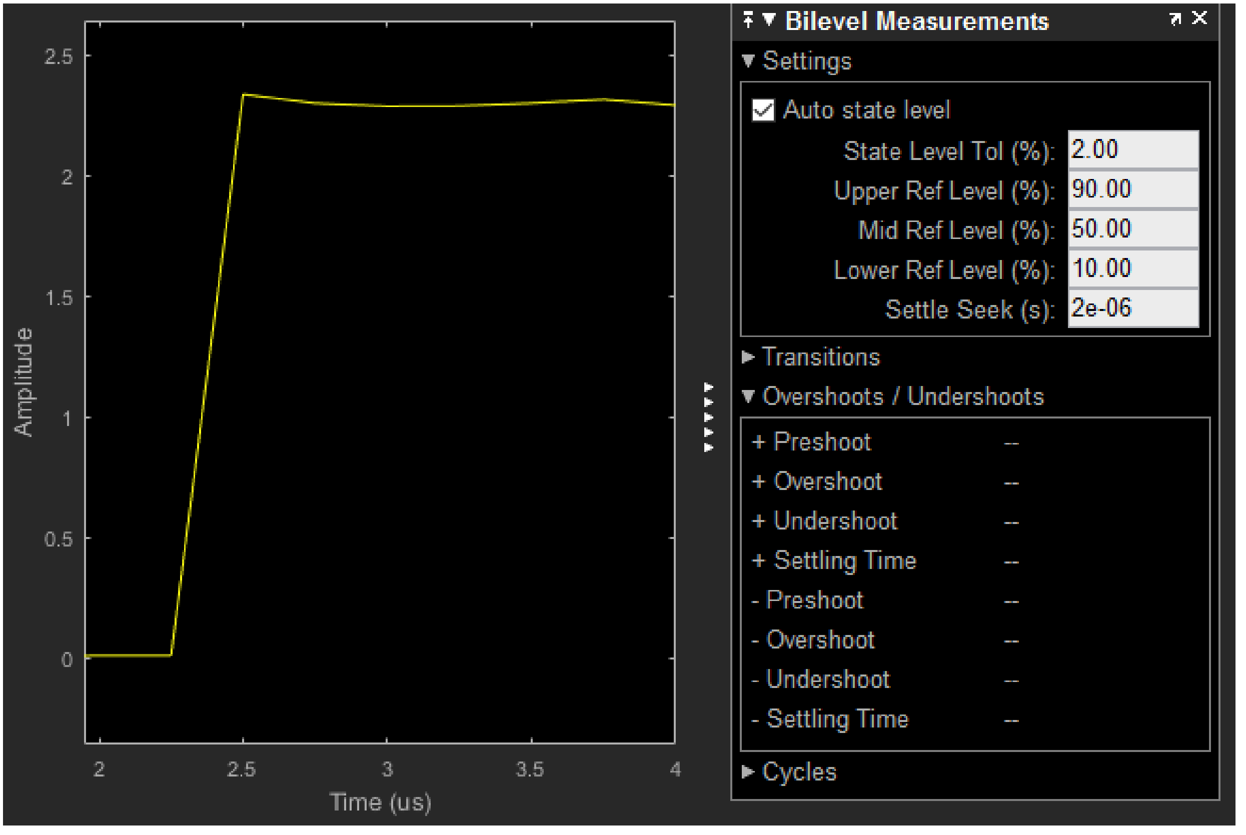
This settling time value is actually the statistical average of the settling times for all five rising edges. To show the settling time for only one rising edge, you can zoom in on that transition.

4. In the Time Scope toolbar, click the arrow next to the Zoom button, and then click the Zoom X button.

5. Click the display near a value of 2 microseconds on the time axis.

6. Drag your cursor right and release it near a value of 4 microseconds on the time axis.

Time Scope updates the rising edge **Settling Time** value to reflect the new time window.



7. Close the Time Scope and remove the variables you created from the workspace.

```
clear scope x t ts
```

Find Heart Rate Using Peak Finder Panel with ECG Input Signal

Use Peak Finder panel of the Time Scope to measure a heart rate.

Create and Display ECG Signal

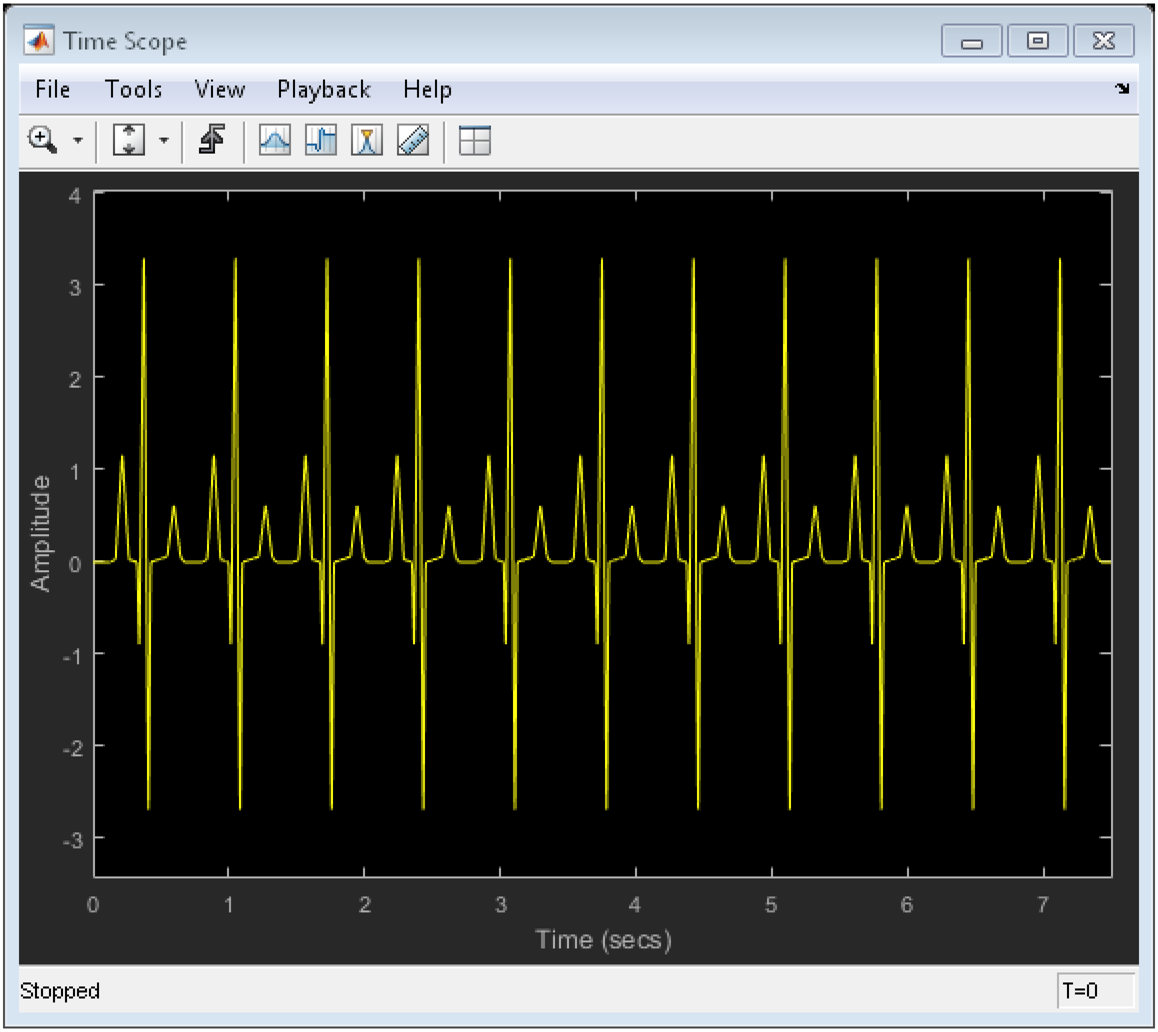
Create the electrocardiogram (ECG) signal. The custom `ecg` fuction helps generate the heartbeat signal.

```
function x = ecg(L)
a0 = [0,  1, 40,  1,   0, -34, 118, -99,   0,   2,  21,   2,   0,   0,   0];
d0 = [0, 27, 59, 91, 131, 141, 163, 185, 195, 275, 307, 339, 357, 390, 440];
a = a0 / max(a0);
d = round(d0 * L / d0(15));
d(15) = L;
for i = 1:14
    m = d(i) : d(i+1) - 1;
    slope = (a(i+1) - a(i)) / (d(i+1) - d(i));
    x(m+1) = a(i) + slope * (m - d(i));
end
```

```
x1 = 3.5*ecg(2700).';
y1 = sgolayfilt(kron(ones(1,13),x1),0,21);
n = (1:30000)';
del = round(2700*rand(1));
mhb = y1(n + del);
ts = 0.00025;
```

Create a `dsp.TimeScope` object and call the object to display the signal. To autoscale the axes and enable changes to property values and input characteristics, call `release`.

```
scope = dsp.TimeScope(1,1/ts,'TimeSpanSource','Auto');
scope(mhb);
release(scope)
```

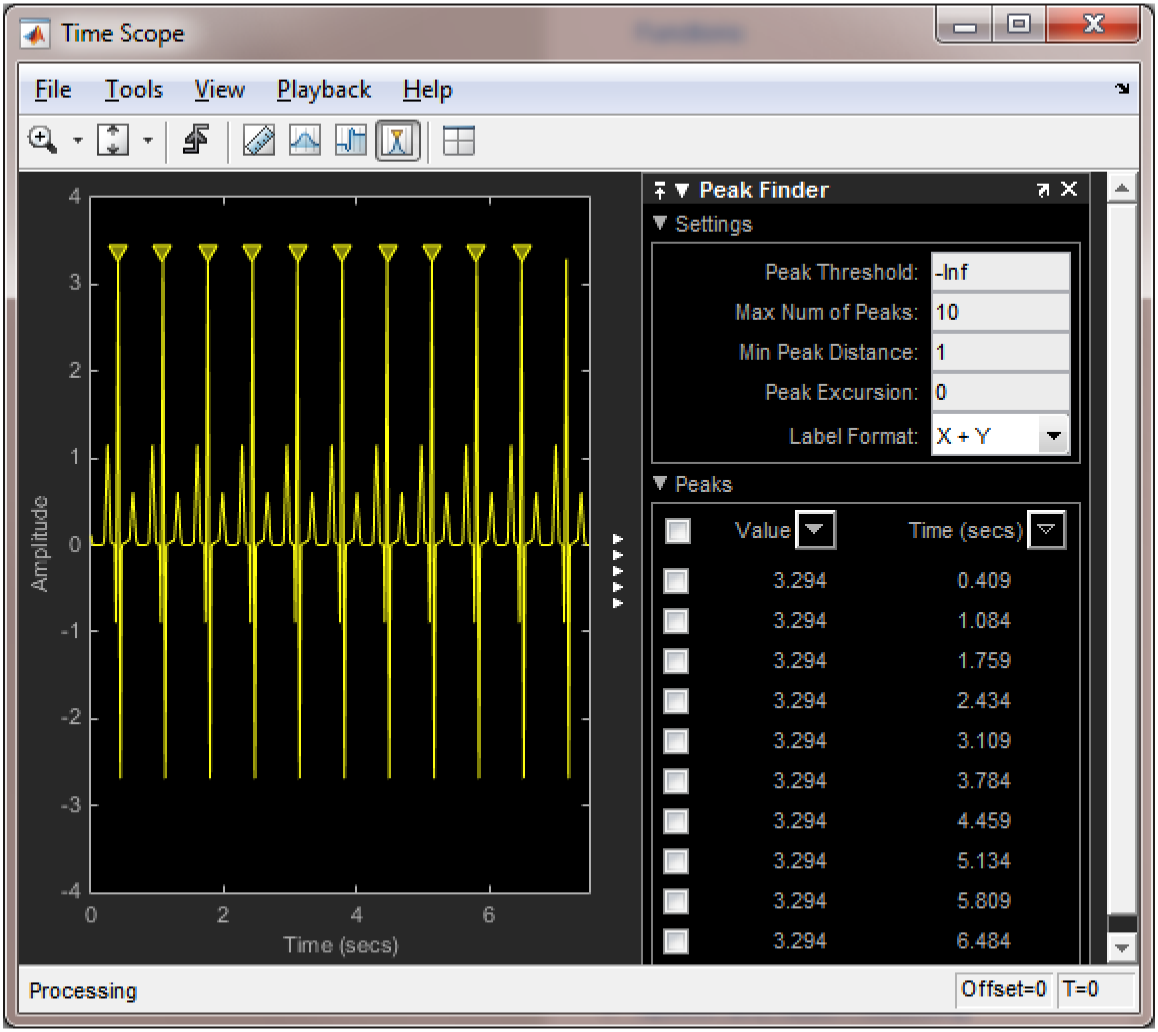


Find Heart Rate

Use the Peak Finder measurements to measure the time between heart beats.

1 In the Time Scope menu, select **Tools > Measurements > Peak Finder**.
2 Expand the **Settings** pane.
3 In the **Max Num of Peaks** property, enter 10 and press **Enter**.

In the **Peaks** pane, Time Scope displays a list of ten peak amplitude values and the times at which they occur.



The list of peak values shows a constant time difference of 0.675 second between each heartbeat. Based on the following equation, the heart rate of this ECG signal is about 89 beats per minute.

$$\frac{60\ \mathrm{s/min}}{0.675\ \mathrm{s/beat}} = 88.89\ \mathrm{bpm}$$

Close the Time Scope window and remove the variables you created from the workspace.

```
clear scope x1 y1 n del mhb ts
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supports MEX code generation by treating the calls to the object as extrinsic. Does not support code generation for standalone applications.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`Time Scope` | dsp.SpectrumAnalyzer | `sptool`

Introduced in R2012a

hide

System object: dsp.TimeScope
Package: dsp

Hide time scope window

Syntax

```
hide(scope)
```

Description

`hide(scope)` hides the time scope window, associated with System object, `scope`.

See Also

dsp.TimeScope.show

reset

System object: dsp.TimeScope
Package: dsp

Reset internal states of time scope object

Syntax

```
reset(scope)
```

Description

`reset(scope)` sets the internal states of the `TimeScope` object `scope` to their initial values.

You should call the `reset` method after calling the `step` method when you want to clear the Time Scope figure displays, prior to releasing system resources. This action enables you to start a simulation from the beginning. When you call the `reset` method, the displays will become blank again. In this sense, its functionality is similar to that of the MATLAB `clf` function. Do not call the `reset` method after calling the `release` method.

Examples

View a sine wave on the time scope. When you finish the simulation, pause for 5 seconds and then reset the screen. Then, view a different sine wave on the same time scope.

```
sinSignal1 = dsp.SineWave('Frequency', 100, 'SampleRate', 1000, 'SamplesPerFrame', 10);
sinSignal2 = dsp.SineWave('Frequency', 50, 'SampleRate', 1000, 'SamplesPerFrame', 10);

scope = dsp.TimeScope('SampleRate', sinSignal1.SampleRate,'TimeSpan', 0.1);
for ii = 1:10
     x = step(sinSignal1);
     step(singSignal2, x);
end

% Pause and then reset internal states
```

```
pause(5);
reset(scope);

for ii = 1:10
     x = step(sinSignal1);
     step(sinSignal2, x);
end

release(scope);
```

Algorithms

In operation, the `reset` method is similar to a consecutive execution of the `mdlTerminate` function and the `mdlInitializeConditions` function.

See Also

dsp.TimeScope

show

System object: dsp.TimeScope
Package: dsp

Make time scope window visible

Syntax

```
show(scope)
```

Description

`show(scope)` makes the time scope window, associated with System object, `scope`, visible.

See Also

dsp.TimeScope.hide

step

System object: dsp.TimeScope
Package: dsp

Display signal in time scope figure

Syntax

```
step(scope,signal)
step(scope,signal1,signal2,...,signalN)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`step(scope,signal)` displays the signal, `signal`, in the time scope figure.

`step(scope,signal1,signal2,...,signalN)` displays the signals `signal1`, `signal2`,...,`signalN` in the time scope figure when you set the `NumInputPorts` property to `N`. In this case, `signal1`, `signal2`,...,`signalN` can have different data types and dimensions.

dsp.TransferFunctionEstimator System object

Package: dsp

Estimate transfer function

Description

The `dsp.TransferFunctionEstimator` computes the transfer function of a system, using the Welch algorithm and the Periodogram method.

To implement the transfer function estimation object:

1. Define and set up your transfer function estimator object. See "Construction" on page 4-1785.
2. Call step to implement the estimator according to the properties of `dsp.TransferFunctionEstimator`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`tfe = dsp.TransferFunctionEstimator` returns a System object, `tfe`, that computes the transfer function of real or complex signals. This System object uses the periodogram method and Welch's averaged, modified periodogram method.

`tfe = dsp.TransferFunctionEstimator('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a `Transfer Function Estimator` System object, `tfe`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

SpectralAverages

Number of spectral averages

Specify the number of spectral averages as a positive, integer scalar. The Transfer Function Estimator computes the current estimate by averaging the last `N` estimates. `N` is the number of spectral averages defined in the `SpectralAverages` property. The default value is `8`.

FFTLengthSource

Source of the FFT length value

Specify the source of the FFT length value as one of `'Auto' | 'Property'`. The default value is `'Auto'`. If you set this property to `'Auto'`, the Transfer function Estimator sets the FFT length to the input frame size. If you set this property to `'Property'`, then you specify the number of FFT points using the `FFTLength` property.

FFTLength

FFT Length

Specify the length of the FFT that the Transfer Function Estimator uses to compute spectral estimates as a positive, integer scalar. This property applies when you set the `FFTLengthSource` property to `'Property'`. The default value is `128`.

Window

Window function

Specify a window function for the Transfer Function estimator as one of `'Rectangular' | 'Chebyshev' | 'Flat Top' | 'Hamming' | 'Hann' | 'Kaiser'`. The default value is `'Hann'`.

SidelobeAttenuation

Side lobe attenuation of window

Specify the side lobe attenuation of the window as a real, positive scalar, in decibels (dB). This property applies when you set the `Window` property to `'Chebyshev'` or `'Kaiser'`. The default value is `60 dB`.

FrequencyRange

Frequency range of the transfer function estimate

Specify the frequency range of the transfer function estimator as one of `'twosided' | 'onesided' | 'centered'`.

If you set the `FrequencyRange` to `'onesided'`, the transfer function estimator computes the onesided transfer function of real input signals, `x` and `y`. If the FFT length, NFFT, is even, the length of the transfer function estimate is `NFFT/2+1` and is computed over the interval `[0,SampleRate/2]`. If NFFT is odd, the length of the transfer function estimate is equal to `(NFFT+1)/2` and the interval is `[0,SampleRate/2]`.

If `FrequencyRange` is set to `'twosided'`, the transfer function estimator computes the twosided transfer function of complex or real input signals, `x` and `y`. The length of the transfer function estimate is equal to NFFT and is computed over `[0, SampleRate]`.

If you set the `FrequencyRange` to `'centered'`, the transfer function estimator computes the centered twosided transfer function of complex or real input signals, `x` and `y`. The length of the transfer function estimate is equal to NFFT and it is computed over `[-SampleRate/2, SampleRate/2]` for even lengths, and `[-SampleRate/2, SampleRate/2]` for odd lengths. The default value is `'Twosided'`.

OutputCoherence

Magnitude squared coherence estimate

Specify `true` to compute and output the magnitude squared coherence estimate using Welch's averaged, modified periodogram method. The magnitude squared coherence estimate has values between 0 and 1 that indicate the correspondence at each frequency between two input signals. If you specify `false`, the magnitude squared coherence estimate is not computed. The default value is `false`.

Methods

getFrequencyVector	Get vector of frequencies at which transfer function is estimated
getRBW	Get resolution bandwidth of transfer function

reset	Reset internal states of transfer function estimator
step	Estimate transfer function of system

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Estimate transfer function of a system represented by an order-64 FIR filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

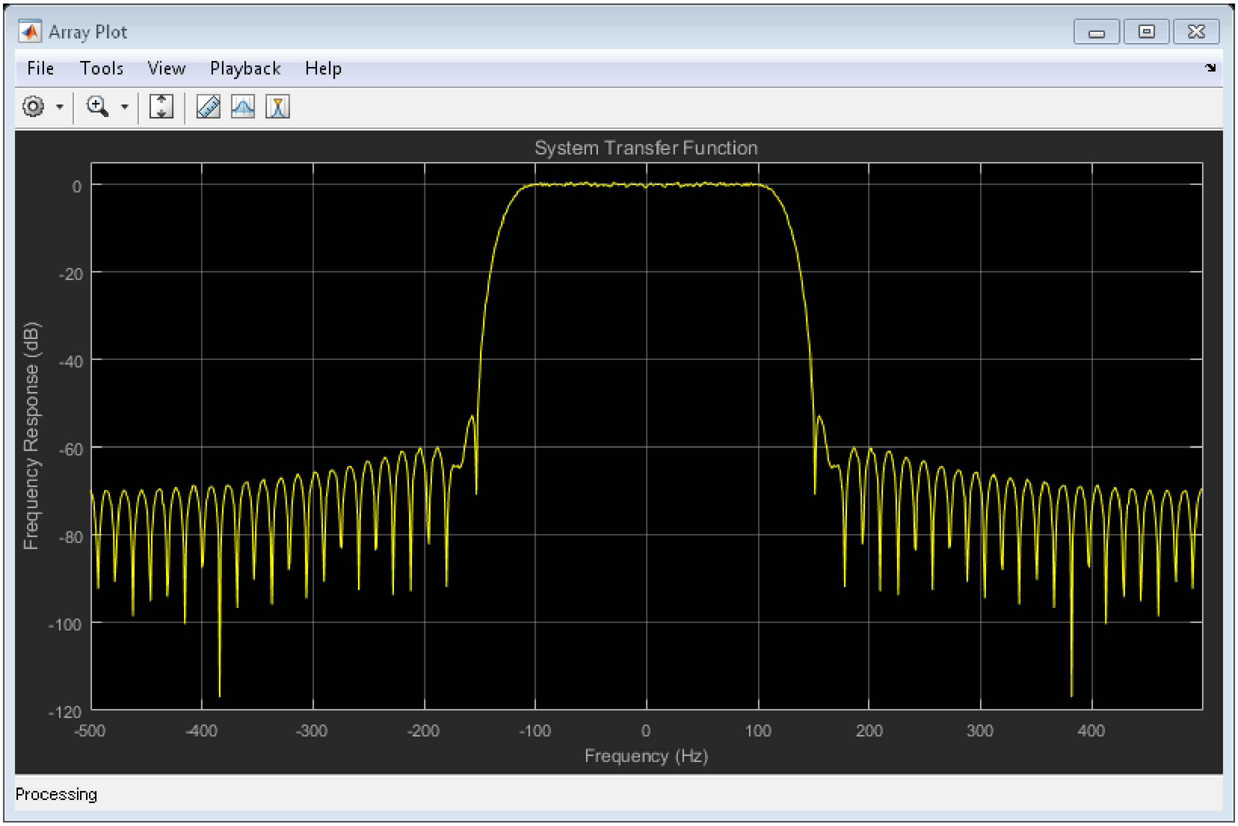
Generate a sine wave. Use the dsp.TransferFunctionEstimator System object to estimate the system transfer function, and the dsp.ArrayPlot System object to display it.

```
sin = dsp.SineWave('Frequency',100, 'SampleRate', 1000);
sin.SamplesPerFrame = 1000;
tfe = dsp.TransferFunctionEstimator('FrequencyRange','centered');
aplot = dsp.ArrayPlot('PlotType','Line','XOffset',-500,'YLimits',...
        [-120 5],'YLabel','Frequency Response (dB)',...
        'XLabel','Frequency (Hz)',...
        'Title','System Transfer Function');
```

Create an FIR Filter System object of order 64 and (normalized) cutoff frequency of 1/4. Add random noise to the sine wave. Step through the System objects to obtain the data streams, and plot the log of the magnitude of the transfer function.

```
firFilt = dsp.FIRFilter('Numerator',fir1(64,1/4));
for ii = 1:100
x = sin() + 0.05*randn(1000,1);
```

```
y = firFilt(x);
Txy = tfe(x,y);
aplot(20*log10(abs(Txy)))
end
```



Algorithms

Given two signals `x` and `y` as inputs. We first window the two inputs, and scale them by the window power. We then take FFT of the signals, calling them X and Y. This is followed by calculating P_{xx} which is the square magnitude of the FFT, X, and P_{yx} which is X multiplied by the conjugate of Y. The transfer function estimate is calculated by dividing P_{yx} by P_{xx}.

For further information refer to the "Algorithms" on page 2-1592 section in Spectrum Analyzer, which uses the same algorithm.

The magnitude squared coherence, C_{xy}, is defined as

$$C_{xy} = \frac{\left(abs\left(P_{xy}\right).\hat{}2\right)}{\left(P_{xx}.*P_{yy}\right)}$$

where, x and y are input signals. P_{xx} and P_{yy} are the power spectral density (PSD) estimates of x and y, respectively. P_{xy} is the cross power spectral density (CPSD) estimate of x and y.

References

[1] Hayes, Monson H. *Statistical Digital Signal Processing and Modeling*. Hoboken, NJ: John Wiley & Sons, 1996

[2] Kay, Steven M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1999

[3] Stoica, Petre and Randolph L. Moses. *Spectral Analysis of Signals*. Englewood Cliffs, NJ: Prentice Hall, 2005

[4] Welch, P. D. ``The use of fast Fourier transforms for the estimation of power spectra: A method based on time averaging over short modified periodograms,'' *IEEE Transactions on Audio and Electroacoustics*, Vol. 15, pp. 70–73, 1967.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- "System Objects in MATLAB Code Generation" (MATLAB Coder)

- When the FFT length is not a power of two, the executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.
- When the FFT length is a power of two, you can generate standalone C and C++ code from this System object.

See Also

See Also

System Objects

dsp.CrossSpectrumEstimator | dsp.SpectrumAnalyzer | dsp.SpectrumEstimator

Introduced in R2013b

getFrequencyVector

System object: dsp.TransferFunctionEstimator
Package: dsp

Get vector of frequencies at which transfer function is estimated

Syntax

`getFrequencyVector(tfe)`

Description

`getFrequencyVector(tfe)` returns the vector of frequencies at which the transfer function is estimated.

If you set the `FrequencyRange` to `'onesided'` and the FFT length, `NFFT`, is even, the frequency vector is of length `NFFT/2+1`, and covers the interval `[0,SampleRate/2]`. If you set the `FrequencyRange` to `'onesided'` and `NFFT` is odd, the frequency vector is of length `(NFFT+1)/2` and covers the interval `[0,SampleRate/2]`. If you set the `FrequencyRange` to `'twosided'`, the frequency vector is of length `NFFT` and covers the interval `[0, SampleRate]`. If you set the `FrequencyRange` to `'centered'`, the frequency vector is of length `NFFT` and covers the range `[-SampleRate/2, SampleRate/2]` and `[-SampleRate/2, SampleRate/2]` for even and odd length `NFFT`, respectively.

getRBW

System object: dsp.TransferFunctionEstimator
Package: dsp

Get resolution bandwidth of transfer function

Syntax

```
getRBW(tfe)
```

Description

`getRBW(tfe)` returns the resolution bandwidth of the transfer function.

The resolution bandwidth, `RBW`, is the smallest positive frequency, or frequency interval, that can be resolved. It is equal to `ENBW*SampleRate/ L`, where `L` is the input length, and `ENBW` is the two-sided equivalent noise bandwidth of the window (in Hz). For example, if `SampleRate=100`, `L=1024`, and `Window='Hann'`, `RBW=enbw(hann(1024))*100/1024`.

reset

System object: dsp.TransferFunctionEstimator
Package: dsp

Reset internal states of transfer function estimator

Syntax

`reset(sysObj)`

Description

`reset(sysObj)` resets the internal states of the System object, `sysObj`, to their initial values. The reset method is always a no-op for unlocked System objects, as the states may not be allocated when the object is not locked.

step

System object: dsp.TransferFunctionEstimator
Package: dsp

Estimate transfer function of system

Syntax

```
Txy = step(tfe,x,y)
Y = step(sysObj,x)
[Y1,...,YN] = step(sysObj,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Txy = step(tfe,x,y)` computes the transfer function, `Txy`, of the system with input x and output y. This computation uses the Welch's averaged periodogram method. `Txy` is the ratio of the Cross Power Spectral Density of `x` and `y`, and the Power Spectral Density of `x`. The columns of `x` and `y` are treated as independent channels.

`Y = step(sysObj,x)` processes the input data, `x`, to produce the output, `Y`, from the System object, `sysObj`. `[Y1,...,YN] = step(sysObj,x)` produces `N` outputs.

Every System object has a step method. The step method processes the input data according to the object algorithm. The number of input and output arguments depends on the algorithm, and may depend also on one or more property settings. The step method for some objects accepts fixed-point (fi) inputs.

Calling step on an object puts that object into a locked state. When locked, you cannot change nontunable properties or any input characteristics (size, data type and complexity) without reinitializing (unlocking and relocking) the object.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.TransitionMetrics System object

Package: dsp

Transition metrics of bilevel waveforms

Description

The `TransitionMetrics` object extracts information such as duration, slew rate, and reference-level crossings for each transition found in the bilevel waveform. The `TransitionMetrics` object can additionally return preshoot, postshoot and settling metrics for the regions immediately before and after each transition.

To obtain transition metrics for a bilevel waveform:

1. Define and set up your transition metrics. See "Construction" on page 4-1797.
2. Call step to calculate the transition metrics according to the properties of `dsp.TransitionMetrics`. The behavior of step is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`tm = dsp.TransitionMetrics` creates a transition metrics System object, `tm`. The object computes the rise time, fall time and width of a pulse. `TransitionMetrics` additionally computes cycle metrics such as pulse separations, periods, and duty cycles.

`tm = dsp.TransitionMetrics('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a `TransitionMetrics` System object, `tm`, with each specified property set to the specified value.

Properties

MaximumRecordLength

Maximum samples to preserve between calls to step. This property requires a positive integer that specifies the maximum number of samples to save between calls to the step method. When the number of samples to be saved exceeds this length, the oldest excess samples are discarded. This property applies when RunningMetrics is true and is tunable.

Default: 1000

PercentReferenceLevels

Lower-, middle-, and upper-percent reference levels. This property contains a 3-element numeric row vector that contains the lower-, middle-, and upper-percent reference levels. These reference levels are used as an offset between the low and high states of the waveform when computing the duration of each transition.

Default: [10 50 90]

PercentStateLevelTolerance

Tolerance of the state level (in percent). This property requires a scalar that specifies the maximum deviation from either the low or high state before it is considered to be outside that state. The tolerance is expressed as a percentage of the waveform amplitude.

Default: 2

PostshootOutputPort

Enable posttransition aberration metrics. If this property is set totrue, overshoot and undershoot metrics are reported for a region defined immediately after each transition. The posttransition aberration region is defined as the waveform interval that begins at the end of each transition and whose duration is the value of PostshootSeekFactor times the computed transition duration. If a complete subsequent transition is detected before the interval is over, the region is truncated at the start of the subsequent transition. The metrics are computed for each transition that has a complete posttransition aberration region.

Default: false

PostshootSeekFactor

Corresponds to the duration of time to search for the overshoot and undershoot metrics immediately following each transition. The duration is expressed as a factor of the duration of the transition. This property is enabled only when the `PostshootOutputPort` property is set to `true` and is tunable.

Default: 3

PreshootOutputPort

Enable pretransition aberration metrics. If the `PreshootOutputPort` property is set to `true`, overshoot and undershoot metrics are reported for a region defined immediately before each transition. The pretransition aberration region is defined as the waveform interval that ends at the start of each transition and whose duration is `PreshootSeekFactor` times the computed transition duration.

Default: `false`

PreshootSeekFactor

Corresponds to the duration of time to search for the overshoot and undershoot metrics immediately preceding each transition. The duration is expressed as a factor of the duration of the transition. This property is enabled only when the `PreshootOutputPort` property is set to `true` and is tunable.

Default: 3

RunningMetrics

Enable metrics over all calls to `step`. If `RunningMetrics` is set to `false`, metrics are computed for each call to `step` independently. If `RunningMetrics` is set to `true`, metrics are computed across subsequent calls to `step`. If there are not enough samples to compute metrics associated with the last transition, posttransition aberration region, or settling seek duration in the current record, the object defers reporting all transition, aberration, and settling metrics associated with the last transition until a subsequent call to `step` is made with enough data to compute all enabled metrics for that transition.

Default: `false`

SampleRate

Sampling rate of uniformly-sampled signal. Specify the sample rate in hertz as a positive scalar. This property is used to construct the internal time values that correspond to the input sample values. Time values start with zero. This property applies when the `TimeInputPort` property is set to `false`.

Default: `1`

SettlingOutputPort

Enable settling metrics. If `SettlingOutputPort` is set to true, settling metrics are reported for each transition. The region used to compute the settling metrics starts at the midcrossing and lasts until the `SettlingSeekDuration` has elapsed. If an intervening transition occurs, or the signal has not settled within the `PercentStateLevelTolerance` of the final level, `NaN` is returned for each metric. If there are not enough samples after the last transition to complete the `SettlingSeekDuration`, no metrics are reported for the last transition. The metrics are reported for the transition the next time `step` is called if the `RunningMetrics` property is set to `true`.

Default: `false`

SettlingSeekDuration

Duration of time over which to search for settling. This property is a scalar that specifies the amount of time to inspect from the mid-reference level crossing (in seconds). If the transition has not yet settled, or a subsequent complete transition is detected within this duration, the `TransitionMetrics` object reports `NaN` for all settling metrics. This property is tunable and applies only when you set the `SettlingOutputPort` property to `true`.

Default: `0.02`

StateLevels

Low- and high-state levels. This property is a 2-element numeric row vector that contains the low and high state levels respectively. These state levels correspond to the nominal logic low and high levels of the pulse waveform. This property is tunable.

Default: `[0 2.3]`

StateLevelsSource

Auto or manual state level computation. If the `StateLevelsSource` property is set to `'Auto'`, the first record sent to `step` is sent to dsp.StateLevels with the default settings to determine the state levels of the incoming waveform. If this property is set to `'Property'`, the object uses the values the user specifies in the `StateLevels` property.

Default: `'Property'`

`TimeInputPort`

Add input to specify sample instants. Set `TimeInputPort` to `true` to enable an additional real input column vector to `step` to specify the sample instants that correspond to the sample values. If this property is `false`, the sample instants are built internally. The sample instants start at zero and increment by the reciprocal of the `SampleRate` property for subsequent samples. The sample instants continue to increment if the `RunningMetrics` property is set to `true` and no intervening calls to the `reset` or `release` methods are encountered.

Default: `false`

Methods

plot	Plots signal and metrics computed in last call to `step`
reset	Reset the running transition metrics object
step	Transition metrics of bilevel waveform

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Transition and Preshoot Information of a 2.3 V Step Waveform

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Compute transition and preshoot information of a 2.3 V step waveform sampled at 4 MHz. Load the data.

```
load('transitionex.mat', 'x');
```

Construct the `dsp.TransitionMetrics` object. Set the `SampleRate` property to `4 MHz`, `StateLevelsSource` property to `'Auto'` to estimate the state levels from the data. Set the `PreshootOutputPort` property to `true` to output pretransition aberration metrics when you call `step`.

```
tm = dsp.TransitionMetrics('SampleRate',4e6, ...
                           'StateLevelsSource','Auto', ...
                           'PreshootOutputPort',true)
```

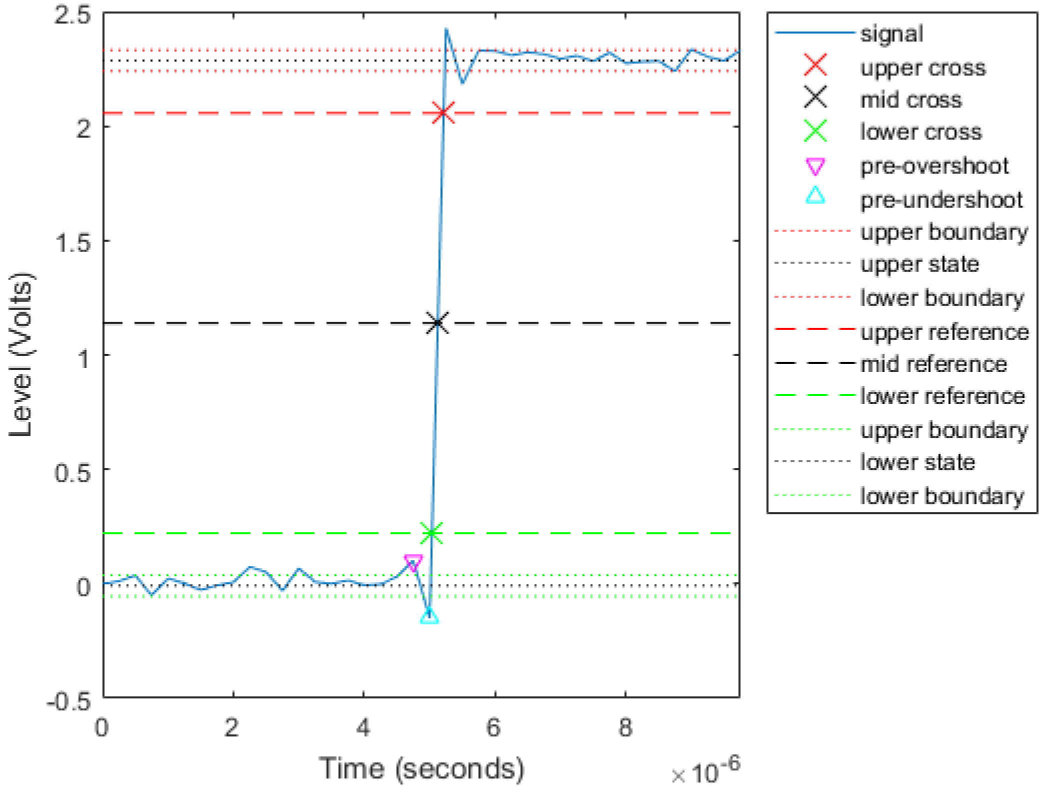
```
tm = 

  dsp.TransitionMetrics with properties:

            StateLevelsSource: 'Auto'
                  StateLevels: [0 2.3000]
    PercentStateLevelTolerance: 2
       PercentReferenceLevels: [10 50 90]
               RunningMetrics: false
                TimeInputPort: false
                   SampleRate: 4000000
           PreshootOutputPort: true
           PreshootSeekFactor: 3
          PostshootOutputPort: false
           SettlingOutputPort: false
```

Call `step` to compute the transition and preshoot information. Plot the result.

```
[transition, preshoot] = tm(x)
```

```
plot(tm)
```

```
transition =

  struct with fields:

        Duration: 1.7800e-07
        Polarity: 1
        SlewRate: 1.0310e+07
     MiddleCross: 5.1250e-06
      LowerCross: 5.0360e-06
      UpperCross: 5.2140e-06


preshoot =

  struct with fields:

             Overshoot: 4.8050
            Undershoot: 6.1798
        OvershootLevel: 0.1020
       UndershootLevel: -0.1500
      OvershootInstant: 4.7500e-06
     UndershootInstant: 5.0000e-06
```



Transition, Postshoot, and Settling Information of a 2.3 V Step Waveform

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Compute transition, postshoot, and settling information of a 2.3 V step waveform sampled at 4 MHz. Load the data along with the sampling instants.

```
load('transitionex.mat', 'x', 't');
```

Construct the `dsp.TransitionMetrics` object. Set the `TimeInputPort` property to `true`, `StateLevels` property to `[0 2.3]` to match waveform statelevels. To output the postshoot information and settling information, set the `PostShootOutputPort` and

`SettlingOutputPort` properties to `true`. Set the `SettlingSeekDuration` property to `2 ms`.

```
tm = dsp.TransitionMetrics('TimeInputPort',true, ...
                                 'StateLevels',[0 2.3], ...
                                 'PostshootOutputPort',true, ...
                                 'SettlingOutputPort',true, ...
                                 'SettlingSeekDuration',2e-6)
```

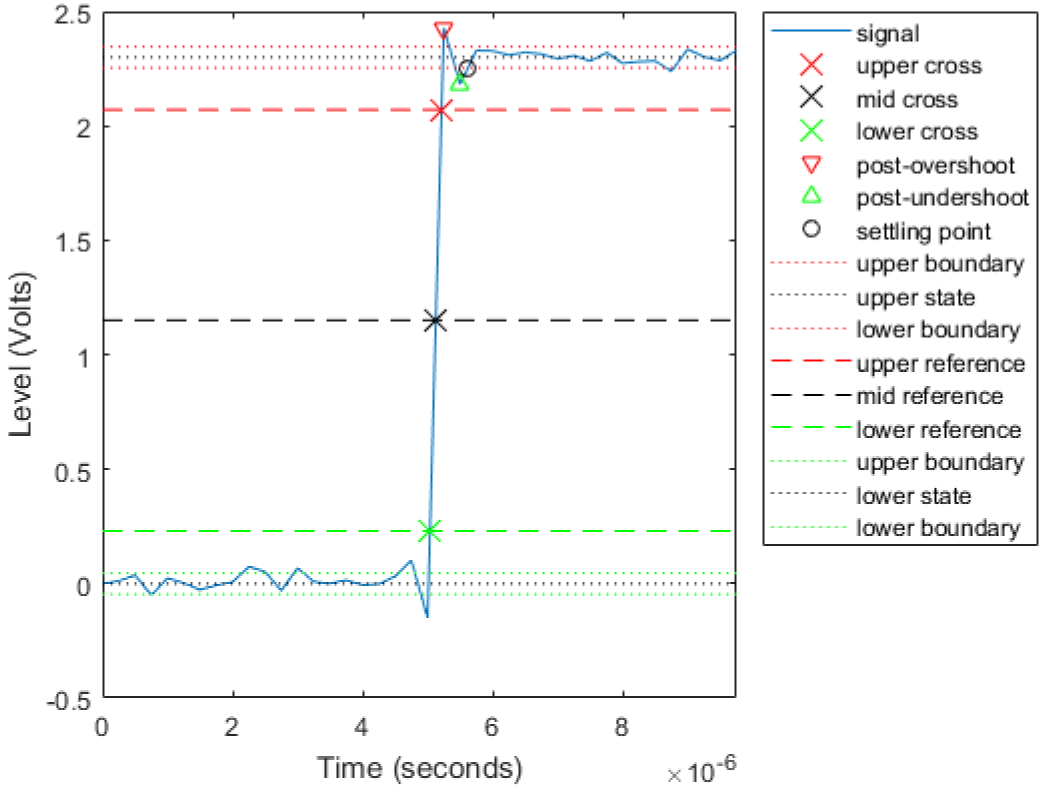
```
tm =

  dsp.TransitionMetrics with properties:

             StateLevelsSource: 'Property'
                   StateLevels: [0 2.3000]
    PercentStateLevelTolerance: 2
        PercentReferenceLevels: [10 50 90]
                RunningMetrics: false
                 TimeInputPort: true
             PreshootOutputPort: false
           PostshootOutputPort: true
           PostshootSeekFactor: 3
            SettlingOutputPort: true
          SettlingSeekDuration: 2.0000e-06
```

Compute the transition, postshoot, and settling information with `step` and plot the result.

```
 [transition, postshoot, settling] = tm(x,t)
 plot(tm)
```

```
transition =

  struct with fields:

       Duration: 1.7846e-07
       Polarity: 1
       SlewRate: 1.0310e+07
    MiddleCross: 5.1261e-06
     LowerCross: 5.0369e-06
     UpperCross: 5.2153e-06
```

```
postshoot = 

  struct with fields:

            Overshoot: 5.5483
           Undershoot: 5.1051
       OvershootLevel: 2.4276
      UndershootLevel: 2.1826
     OvershootInstant: 5.2500e-06
    UndershootInstant: 5.5000e-06


settling = 

  struct with fields:

    Duration: 4.9529e-07
       Level: 2.2540
     Instant: 5.6214e-06
```



References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

See Also

See Also

dsp.PulseMetrics | dsp.StateLevels

Introduced in R2012a

plot

System object: dsp.TransitionMetrics
Package: dsp

Plots signal and metrics computed in last call to `step`

Syntax

`plot(tm)`

Description

`plot(tm)` plots the signal and metrics resulting from the last call of the step method.

By default `plot` displays

- the low and high state levels and the state-level boundaries defined by the `PercentStateLevelTolerance` property.
- the lower-, middle-, and upper-reference levels.
- the locations of the mid-reference level crossings of the positive (+) and negative (-) transitions of each detected pulse.

When you set the `TransitionOutputPort` property to `true`, the locations of the upper and lower crossings are also plotted. When the `PreshootOutputPort` or `PostShootOutputPort` properties are set to `true`, the corresponding overshoots and undershoots are plotted as inverted or non-inverted triangles. When the `SettlingOutputPort` property is set to `true`, the locations where the signal enters and remains within the upper- and lower-state boundaries over the specified seek duration are plotted.

reset

System object: dsp.TransitionMetrics
Package: dsp

Reset the running transition metrics object

Syntax

`reset(tm)`

Description

`reset(tm)` clears information from previous calls to step.

step

System object: dsp.TransitionMetrics
Package: dsp

Transition metrics of bilevel waveform

Syntax

```
PULSE = step(tm,X)
[PULSE,CYCLE] = step(tm,X)
[PULSE,TRANSITION] = step(tm,X)
[PULSE,PRESHOOT] = step(tm,X)
[PULSE,POSTSHOOT] = step(tm,X)
[PULSE,SETTLING] = step(tm,X)
[...] = step(tm,X,T)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`PULSE = step(tm,X)` returns a structure array, `PULSE`, whose fields contain real-valued column vectors. The number of rows of each field corresponds to the number of complete pulses found in the real-valued column vector input, `X`. Each pulse starts with a transition of the polarity specified by the `Polarity` property and ends with a transition of the opposite polarity.

`PULSE` fields:

- `PositiveCross` — Instants where the positive-going transitions cross the mid-reference level of each pulse
- `NegativeCross` — Instants where the negative-going transitions cross the mid-reference level of each pulse

- `Width` — Absolute difference between `PositiveCross` and `NegativeCross` of each pulse
- `RiseTime` — Duration between the linearly-interpolated instants when the positive-going (rising) transition of each pulse crosses the lower- and upper-reference levels
- `FallTime` — Duration between the linearly-interpolated instants when the negative-going (falling) transition of each pulse crosses the upper- and lower-reference levels

`[PULSE,CYCLE] = step(tm,X)` returns a structure array, `CYCLE`, whose fields contain real-valued column vectors when you set the `CycleOutputPort` property to `true`. The number of rows of each field corresponds to the number of complete pulse periods found in the real-valued column vector input, `X`. You need at least three consecutive alternating polarity transitions that start and end with the same polarity as the value of the `Polarity` property if you want to compute cycle metrics. If the last transition found the input, `X`, does not match the polarity of the `Polarity` property, the pulse separation, period, frequency, and duty cycle are not reported for the last pulse. If the `RunningMetrics` property is set to `true` when this occurs, all pulse, cycle, transition, preshoot, postshoot, and settling metrics associated with the last pulse are deferred until a subsequent call to `step` detects the next transition.

`CYCLE` fields:

- `Period` — Duration between the first transition of the current pulse and the first transition of the next pulse
- `Frequency` — Reciprocal of the period
- `Separation` — Durations between the mid-reference level crossings of the second transition of each pulse and the first transition of the next pulse
- `Width` — Durations between the mid-reference level crossings of the first and second transitions of each pulse. This is equivalent to the width parameter of the `PULSE` structure.
- `DutyCycle` — Ratio of the width to the period for each pulse

`[PULSE,TRANSITION] = step(tm,X)` returns a structure array, `TRANSITION`, when you set the `TransitionOutputPort` property to `true`. The fields of `TRANSITION` contain real-valued matrices with two columns which correspond to the metrics of the first and second transitions. The number of rows corresponds to the number of pulses found in the input waveform.

`TRANSITION` fields:

- `Duration` — Amount of time between the interpolated instants where the transition crosses the lower- and upper-reference levels
- `SlewRate` — Ratio of absolute difference between the upper- and lower-reference levels to the transition duration
- `MiddleCross` — Linearly-interpolated instant where the transition first crosses the mid-reference level
- `LowerCross` — Linearly-interpolated instant where the signal crosses the lower-reference level
- `UpperCross` — Linearly-interpolated instant where the signal crosses the upper-reference level

`[PULSE,PRESHOOT] = step(tm,X)` returns a structure array, `PRESHOOT`, when you set the `PreshootOutputPort` property to `true`. The fields of `PRESHOOT` contain real-valued 2-column matrices whose row length corresponds to the number of transitions found in the input waveform. The field names are identical to those of the `POSTSHOOT` structure.

`[PULSE,POSTSHOOT] = step(tm,X)` returns a structure array, `POSTSHOOT`, when you set the `PostshootOutputPort` property to `true`. The fields of `POSTSHOOT` contain real-valued 2-column matrices whose row length corresponds to the number of transitions found in the input waveform.

`PRESHOOT` and `POSTSHOOT` fields:

- `Overshoot` — Overshoot of the region of interest expressed as a percentage of the waveform amplitude
- `Undershoot` — Undershoot of the region of interest expressed as a percentage of the waveform amplitude
- `OvershootLevel` — Level of the overshoot
- `UndershootLevel` — Level of the undershoot
- `OvershootInstant` — Instant that corresponds to the overshoot
- `UndershootInstant` — Instant that corresponds to the undershoot

`[PULSE,SETTLING] = step(tm,X)` returns a structure, `SETTLING`, when you set the `SettlingOutputPort` property to `true`. The fields of `SETTLING` correspond to the settling metrics for each transition. Each field is a column vector whose elements correspond to the individual settling durations, levels, and instants.

`SETTLING` fields:

- `Duration` — Amount of time from when the signal crosses the mid-reference level to the time where the signal enters and remains within the specified `PercentStateLevelTolerance` of the waveform amplitude over the specified settling seek duration
- `Instant` — Instant in time where the signal enters and remains within the specified tolerance
- `Level` — Level of the waveform where it enters and remains within the specified tolerance

The above operations can be used simultaneously, provided the System object properties are set appropriately. One example of providing all possible inputs and returning all possible outputs is :

`[PULSE,CYCLE,TRANSITION,PRESHOOT,POSTSHOOT,SETTLING] = step(tm,X)` which returns the `PULSE`, `CYCLE`, `TRANSITION`, `PRESHOOT`, `POSTSHOOT`, and `SETTLING` structure arrays when the `CycleOutputPort`, `PreshootOutputPort`, `PostshootPort`, and `SettlingOutputPort` properties are `true`. You may enable or disable any combination of output ports. However, the output arguments are defined in the order shown here.

`[...] = step(tm,X,T)` performs the above metrics with respect to a sampled signal, whose sample values, `X`, and sample instants, `T`, are real-valued column vectors of the same length. The additional input `T` applies only when you set the `TimeInputPort` property to `true`.

dsp.UDPReceiver System object

Package: dsp

Receive UDP packets from network

Description

The `UDPReceiver` object receives UDP packets from the network.

To receive UDP packets from the network:

1. Define and set up your UDP receiver. See "Construction" on page 4-1815.
2. Call step to receive the UDP packets according to the properties of `dsp.UDPReceiver`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`udpr = dsp.UDPReceiver` returns a UDP receiver System object, `udpr`, that receives UDP packets from a specified port.

`udpr = dsp.UDPReceiver('PropertyName',PropertyValue, ...)` returns a window object, `udpr`, with each property set to the specified value.

Properties

`LocalIPPort`

Local port on which to receive data

Specify the port on which to receive data. This property is tunable in generated code but not tunable during simulation. The default is `25000`. The value can be in the range [1 65535].

`RemoteIPAddress`

Address from which data was sent

Specify the remote IP address from which to receive data. The default is `'0.0.0.0'`, which indicates that data can be accepted from any remote IP address.

`ReceiveBufferSize`

Maximum size of internal buffer

Specify the size of the buffer that receives UDP packets, in bytes. The default is `8192`.

`MaximumMessageLength`

Maximum size of output message

Specify the size of the output message. If received packets exceed the specified value, their contents are truncated. The default is `255`.

`MessageDataType`

Data type of the message

Specify the data type of the message as `| double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | logical |`. The default is `uint8`.

Methods

step	Receive UDP packet

Common to All System Objects	
`clone`	Create System object with same property values

Common to All System Objects	
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Byte Transmission using UDP

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For `dsp.UDPReceiver` System object™, myObject() becomes step(myObject). For all other objects, myObject(x) becomes step(myObject,x).

Send some UDP packets, and calculate the number of successfully transmitted bytes:

```
udpr = dsp.UDPReceiver('LocalIPPort',31000);
udps = dsp.UDPSender('RemoteIPPort',31000);
% To prevent the loss of packets, call the |setup| method
% on the object before the first call to the |step| method.
setup(udpr);

bytesSent = 0;
bytesReceived = 0;
dataLength = 128;

for k = 1:20
   dataSent = uint8(255*rand(1,dataLength));
   bytesSent = bytesSent + dataLength;
   udps(dataSent);
   dataReceived = udpr();
   bytesReceived = bytesReceived + length(dataReceived);
end

release(udps);
release(udpr);

fprintf('Bytes sent:     %d\n', bytesSent);
```

```
fprintf('Bytes received: %d\n', bytesReceived);
```

```
Bytes sent:     2560
Bytes received: 2560
```

Tune the UDP Port Number in MATLAB

Note: This example runs only in R2017a or later.

The `TunableUDP` function contains the algorithm to send and receive sine wave data over a UDP network. The inputs to this function are the local IP port number and the remote IP port number of the `dsp.UDPReceiver` and the `dsp.UDPSender` System objects, respectively. The outputs of this function are the number of bytes sent and received over the UDP network.

```
type TunableUDP
```

```
function [bytesSent,bytesReceived] = TunableUDP(localipport,remoteipport)

persistent udpRx
persistent udpSend
persistent sine

if isempty(udpRx)
    udpRx = dsp.UDPReceiver('MessageDataType','double');
    udpSend = dsp.UDPSender;
    sine = dsp.SineWave('SamplesPerFrame',250);
end

udpRx.LocalIPPort = localipport;
udpSend.RemoteIPPort = remoteipport;

bytesSent = 0;
bytesReceived = 0;
dataLength = 250;
setup(udpRx);
dataSent = sine();
udpSend(dataSent);
udpRx();

for i = 1:10
    dataSent = sine();
    bytesSent = bytesSent + dataLength;
```

```
    udpSend(dataSent);
    dataReceived = udpRx();
    bytesReceived = bytesReceived + length(dataReceived);
end
```

You can change the local IP port number and the remote IP port number in the generated code without having to regenerate the code. Generate a MEX file from the `TunableUDP.m` function.

```
codegen TunableUDP -args {65000,65000}
```

Call the MEX function, `TunableUDP_mex`, with `RemoteIPPort` and `LocalIPPort` numbers set to `65000`.

```
[bytesSentPort65000,bytesReceivedPort65000] = TunableUDP_mex(65000,65000)
```

```
bytesSentPort65000 =

        2500


bytesReceivedPort65000 =

        2500
```

All the 2500 bytes sent are successfully received.

Clear the MEX function and change the port numbers to `25000`, `20000`, and `45000`. Verify the number of bytes transmitted and received.

```
clear mex;
[bytesSentPort25000,bytesReceivedPort25000] = TunableUDP_mex(25000,25000)
```

```
bytesSentPort25000 =

        2500


bytesReceivedPort25000 =

        2500
```

Clear the MEX function and change the port numbers to 20000.

```
clear mex;
[bytesSentPort20000,bytesReceivedPort20000] = TunableUDP_mex(20000,20000)
```

```
bytesSentPort20000 =

        2500


bytesReceivedPort20000 =

        2500
```

Clear the MEX function and change the port numbers to 45000.

```
clear mex;
[bytesSentPort45000,bytesReceivedPort45000] = TunableUDP_mex(45000,45000)
```

```
bytesSentPort45000 =

        2500


bytesReceivedPort45000 =

        2500
```

- “Voice Over IP (VOIP)”

Algorithms

This object implements the algorithm, inputs, and outputs described on the `UDP Receive` block reference page. The object properties correspond to the block parameters, except:

- The **Output variable-size signal** parameter is not included in the System object.
- The **Sample time** parameter is not included in the System object.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- "System Objects in MATLAB Code Generation" (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "How To Run a Generated Executable Outside MATLAB".

The `LocalIPPort` property is tunable in generated code but not tunable during simulation.

See Also

See Also

System Objects
dsp.UDPSender

Blocks
`UDP Receive` | `UDP Send`

Topics
"Voice Over IP (VOIP)"

Introduced in R2012a

step

System object: dsp.UDPReceiver
Package: dsp

Receive UDP packet

Syntax

```
P = step (udpr)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`P = step (udpr)` receives one UDP packet, `P`, from the network. The input argument, `udpr`, is a `dsp.UDPReceiver` object.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

The first time you call the `step` method on a `UDPReceiver` object, the object also allocates resources and begins listening for data. As a result, the first `step` call may not receive data. To prevent the loss of packets, call the `setup` method on the object before the first call to the `step` method.

Examples

Receive one UDP Packet

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For `dsp.UDPReceiver` System object™, myObject() becomes step(myObject). For all other objects, myObject(x) becomes step(myObject,x).

This example shows how to send and receive one UDP packet. Set up the objects to send and receive UDP packets.

```
udps = dsp.UDPSender('RemoteIPPort',31000);
udpr = dsp.UDPReceiver('LocalIPPort',31000);
```

Create some data to send and receive.

```
dataSent = uint8(255*rand(1,128));
bytessent = length(dataSent);
```

Send and receive the data. Verify that the number of bytes is equal.

```
udps(dataSent);
datain = udpr();
bytesreceived = length(datain);
isequal(length(bytessent),length(bytesreceived))
```

```
ans =

  logical

   1
```

```
release(udps);
release(udpr);
```

dsp.UDPSender System object

Package: dsp

Send UDP packets to network

Description

The `UDPSender` object sends UDP packets to the network.

To send UDP packets to the network:

1. Define and set up your UDP sender. See “Construction” on page 4-1824.
2. Call step to send the packets according to the properties of `dsp.UDPSender`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`udps = dsp.UDPSender` returns a UDP sender object, `udps`, that sends UDP packets to a specified port.

`udps = dsp.UDPSender('PropertyName',PropertyValue, ...)` returns a UDP sender object, `udps`, with each property set to the specified value.

Properties

`RemoteIPAddress`

Remote address to which to send data

Specify the remote (that is, host) IP address to which the data is sent. The default is '127.0.0.1', which is the local host.

RemoteIPPort

Remote port to which to send data

Specify the port at the remote IP address to which the data is sent. This property is tunable in generated code but not tunable during simulation. The default is 25000.

LocalIPPortSource

Source of the LocalIPPort property

Specify how to determine the port on the host as | Auto | Property |. If you specify Auto, the object selects the port dynamically from the available ports. If you specify Property, the object uses the source specified in the LocalIPPort property. The default is Auto.

LocalIPPort

Local port from which to send data

Specify the port from which to send data. This property applies when you set the LocalIPPortSource property to Auto. The default is 25000.

Methods

step	Send a UDP packet to network

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Byte Transmission using UDP

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For `dsp.UDPReceiver` System object™, myObject() becomes step(myObject). For all other objects, myObject(x) becomes step(myObject,x).

Send some UDP packets, and calculate the number of successfully transmitted bytes:

```
udpr = dsp.UDPReceiver('LocalIPPort',31000);
udps = dsp.UDPSender('RemoteIPPort',31000);
% To prevent the loss of packets, call the |setup| method
% on the object before the first call to the |step| method.
setup(udpr);

bytesSent = 0;
bytesReceived = 0;
dataLength = 128;

for k = 1:20
   dataSent = uint8(255*rand(1,dataLength));
   bytesSent = bytesSent + dataLength;
   udps(dataSent);
   dataReceived = udpr();
   bytesReceived = bytesReceived + length(dataReceived);
end

release(udps);
release(udpr);

fprintf('Bytes sent:     %d\n', bytesSent);
fprintf('Bytes received: %d\n', bytesReceived);
```

```
Bytes sent:     2560
Bytes received: 2560
```

Tune the UDP Port Number in MATLAB

Note: This example runs only in R2017a or later.

The `TunableUDP` function contains the algorithm to send and receive sine wave data over a UDP network. The inputs to this function are the local IP port number and the remote IP port number of the `dsp.UDPReceiver` and the `dsp.UDPSender` System objects, respectively. The outputs of this function are the number of bytes sent and received over the UDP network.

```
type TunableUDP
```

```
function [bytesSent,bytesReceived] = TunableUDP(localipport,remoteipport)

persistent udpRx
persistent udpSend
persistent sine

if isempty(udpRx)
    udpRx = dsp.UDPReceiver('MessageDataType','double');
    udpSend = dsp.UDPSender;
    sine = dsp.SineWave('SamplesPerFrame',250);
end

udpRx.LocalIPPort = localipport;
udpSend.RemoteIPPort = remoteipport;

bytesSent = 0;
bytesReceived = 0;
dataLength = 250;
setup(udpRx);
dataSent = sine();
udpSend(dataSent);
udpRx();

for i = 1:10
    dataSent = sine();
    bytesSent = bytesSent + dataLength;
    udpSend(dataSent);
    dataReceived = udpRx();
    bytesReceived = bytesReceived + length(dataReceived);
end
```

You can change the local IP port number and the remote IP port number in the generated code without having to regenerate the code. Generate a MEX file from the `TunableUDP.m` function.

```
codegen TunableUDP -args {65000,65000}
```

Call the MEX function, `TunableUDP_mex`, with `RemoteIPPort` and `LocalIPPort` numbers set to `65000`.

```
[bytesSentPort65000,bytesReceivedPort65000] = TunableUDP_mex(65000,65000)
```

```
bytesSentPort65000 =

        2500


bytesReceivedPort65000 =

        2500
```

All the 2500 bytes sent are successfully received.

Clear the MEX function and change the port numbers to `25000`, `20000`, and `45000`. Verify the number of bytes transmitted and received.

```
clear mex;
[bytesSentPort25000,bytesReceivedPort25000] = TunableUDP_mex(25000,25000)
```

```
bytesSentPort25000 =

        2500


bytesReceivedPort25000 =

        2500
```

Clear the MEX function and change the port numbers to `20000`.

```
clear mex;
[bytesSentPort20000,bytesReceivedPort20000] = TunableUDP_mex(20000,20000)
```

```
bytesSentPort20000 =

        2500
```

```
bytesReceivedPort20000 =

        2500
```

Clear the MEX function and change the port numbers to 45000.

```
clear mex;
[bytesSentPort45000,bytesReceivedPort45000] = TunableUDP_mex(45000,45000)
```

```
bytesSentPort45000 =

        2500


bytesReceivedPort45000 =

        2500
```

- “Voice Over IP (VOIP)”

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “How To Run a Generated Executable Outside MATLAB”.

The `RemoteIPPort` property is tunable in generated code but not tunable during simulation.

See Also

See Also

System Objects
dsp.UDPReceiver

Blocks
`UDP Receive` | `UDP Send`

Topics
"Voice Over IP (VOIP)"

Introduced in R2012a

step

System object: dsp.UDPSender
Package: dsp

Send a UDP packet to network

Syntax

`Y = step(udps,PACKET)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(udps,PACKET)` sends one UDP packet, PACKET, to the network.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.UniformDecoder System object

Package: dsp

Decode integer input into floating-point output

Description

The `UniformDecoder` object decodes integer input into floating-point output. The decoder adheres to the definition for uniform decoding specified in ITU-T Recommendation G.701.

To decode an integer input into a floating-point output:

1. Define and set up your uniform decoder. See "Construction" on page 4-1832.
2. Call step to decode the input according to the properties of `dsp.UniformDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ud = dsp.UniformDecoder` returns a uniform decoder, `ud`, that performs the inverse operation of the `UniformEncoder` on page 4-1837 object, reconstructing quantized floating-point values from encoded integer input.

`ud = dsp.UniformDecoder('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a uniform decoder, `ud`, with each specified property set to the specified value.

`ud = dsp.UniformDecoder(peakvalue,numbits,` `'`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a uniform decoder, `ud`, with the `PeakValue` on page 4-1833 property set to `peakvalue`, the `NumBits` on page 4-1833 property set to `numbits`, and other specified properties set to the specified values.

Properties

PeakValue

Largest amplitude represented in encoded input

Specify the largest amplitude represented in the encoded input as a nonnegative numeric scalar. To correctly decode values encoded with the `UniformEncoder` on page 4-1837 object, set the `PeakValue` property in both objects to the same value. For more information on setting this property, see the `PeakValue` on page 4-1833 property description on the `dsp.UniformEncoder` reference page. The default is `1`.

NumBits

Number of input bits used to encode data

Specify the number of bits used to encode the input data as an integer value between `2` and `32`. The value of this property can be less than the total number of bits supplied by the input data type. To correctly decode values encoded with the `UniformEncoder` on page 4-1837 object, set the `NumBits` property in both objects to the same value. For more information on setting this property, see the `NumBits` on page 4-1838 property description on the `dsp.UniformEncoder` reference page. The default is `3`.

OverflowAction

Action to take when integer input is out of range

Specify the behavior of the uniform decoder when the integer input is out of range as `Saturate` or `Wrap`. The value of the `NumBits` on page 4-1833 property specifies the representable range of the input. The default is `Saturate`.

OutputDataType

Output data type as single or double

Specify the data type of the output as `single` or `double`. The default is `double`.

Methods

step	Decode integer input into quantized floating-point output

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Decode An Encoded Sequence

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
ue = dsp.UniformEncoder;
ue.PeakValue = 2;
ue.NumBits = 4;
ue.OutputDataType = 'Signed integer';
x = (0:0.25:2)'; % Create an input sequence
ud = dsp.UniformDecoder;
ud.PeakValue = 2;
ud.NumBits = 4;
x_encoded = ue(x);
```

Check that the last element has been saturated.

```
x_decoded = ud(x_encoded);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Uniform Decoder` block reference page. The object properties correspond to the block parameters.

See Also

dsp.UniformEncoder

Introduced in R2012a

step

System object: dsp.UniformDecoder
Package: dsp

Decode integer input into quantized floating-point output

Syntax

`Y = step(ud,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(ud,X)` reconstructs quantized floating-point output `Y` from the encoded integer input `X`. Input `X` can be real or complex values of the following six integer data types: `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.UniformEncoder System object

Package: dsp

Quantize and encode floating-point input into integer output

Description

The `UniformEncoder` object quantizes floating-point input, using the precision you specify in the `NumBits` on page 4-1838 property, and encodes the quantized input into integer output. The operations of the uniform encoder adhere to the definition for uniform encoding specified in ITU-T Recommendation G.701.

To quantize and encode a floating-point input into an integer output:

1. Define and set up your uniform encoder. See "Construction" on page 4-1837.
2. Call step to encode the input according to the properties of `dsp.UniformEncoder`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ue = dsp.UniformEncoder` returns a uniform encoder, `ue`, that quantizes floating-point input samples and encodes them as integers using 2*N*-level quantization, where *N* is an integer.

`ue = dsp.UniformEncoder('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns an uniform encoder, `ue`, with each specified property set to the specified value.

`ue = dsp.UniformEncoder(peakvalue,numbits,`
`'`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a uniform encoder, `ue`, with the `PeakValue` on page 4-1838 property set to `peakvalue`, the `NumBits` on page 4-1838 property set to `numbits`, and other specified properties set to the specified values.

Properties

PeakValue

Largest input amplitude to be encoded

Specify the largest input amplitude to be encoded, as a nonnegative numeric scalar. If the real or imaginary input are outside of the interval $[-P,(1-2^{(1-B)})P]$, where P is the peak value and B is the value of the `NumBits` on page 4-1838 property, the uniform encoder saturates (independently for complex inputs) at those limits. The default is `1`.

NumBits

Number of bits needed to represent output

Specify the number of bits needed to represent the integer output as an integer value between `2` and `32`. The number of levels at which the uniform encoder quantizes the floating-point input is 2^B, where B is the number of bits. The default is `8`.

OutputDataType

Data type of output

Specify the data type of the output as `Unsigned integer` or `Signed integer`. Unsigned outputs are `uint8`, `uint16`, or `uint32`, and signed outputs are `int8`, `int16`, or `int32`. The quantized inputs are linearly (uniformly) mapped to the intermediate integers in the interval $[0,2^{(B-1)}]$ when you set this property to `Unsigned integer`, and in the interval $[-2^{(B-1)},2^{(B-1)}-1]$ when you set this property to `Signed integer`. The variable B in both expressions corresponds to the value of the `NumBits` on page 4-1838 property.

Methods

step	Quantize and encode input

Common to All System Objects	
`clone`	Create System object with same property values

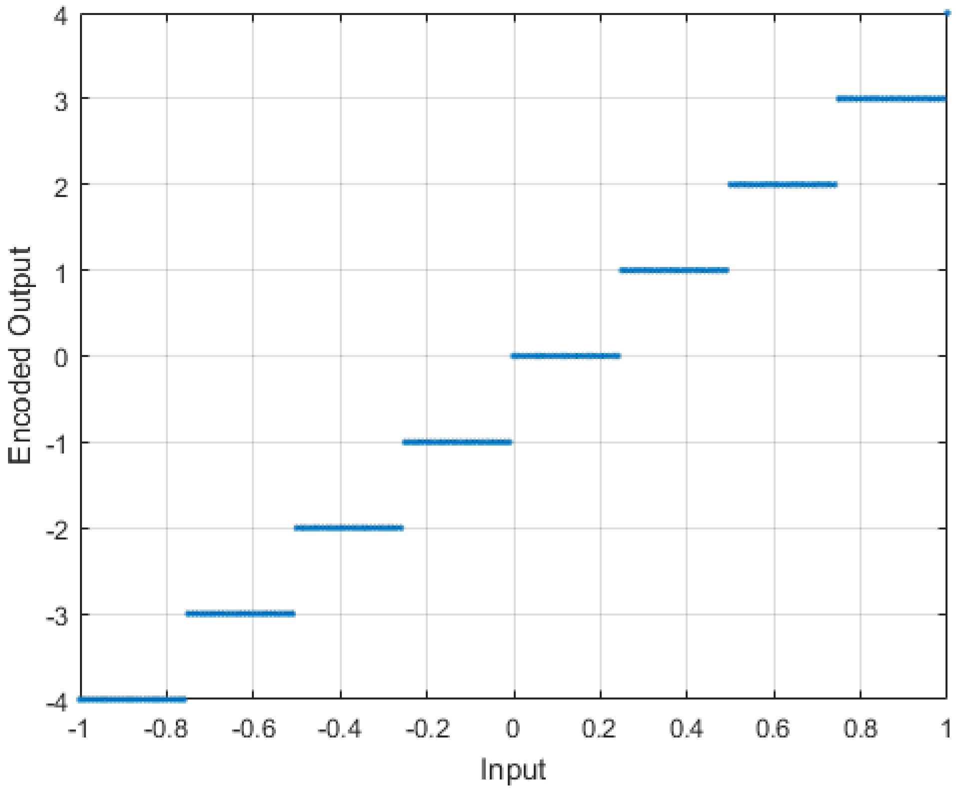
Common to All System Objects	
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Encode a Sequence Of Numbers

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
ue = dsp.UniformEncoder;
ue.PeakValue = 2;
ue.NumBits = 4;
ue.OutputDataType = 'Signed integer';
x = [-1:0.01:1]'; % Create an input sequence
x_encoded = ue(x);
plot(x, x_encoded,'.');
xlabel('Input'); ylabel('Encoded Output'); grid
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Uniform Encoder` block reference page. The object properties correspond to the block parameters.

See Also

dsp.UniformDecoder

Introduced in R2012a

step

System object: dsp.UniformEncoder
Package: dsp

Quantize and encode input

Syntax

```
Y = step(ue,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(ue,X)` quantizes and encodes the input `X` to `Y`. Input `X` can be real or complex, double-, or single-precision values. The uniform encoder chooses the output data type according to the following table.

Number of Bits	Unsigned Integer	Signed Integer
2 to 8	`uint8`	`int8`
9 to 16	`uint16`	`int16`
17 to 32	`uint32`	`int32`

The row corresponds to the value of the `NumBits` property, and the column corresponds to the value of the `OutputDataType` property.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as

dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.UpperTriangularSolver System object

Package: dsp

Solve upper-triangular matrix equation

Description

The `UpperTriangularSolver` object solves $UX = B$ for X when U is a square, upper-triangular matrix with the same number of rows as B.

To solve $UX = B$:

1 Define and set up your linear system solver. See "Construction" on page 4-1843.
2 Call step to solve the equation according to the properties of `dsp.UpperTriangularSolver`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = dsp.UpperTriangularSolver` returns a linear system solver, `H`, used to solve $UX = B$ where U is an upper (or unit-upper) triangular matrix.

`H = dsp.UpperTriangularSolver('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a linear system solver, `H`, with each specified property set to the specified value.

Properties

`OverwriteDiagonal`

Replace diagonal elements of input with ones

When you set this property to `true`, the linear system solver replaces the elements on the diagonal of the input, *U*, with ones. Set this property to either `true` or `false`. The default is `false`.

`RealDiagonalElements`

Indicate that diagonal of complex input is real

When you set this property to `true`, the linear system solver optimizes computation speed if the diagonal elements of complex input, *U*, are real. This property applies only when you set the `OverwriteDiagonal` on page 4-1843 property to `false`. Set this property to either `true` or `false`. The default is `false`.

Fixed-Point Properties

`RoundingMethod`

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`.

`OverflowAction`

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

`ProductDataType`

Data type of product

Specify the product data type as `Full precision`, `Same as input`, or `Custom`. The default is `Full precision`.

`CustomProductDataType`

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `ProductDataType` on page 4-1844 property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Data type of accumulator

Specify the accumulator data type as `Full precision`, `Same as first input`, `Same as product`, or `Custom`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `AccumulatorDataType` on page 4-1845 property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Data type of output

Specify the output data type as `Same as first input` or `Custom`. The default is `Same as first input`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` on page 4-1845 property to `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Solve matrix equation for specified inputs

Common to All System Objects	
`clone`	Create System object with same property values

Common to All System Objects	
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Solve an Upper Traingular Matrix

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
uptriang = dsp.UpperTriangularSolver;
u = triu(rand(4, 4));
b = rand(4, 1);
```

Check that result is the solution to the linear equations.

```
x1 = inv(u)*b
x = uptriang(u, b)
```

```
x1 =

 -179.1887
  265.6759
  -29.3098
    6.7624


x =

 -179.1887
  265.6759
  -29.3098
    6.7624
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Backward Substitution` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.LowerTriangularSolver

Introduced in R2012a

step

System object: dsp.UpperTriangularSolver
Package: dsp

Solve matrix equation for specified inputs

Syntax

`X = step(uptriang,U,B)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`X = step(uptriang,U,B)` computes the solution, *X*, of the matrix equation *UX* = *B*, where *U* is a square, upper-triangular matrix with the same number of rows as the matrix *B*.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.VariableBandwidthFIRFilter System object

Package: dsp

Variable bandwidth FIR filter

Description

The `VariableBandwidthFIRFilter` object filters each channel of the input using FIR filter implementations. It does so while having the capability of tuning the bandwidth. This filter supports double and single precision inputs.

To filter each channel of the input:

1. Define and set up your variable bandwidth FIR filter. See “Construction” on page 4-1849.
2. Call step to filter each channel of the input according to the properties of `dsp.VariableBandwidthFIRFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`vbw = dsp.VariableBandwidthFIRFilter` returns a System object, `vbw`, which independently filters each channel of the input over successive calls to the `step` method. This System object uses a specified FIR filter implementation. The filter’s cutoff frequency may be tuned during the filtering operation. The variable bandwidth FIR filter is designed using the window method.

`vbw = dsp.VariableBandwidthFIRFilter('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a variable bandwidth FIR filter System object, `vbw`, with each property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

SampleRate

Input sample rate

Specify the sampling rate of the input in Hertz as a finite numeric scalar. The default is 44.1 kHz. This property is non-tunable.

FilterType

Filter type

Specify the type of the filter as one of 'Lowpass' | 'Highpass' | 'Bandpass' | 'Bandstop'. The default is 'Lowpass'. This property is non-tunable.

FilterOrder

FIR filter order

Specify the order of the FIR filter as a positive integer scalar. The default is 30. This property is non-tunable.

Window

Window function

Specify the window function used to design the FIR filter as one of 'Hann' | 'Hamming' | 'Chebyshev' | 'Kaiser'. The default is 'Hann'. This property is non-tunable.

KaiserWindowParameter

Kaiser window parameter

Specify the Kaiser window parameter as a real scalar. This property applies when you set the window property to 'Kaiser'. The default is 0.5. This property is non-tunable.

CutoffFrequency

Filter cutoff frequency

Specify the filter cutoff frequency in Hz as a real, positive scalar, smaller than the `SampleRate/2`. This property applies if you set the `FilterType` property to `'Lowpass'` or `'Highpass'`. The default is 512 Hz. This property is tunable.

`CenterFrequency`

Filter center frequency

Specify the filter center frequency in Hz as a real, positive scalar, smaller than `SampleRate/2`. This property applies when you set the `FilterType` property to `'Bandpass'` or `'Bandstop'`. The default is 11025 Hz. This property is tunable.

`Bandwidth`

Filter bandwidth

Specify the filter bandwidth in Hertz as a real, positive scalar, smaller than `SampleRate/2`. This property applies if you set the `FilterType` property to `'Bandpass'` or `'Bandstop'`. The default is 7680 Hz. This property is tunable.

`SidelobeAttenuation`

Chebyshev window sidelobe attenuation

Specify the Chebyshev window attenuation as a real, positive scalar in decibels (dB). This property applies if you set the `Window` property to `'Chebyshev'`. The default is 60 dB. This property is non-tunable.

Methods

reset	Reset internal states of variable bandwidth FIR filter
step	Filter signal using variable bandwidth algorithm

More “Analysis Methods for Filter System Objects” on page 3-2.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object

Common to All System Objects	
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

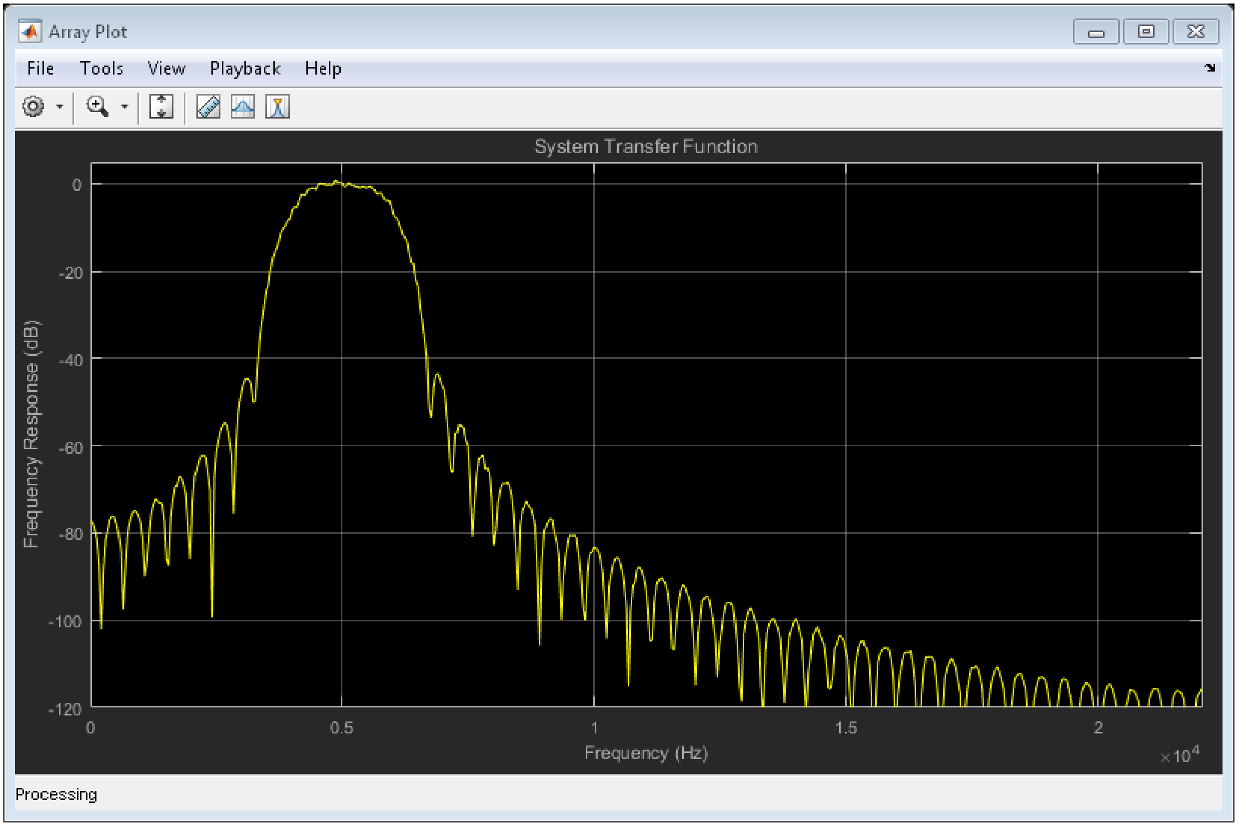
Filtering through a variable bandwidth bandpass FIR filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

This example shows you how to tune the center frequency and the bandwidth of the FIR filter.

```
Fs = 44100; % Input sample rate
% Define a bandpass variable bandwidth FIR filter:
vbw = dsp.VariableBandwidthFIRFilter('FilterType','Bandpass',...
    'FilterOrder',100,...
    'SampleRate',Fs,...
    'CenterFrequency',1e4,...
    'Bandwidth',4e3);
tfe = dsp.TransferFunctionEstimator('FrequencyRange','onesided');
aplot = dsp.ArrayPlot('PlotType','Line',...
    'XOffset',0,...
    'YLimits',[-120 5], ...
    'SampleIncrement', 44100/1024,...
    'YLabel','Frequency Response (dB)',...
    'XLabel','Frequency (Hz)',...
    'Title','System Transfer Function');
FrameLength = 1024;
sine = dsp.SineWave('SamplesPerFrame',FrameLength);
for i=1:500
    % Generate input
    x = sine() + randn(FrameLength,1);
    % Pass input through the filter
    y = vbw(x);
    % Transfer function estimation
    h = tfe(x,y);
    % Plot transfer function
```

```
    aplot(20*log10(abs(h)))
    % Tune bandwidth and center frequency of the FIR filter
    if (i==250)
        vbw.CenterFrequency = 5000;
        vbw.Bandwidth = 2000;
    end
end
```



Algorithms

FIR Transformations

All transformations assume a lowpass filter of length 2N+1.

Lowpass to Lowpass

Consider an ideal lowpass brickwall filter with normalized cutoff frequency ω_{c1}. By taking the inverse Discrete Fourier Transform of the ideal frequency response, and clipping the resulting sequence to length 2N+1, the impulse response is:

$$\begin{aligned}
&\textit{for } n = 0 \\
&h_{LP}(n) = \frac{\omega_c}{\pi}.w(0) \\
&\textit{for } 1 \le |n| \le N \\
&h_{LP}(n) = \frac{\sin(\omega_c n)}{\pi n}.w(n)
\end{aligned}$$

where w(n) is the window vector. The lowpass filter coefficients are tuned to a new cutoff frequency ω_{c2} as follows:

$$\begin{aligned}
&\textit{for } n = 0 \\
&h_{LP}(n) = \frac{\omega_{c2}}{\pi}.\omega(0) \\
&\textit{for } 1 \le |n| \le N \\
&h_{LP}(n) = \frac{\sin(\omega_{c2} n)}{\pi n}.\omega(n)
\end{aligned}$$

There is no need to recompute the window every time you tune the cutoff frequency.

Lowpass to Highpass

Assuming a lowpass filter with normalized 6–dB cutoff frequency ω_c, a highpass filter with the same cutoff frequency can be obtained by taking the complementary of the lowpass frequency response: $H_{HP}(e^{j\omega}) = 1 — H_{LP}(e^{j\omega})$

Taking the inverse discrete Fourier transform of the above response, we get the following highpass filter coefficients:

$$\begin{aligned}
&\textit{for } n = 0 \\
&h_{hp}(n) = 1 - h_{LP}(n) \\
&\textit{for } 1 \le |n| \le N \\
&h_{hp}(n) = -h_{LP}(n)
\end{aligned}$$

Lowpass to Bandpass

A bandpass filtered centered at frequency ω_0 can be obtained by shifting the lowpass response:

$H_{BP}(e^{j\omega}) = H_{LP}(e^{j(\omega-\omega_0)}) + H_{LP}(e^{j(\omega-\omega_0)})$

The bandwidth of the resulting bandpass filter is $2\omega_c$, as measured between the two cutoff frequencies of the bandpass filter. The equivalent bandpass filter coefficients are then:

$$h_{BP}(n) = (e^{j\omega_0 n} + e^{-j\omega_0 n})h_{LP}(n)$$

which can be written as:

$$h_{BP}(n) = 2\cos(\omega_0 n)h_{LP}(n)$$

Lowpass to Bandstop

We can transform a lowpass filter to a bandstop filter by combining the highpass and bandpass transformations. That is, first make the filter bandpass by shifting the lowpass response, and then invert in to get a bandstop response centered at ω_0.

$H_{BS}(e^{j\omega}) = 1 - (H_{LP}(e^{j(\omega-\omega_0)}) + H_{LP}(e^{j(\omega+\omega_0)}))$

This yields the following coefficients:

$$\textit{for } n = 0$$
$$h_{BS}(n) = 1 - 2\cos(\omega_0 n)h_{LP}(n)$$
$$\textit{for } 1 \leq |n| \leq N$$
$$h_{BS}(n) = -2\cos(\omega_0 n)h_{LP}(n)$$

Generalized Transformation

The transformations highlighted above can be combined to transform a lowpass filter to a lowpass, highpass, bandpass or bandstop filter with arbitrary cutoffs.

For example, to transform a lowpass filter with cutoff ω_{c1} to a highpass with cutoff ω_{c2}, you first apply the lowpass-to-lowpass transformation to get a lowpass filter with cutoff ω_{c2}, and then apply the lowpass-to-highpass transformation to get the highpass with cutoff ω_{c2}.

To get a bandpass filter with center frequency ω_0 and bandwidth ß, we first apply the lowpass-to-lowpass transformation to go from a lowpass with cutoff ω_c to a lowpass with cutoff ß/2, and then apply the lowpass-to-bandpass transformation to get the desired bandpass filter. The same approach can be applied to a bandstop filter.

References

[1] Jarske, P.,Y. Neuvo, and S. K. Mitra, "A simple approach to the design of linear phase FIR digital filters with variable characteristics." Signal Processing. Vol. 14, Issue 4, June 1988, pp. 313-326.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

dsp.BiquadFilter | dsp.FIRFilter | dsp.VariableBandwidthIIRFilter | `Variable Bandwidth FIR Filter` | dsp.IIRFilter | dsp.AllpoleFilter | `Variable Bandwidth IIR Filter`

Introduced in R2014a

reset

System object: dsp.VariableBandwidthFIRFilter
Package: dsp

Reset internal states of variable bandwidth FIR filter

Syntax

```
reset(vbw)
```

Description

`reset(vbw)` resets the filter states of the variable bandwidth FIR filter object, `vbw`, to their initial values of 0. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the variable bandwidth FIR filter object to nonzero input data, the states may be nonzero. Invoking the `step` method again without first invoking the `reset` method may produce different outputs for an identical input.

step

System object: dsp.VariableBandwidthFIRFilter
Package: dsp

Filter signal using variable bandwidth algorithm

Syntax

`Y = step(vbw,x)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(vbw,x)` filters the real or complex input signal `X` using the variable bandwidth FIR filter, `vbw`, to produce the output `Y` . The variable bandwidth FIR filter object operates on each channel, which means every column of the input signal independently over successive calls to step method.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.VariableBandwidthIIRFilter System object

Package: dsp

Variable bandwidth IIR filter

Description

The `VariableBandwidthIIRFilter` object filters each channel of the input using IIR filter implementations. It does so while having the capability of tuning the bandwidth. This filter supports double and single precision inputs.

To filter each channel of the input:

1. Define and set up your variable bandwidth IIR filter. See "Construction" on page 4-1859.
2. Call step to filter each channel of the input according to the properties of `dsp.VariableBandwidthIIRFilter`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`vbwIIR = dsp.VariableBandwidthIIRFilter` returns a System object, `vbwIIR`, which independently filters each channel of the input over successive calls to the `step` method. This System object uses a specified IIR filter implementation. The filter's passband frequency may be tuned during the filtering operation. The variable bandwidth IIR filter is designed using the elliptical method. The filter is tuned using IIR spectral transformations based on allpass filters.

```
vbwIIR =
dsp.VariableBandwidthIIRFilter('PropertyName',PropertyValue, ...)
```

returns a variable bandwidth IIR filter System object, `vbwIIR`, with each property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1,Value1,...,NameN,ValueN`).

Properties

`SampleRate`

Input sample rate

Specify the sampling rate of the input in Hertz as a finite numeric scalar. The default is 44.1 kHz. This property is non-tunable.

`FilterType`

Filter type

Specify the type of the filter as one of `'Lowpass' | 'Highpass' | 'Bandpass' | 'Bandstop'`. The default is `'Lowpass'`. This property is non-tunable.

`FilterOrder`

IIR filter order

Specify the order of the IIR filter as a positive integer scalar. The default is 8. This property is non-tunable.

`PassbandFrequency`

Filter passband frequency

Specify the filter passband frequency in Hz as a real, positive scalar, smaller than the `SampleRate/2`. This property applies when you set the `FilterType` property to `'Lowpass'` or `'Highpass'`. The default is 512 Hz. This property is tunable.

`CenterFrequency`

Filter center frequency

Specify the filter center frequency in Hz as a real, positive scalar, smaller than `SampleRate/2`. This property applies when you set the `FilterType` property to `'Bandpass'` or `'Bandstop'`. The default is 11025 Hz. This property is tunable.

Bandwidth

Filter bandwidth

Specify the filter bandwidth in Hertz as a real, positive scalar, smaller than `SampleRate/2`. This property applies when you set the `FilterType` property to `'Bandpass'` or `'Bandstop'`. The default is 7680 Hz. This property is tunable.

PassbandRipple

Filter passband ripple

Specify the filter passband ripple as a real, positive scalar in decibels (dB). The default is 1 dB. This property is non-tunable.

StopbandAttenuation

Filter Stopband attenuation

Specify the filter stopband attenuation as a real, positive scalar in decibels (dB). The default is 60 dB. This property is non-tunable.

Methods

reset	Reset internal states of variable bandwidth IIR filter
step	Filter signal using variable bandwidth algorithm

More “Analysis Methods for Filter System Objects” on page 3-2.

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)

Common to All System Objects	
`release`	Allow System object property value changes

Examples

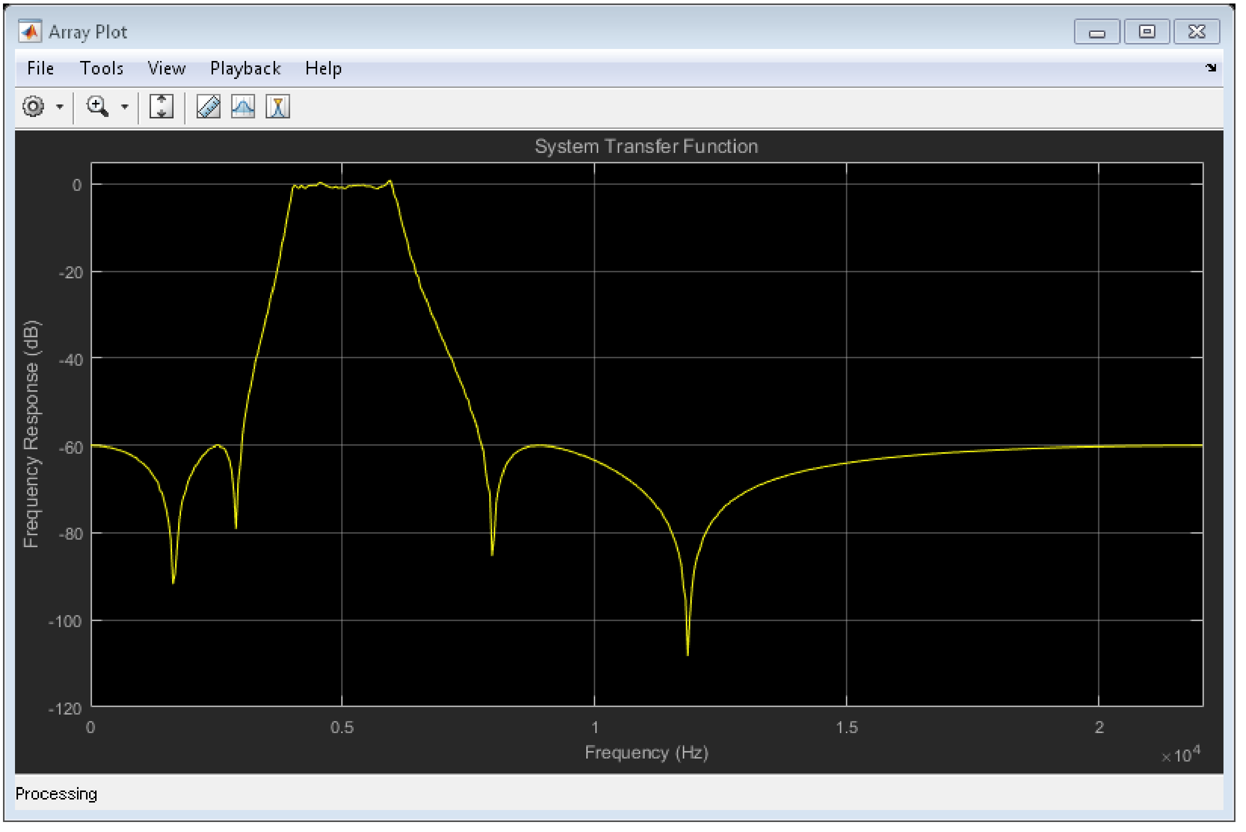
Filtering Through a Variable Bandwidth Bandpass IIR Filter

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

This examples shows you how to tune the center frequency and bandwidth of the IIR filter.

```
Fs = 44100; % Input sample rate
% Define a bandpass variable bandwidth IIR filter:
vbwiir = dsp.VariableBandwidthIIRFilter('FilterType','Bandpass',...
                                      'FilterOrder',8,...
                                      'SampleRate',Fs,...
                                      'CenterFrequency',1e4,...
                                      'Bandwidth',4e3);
tfe = dsp.TransferFunctionEstimator('FrequencyRange','onesided');
aplot = dsp.ArrayPlot('PlotType','Line',...
                      'XOffset',0,...
                      'YLimits',[-120 5], ...
                      'SampleIncrement', 44100/1024,...
                      'YLabel','Frequency Response (dB)',...
                      'XLabel','Frequency (Hz)',...
                      'Title','System Transfer Function');
FrameLength = 1024;
sine = dsp.SineWave('SamplesPerFrame',FrameLength);
for i = 1:500
   % Generate input
   x = sine() + randn(FrameLength,1);
   % Pass input through the filter
   y = vbwiir(x);
   % Transfer function estimation
   h = tfe(x,y);
   % plot transfer function
   aplot(20*log10(abs(h)))
   % Tune bandwidth and center frequency of the IIR filter
```

```
      if (i==250)
        vbwiir.CenterFrequency = 5000;
        vbwiir.Bandwidth = 2000;
      end
    end
```



Algorithms

This filter covers frequency transformations. A lowpass IIR prototype is designed, using the elliptical method by specifying its order, passband frequency, passband ripple and stopband attenuation. The passband ripple and stopband attenuation are equal to the values of the `PassbandRipple` and `StopbandAttenuation` properties. The prototype passband frequency is set to 0.5. If the `FilterType` property is `'Lowpass'` or `'Highpass'`, the prototype's order is equal to the value of `FilterOrder`. If the `FilterType` property is `'Bandpass'` or `'Bandstop'`, the prototype filter order is equal to `FilterOrder/2`. The prototype is a Direct Form II Transposed cascade of second-order sections (Biquad filter). The prototype is transformed into the desired filter using the algorithms used in "Digital Frequency Transformations". Each prototype SOS section is transformed separately. When `FilterType` is `'Lowpass'` or `'Highpass'`, the resulting filter remains a Direct Form II Transposed cascade of second order sections. If the `FilterType` is `'Bandpass'` or `'Bandstop'`, the resulting filter is a cascade of Direct Form II Transposed cascade of fourth order sections.

References

[1] A. G. Constantinides. "Spectral transformations for digital filters", Proc. Inst. Elect. Eng. Vol. 117, No. 8, 1970, pp. 1585-1590.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

dsp.BiquadFilter | dsp.FIRFilter | dsp.VariableBandwidthFIRFilter | `Variable Bandwidth FIR Filter` | dsp.IIRFilter | dsp.AllpoleFilter | `Variable Bandwidth IIR Filter`

Introduced in R2014a

reset

System object: dsp.VariableBandwidthIIRFilter
Package: dsp

Reset internal states of variable bandwidth IIR filter

Syntax

```
reset(vbwIIR)
```

Description

`reset(vbwIIR)` resets the filter states of the variable bandwidth IIR filter object, `vbwIIR`, to their initial values of 0. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the variable bandwidth IIR filter object to nonzero input data, the states may be nonzero. Invoking the `step` method again without first invoking the `reset` method may produce different outputs for an identical input.

step

System object: dsp.VariableBandwidthIIRFilter
Package: dsp

Filter signal using variable bandwidth algorithm

Syntax

`Y = step(vbwIIR,x)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(vbwIIR,x)` filters the real or complex input signal `X` using the variable bandwidth IIR filter, `vbwIIR`, to produce the output `Y`. The variable bandwidth IIR filter object operates on each channel, which means every column of the input signal independently over successive calls to step method.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.VariableFractionalDelay System object

Package: dsp

Delay input by time-varying fractional number of sample periods

Description

The `VariableFractionalDelay` object delays the input by a time-varying fractional number of sample periods.

To delay the input by a time-varying fractional number of sample periods:

1. Define and set up your variable fractional delay object. See "Construction" on page 4-1867.
2. Call step to delay the input according to the properties of `dsp.VariableFractionalDelay`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`vfd = dsp.VariableFractionalDelay` returns a variable fractional delay System object, `vfd`, that delays a discrete-time input by a time-varying fractional number of sample periods.

`vfd = dsp.VariableFractionalDelay('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a variable fractional delay System object, `vfd`, with each property set to the specified value.

Properties

InterpolationMethod

Interpolation method

Specify the method by which the block interpolates between adjacent stored samples to obtain a value for the sample indexed by the input. You can set this property to one of | Linear | FIR | Farrow. When you set this property to FIR, the value of the Bandwidth property is used to generate the FIR filter for interpolation. If the value of the Bandwidth property is 1 (default value), the FIR filter is generated using the firnyquist function. If the value of the Bandwidth property is smaller than 1, the FIR filter is generated using the intfilt function from Signal Processing Toolbox. The default is Linear.

FilterHalfLength

FIR interpolation filter half-length

Specify the half-length of the FIR interpolation filter as a positive scalar integer. This property applies only when you set the InterpolationMethod on page 4-1868 property to FIR. For periodic signals, a larger value of this property (that is, a higher order filter) yields a better estimate of the delayed output sample. Setting this property to a value between 4 and 6 (that is, a 7th to 11th order filter) is usually adequate. The default is 4.

FilterLength

Length of Farrow filter

Specify the length of the FIR filter implemented using the Farrow structure, as a positive scalar integer. This property applies only when you set the InterpolationMethod on page 4-1868 property to Farrow. The default is 4.

InterpolationPointsPerSample

Number of interpolation points per input sample

Specify the number of interpolation points per input sample at which a unique FIR interpolation filter is computed. You must specify the number of interpolation points per input sample as a positive scalar integer. This property applies only when you set the `InterpolationMethod` on page 4-1868 property to `FIR`. The default is `10`.

Bandwidth

Normalized input bandwidth

Specify the bandwidth to which you want to constrain the interpolated output samples. You must enter the bandwidth as a scalar value between `0` and `1`. You can use this property to take advantage of the bandlimited frequency content of the input. For example, if the input signal does not have frequency content above `Fs/4` (where `Fs` is the sampling frequency), you can specify a value of `0.5` for the `Bandwidth` property. A value of `1` for the `Bandwidth` property corresponds to half the sampling frequency (`Fs`). This property applies only when you set the `InterpolationMethod` on page 4-1868property to `FIR`. The default is `1`.

InitialConditions

Initial values in memory

Specify the values with which the object's memory is initialized. The dimensions of this property can vary depending on whether you want fixed or time-varying initial conditions. The default is `0`.

For an M-by-N frame-based input matrix U, you can set the `InitialConditions` property as follows:

- To specify fixed initial conditions, set the `InitialConditions` property to a scalar value. The object initializes every sample of every channel in memory using the value you specify.
- The dimensions you specify for time-varying initial conditions depend on the value of the `InterpolationMethod` property. To specify different time-varying initial conditions for each channel, set the `InitialConditions` property as follows:
  - If you set the `InterpolationMethod` property to `Linear`, set the `InitialConditions` property to an array of size `1`-by-N-by-D, where D is the value of the `MaximumDelay` property.
  - If you set the `InterpolationMethod` property to `FIR` or `Farrow`, set the `InitialConditions` property to an array of size 1-by-N-by-(D+L), where D is the value of the `MaximumDelay` property. For FIR interpolation, `L` is the value of the

FilterHalfLength property. For Farrow interpolation, L equals floor of half the value of the FilterLength property (floor(FilterLength/2)).

MaximumDelay

Maximum delay

Specify the maximum delay the object can produce for any sample. The maximum delay must be a positive scalar integer value. The object clips input delay values greater than the MaximumDelay to the MaximumDelay. The default is 100.

FIRSmallDelayAction

Action for small input delay values in FIR interpolation mode

Specify the action the object should take for small input delay values when using the FIR interpolation method. You can set this property to Clip to the minimum value necessary for centered kernel, or Switch to linear interpolation if kernel cannot be centered. This property applies only when you set the InterpolationMethod on page 4-1868 property to FIR. The default is Clip to the minimum value necessary for centered kernel.

FarrowSmallDelayAction

Action for small input delay values in Farrow interpolation mode

Specify the action the object should take for small input delay values when using the Farrow interpolation method. You can set this property to Clip to the minimum value necessary for centered kernel, or Use off-centered kernel. This property applies only when you set the InterpolationMethod on page 4-1868 property to Farrow. The default is Clip to the minimum value necessary for centered kernel.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent, | Floor | Nearest | Round | Simplest | Zero |. The default is Zero.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | `Wrap` | `Saturate` |. The default is `Wrap`.

CoefficientsDataType

Coefficient word and fraction lengths

Specify the coefficients data type as one of | `Same word length as input` | `Custom` |. The default is `Same word length as input`.

CustomCoefficientsDataType

Coefficients word length

Specify the coefficients word length as an unscaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `CoefficientsDataType` on page 4-1871 property to `Custom`. The default is `numerictype([],32)`.

ProductPolynomialValueDataType

Product polynomial values word and fraction lengths

Specify the product polynomial value data type as one of | `Same as first input` | `Custom` |. This property applies only when you set the `InterpolationMethod` on page 4-1868 property to `Farrow`. The default is `Same as first input`.

CustomProductPolynomialValueDataType

Product polynomial values word and

fraction lengths Specify the product polynomial values fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `InterpolationMethod` on page 4-1868 property to `Farrow` and the `ProductPolynomialValueDataType` on page 4-1871 property to `Custom`. The default is `numerictype([],32,10)`.

AccumulatorPolynomialValueDataType

Accumulator polynomial value word and fraction lengths

Specify the accumulator polynomial value data type as one of | `Same as first input` | `Custom` |. This property applies only when you set the `InterpolationMethod` on page 4-1868 property to `Farrow`. The default is `Same as first input`.

CustomAccumulatorPolynomialValueDataType

Accumulator polynomial value word and fraction lengths

Specify the data type of the accumulator polynomial values as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `InterpolationMethod` on page 4-1868 property to `Farrow` and the `AccumulatorPolynomialValueDataType` on page 4-1871 property to `Custom`. The default is `numerictype([],32,10)`.

MultiplicandPolynomialValueDataType

Multiplicand polynomial value word and fraction lengths

Specify the multiplicand polynomial values data type as one of | `Same as first input` | `Custom` |. This property applies only when you set the `InterpolationMethod` on page 4-1868 property to `Farrow`. The default is `Same as first input`.

CustomMultiplicandPolynomialValueDataType

Multiplicand polynomial value word and fraction lengths

Specify the fixed-point data type of the multiplicand polynomial values as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `InterpolationMethod` on page 4-1868 property to `Farrow` and the `MultiplicandPolynomialValueDataType` on page 4-1872 property to `Custom`. The default is `numerictype([],32,10)`.

ProductDataType

Product word and fraction lengths

Specify the product data type as one of | `Same as first input` | `Custom` |. The default is `Same as first input`.

CustomProductDataType

Product word and fraction lengths

Specify the product data type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `ProductDataType` on page 4-1872 property to `Custom`. The default is `numerictype([],32,10)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator data type as one of | `Same as product` | `Same as first input` | `Custom` |. The default is `Same as product`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the fixed-point accumulator data type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `AccumulatorDataType` on page 4-1873 property to `Custom`. The default is `numerictype([],32,10)`.

OutputDataType

Output word and fraction lengths

Specify the output data type as one of | `Same as accumulator` | `Same as first input` | `Custom` |. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the data type of the output as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` on page 4-1873 property to `Custom`. The default is `numerictype([],32,10)`.

Methods

info	Characteristic information about valid delay range

reset	Reset internal states of variable fractional delay object
step	Delay input by time-varying fractional number of sample periods

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Signal Delay using a Variable Fractional Delay

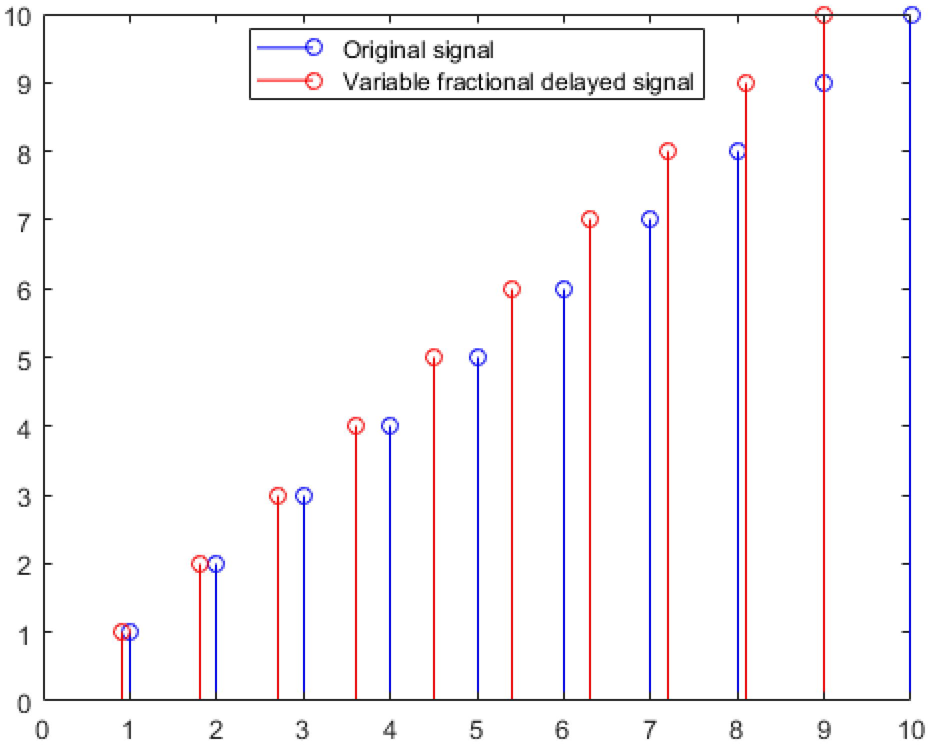
Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Delay a signal by a varying fractional number of sample periods.

```
sr = dsp.SignalSource;
vfd = dsp.VariableFractionalDelay;
sink = dsp.SignalSink;

for ii = 1:10
    delayedsig = vfd(sr(), ii/10);
    sink(delayedsig);
end

sigd = sink.Buffer;
```

The output `sigd` corresponds to the values of the delayed signal that are sampled at fixed-time intervals. For visualization purposes, we can instead plot the time instants at which the amplitudes of signal samples are constant by treating the signals as the sampling instants.

```
stem(sr.Signal, 1:10, 'b')
hold on;
stem(sigd.', 1:10, 'r');
legend('Original signal',...
    'Variable fractional delayed signal', ...
    'Location','best')
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Variable Fractional Delay` block reference page.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.Delay | `Variable Fractional Delay` | dsp.VariableIntegerDelay | dsp.DelayLine

Introduced in R2012a

info

System object: dsp.VariableFractionalDelay
Package: dsp

Characteristic information about valid delay range

Syntax

```
S = info(vfd)
```

Description

`S = info(vfd)` returns a structure, `S` , containing characteristic information about the `VariableFractionalDelay` System object, `vfd` . `S` has one field, `ValidDelayRange` which is a string containing the possible range of delay values based on the current property values of the `VariableFractionalDelay` object. The `ValidDelayRange` is in the format `[MinValidDelay, MaxValidDelay]`. The object clips all input delay values to be within this `ValidDelayRange`.

reset

System object: dsp.VariableFractionalDelay
Package: dsp

Reset internal states of variable fractional delay object

Syntax

`reset(vfd)`

Description

`reset(vfd)` resets the internal states of the `VariableFractionalDelay` System object `vfd` to their initial values.

step

System object: dsp.VariableFractionalDelay
Package: dsp

Delay input by time-varying fractional number of sample periods

Syntax

`Y = step(vfd,X,D)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(vfd,X,D)` delays the input `X` by `D` samples, where `D` should be less than or equal to the value specified in the `MaximumDelay` property.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.VariableIntegerDelay System object

Package: dsp

Delay input by time-varying integer number of sample periods

Description

Note: Certain functionality of this object will be removed in future releases. See "Functionality being removed or replaced for blocks and System objects".

The `VariableIntegerDelay` object delays input by time-varying integer number of sample periods.

To delay the input by a time-varying integer number of sample periods:

1. Define and set up your variable integer delay object. See "Construction" on page 4-1880.
2. Call step to delay the input according to the properties of `dsp.VariableIntegerDelay`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`vid = dsp.VariableIntegerDelay` returns a variable integer delay System object, `vid`, that delays discrete-time input by a time-varying integer number of sample periods.

`vid = dsp.VariableIntegerDelay('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a variable integer delay System object, `vid`, with each property set to the specified value.

Properties

`MaximumDelay`

Maximum delay

Specify the maximum delay the object can produce for any sample. The maximum delay must be a positive scalar integer value. The object clips input delay values greater than the `MaximumDelay` to the `MaximumDelay`. The default is `100`.

`InitialConditions`

Initial values in memory

Specify the values with which the object's memory is initialized. The dimensions of this property can vary depending on whether you want fixed or time-varying initial conditions. The default is `0`.

For an M-by-N frame-based input matrix U, you can set the `InitialConditions` property as follows:

- To specify fixed initial conditions, set the `InitialConditions` property to a scalar value. The object initializes every sample of every channel in memory using the value you specify.
- To specify different time-varying initial conditions for each channel, set the `InitialConditions` property to an array of size 1-by-N-by-D, where D is the value of the `MaximumDelay` property.

`DirectFeedthrough`

Allow direct feedthrough

When you set this property to `true`, the object allows direct feedthrough. When you set this property to `false`, the object increases the minimum possible delay by one. The default is `true`.

Methods

reset	Reset internal states of variable integer delay object

step	Delay input by time-varying integer number of sample periods

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

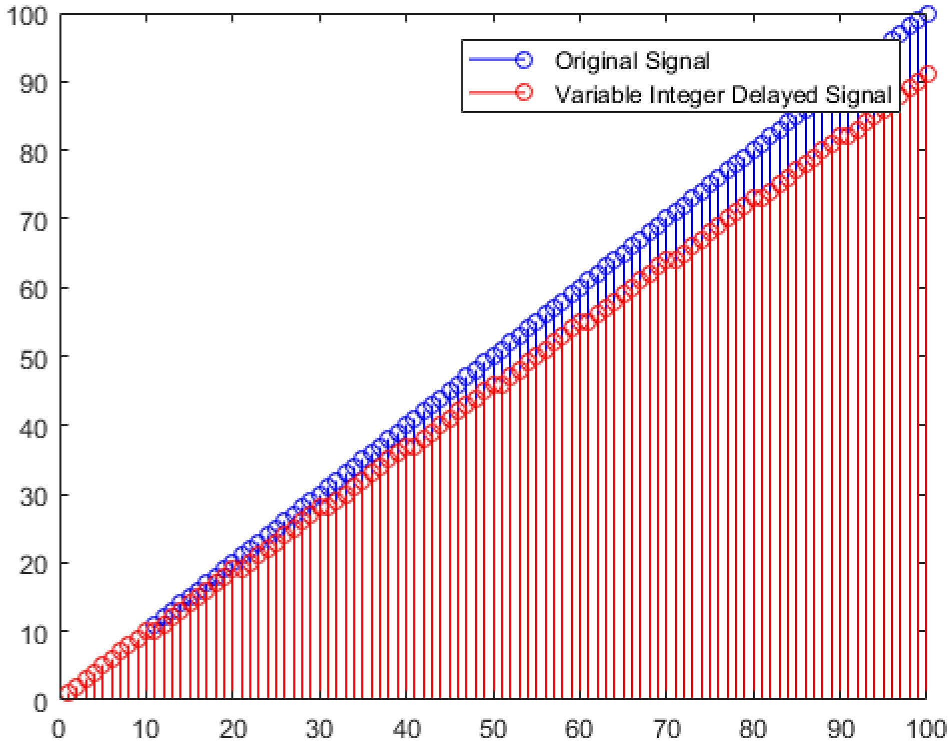
Examples

Delay a Signal with VariableIntegerDelay Object

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Delay a signal by a varying number of integer sample periods.

```
vid = dsp.VariableIntegerDelay;
x = 1:100;
ii = 0;
k = 0;
yout = [];

while(ii+10 <= 100)
     y = vid(x(ii+1:ii+10)',k*ones(10,1));
     yout = [yout;y];
     ii = ii+10;
     k = k+1;
end

stem(x,'b');
hold on; stem(yout,'r');
legend('Original Signal', 'Variable Integer Delayed Signal')
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the Variable Integer Delay block reference page. The object properties correspond to the block properties, except:

When you set the `DirectFeedthrough` property of the System object to `true`, the object allows direct feedthrough. This behavior is different from the way the block behaves when you select the corresponding **Disable direct feedthrough by increasing minimum possible delay by one** check box on the block dialog. When you enable this block parameter, the block does not allow direct feedthrough.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.Delay | dsp.VariableFractionalDelay | dsp.DelayLine

Introduced in R2012a

reset

System object: dsp.VariableIntegerDelay
Package: dsp

Reset internal states of variable integer delay object

Syntax

`reset(vid)`

Description

`reset(vid)` resets the internal states of the `VariableIntegerDelay` System object `vid` to their initial values.

step

System object: dsp.VariableIntegerDelay
Package: dsp

Delay input by time-varying integer number of sample periods

Syntax

```
Y = step(vid,X,D)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(vid,X,D)` delays the input `X` by `D` samples, where `D` should be less than or equal to the value specified in the `MaximumDelay` property and greater than or equal to `0`. The object clips delay values greater than the `MaximumDelay` to the `MaximumDelay`, and clips values less than zero to zero. If you enter a noninteger delay value, the object rounds that value to the nearest integer value.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

dsp.Variance System object

Package: dsp

Variance of input or sequence of inputs

Description

The `Variance` object computes variance for an input or sequence of inputs.

To compute the variance of an input or sequence of inputs:

1. Define and set up your variance System object. See "Construction" on page 4-1887.
2. Call step to compute the variance according to the properties of `dsp.Variance`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Note: The `Running` mode in the `dsp.Variance` System object will be removed in a future release. To compute the running variance in MATLAB, use the dsp.MovingVariance System object instead.

Construction

`var = dsp.Variance` returns a variance System object, `var`, that computes the variance of an input or a sequence of inputs over the specified `Dimension`.

`var = dsp.Variance('PropertyName',PropertyValue,...)` returns a variance System object, `var`, with each specified property set to the specified value.

Properties

`RunningVariance`

Enable calculation over time

Set this property to `true` to enable variance calculation over successive calls to the `step` method. The default is `false`.

`ResetInputPort`

Enable reset input port

Set this property to `true` to enable reset input port. When you set the property to `true`, specify a reset input for the step method. The running variance resets anytime the variance object achieves the condition you specify for the `ResetCondition on page 4-1888` property. This property applies when you set the `RunningVariance on page 4-1888` property to `true`. The default is `false`.

`ResetCondition`

Reset condition for running variance mode

Specify which event resets the running variance as one of `| Rising edge | Falling edge | Either edge | Non-zero |`. This property applies when you set the `ResetInputPort on page 4-1888` property to `true`.

`Dimension`

Dimension to operate along

Specify how the object performs the variance calculation over the data as one of `| All | Row | Column | Custom |`. This property applies when you set the `RunningVariance on page 4-1888` property to `false`. The default is `Column`.

`CustomDimension`

Numerical dimension to operate along

Specify the input signal dimension (one-based value) the object uses to compute variance. The cannot exceed the number of dimensions in the input signal. This property applies when you set the `Dimension on page 4-1888` property to `Custom`. The default is `1`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap.

InputSquaredProductDataType

Input-squared product word and fraction lengths

Specify the input-squared product fixed-point data type as one of | Same as input | Custom |. The default is Same as input.

CustomInputSquaredProductDataType

Input-squared product word and fraction lengths

Specify the input-squared product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the InputSquaredProductDataType on page 4-1889 property to Custom. The default is numerictype([],32,15).

InputSumSquaredProductDataType

Input-sum-squared product word and fraction lengths

Specify the input-sum-squared product fixed-point data type as one of | Same as input-squared product | Custom |. The default is Same as input-squared product.

CustomInputSumSquaredProductDataType

Input-sum-squared product word and fraction lengths

Specify the input-sum-squared product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the

InputSumSquaredProductDataType on page 4-1889 property to Custom. The default is numerictype([],32,23).

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of | Same as input-squared product | Same as input | Custom |. The default is Same as input-squared product.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the AccumulatorDataType on page 4-1890 property to Custom. The default is numerictype([],32,0).

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of | Same as input-squared product | Same as input | Custom |. The default is Same as input-squared product.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled numerictype object with a Signedness of Auto. This property only applies when the OutputDataType on page 4-1890 property to Custom. The default is numerictype([],16,0).

Methods

reset	Reset variance to zero
step	Variance of input

Common to All System Objects	
clone	Create System object with same property values

Common to All System Objects	
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Running Variance

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Compute the running variance of the signal. That is, compute the variance of each sample in the input signal with respect to all the previous samples.

```
var = dsp.Variance;
var.RunningVariance = true;
input = randn(100,1);
variance = var(input);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Variance` block reference page. The object properties correspond to the block parameters, except:

- **Reset port** block parameter corresponds to both the `ResetCondition` and the `ResetInputPort` object properties.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

See Also

System Objects

dsp.MovingVariance | dsp.MovingStandardDeviation | dsp.MovingRMS | dsp.RMS | dsp.Mean | dsp.Minimum

Blocks

`Moving RMS` | `Moving Standard Deviation` | `Moving Variance` | `RMS` | `Standard Deviation` | `Variance`

Introduced in R2012a

reset

System object: dsp.Variance
Package: dsp

Reset variance to zero

Syntax

```
reset(var)
```

Description

`reset(var)` sets the variance to zero when the `RunningVariance` property is `true`.

step

System object: dsp.Variance
Package: dsp

Variance of input

Syntax

```
Y = step(var,X)
Y = step(var,X,R)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(var,X)` computes the variance, `Y`, of input `X` over successive calls to the `step` method, when the `RunningVariance` property is `true`.

`Y = step(var,X,R)` resets its state based on the value of reset signal `R` , the `ResetInputPort` property and the `ResetCondition` property. This option applies when the `RunningVariance` property is `true` and the `ResetInputPort` property is set to `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.VectorQuantizerDecoder System object

Package: dsp

Vector quantizer codeword for given index value

Description

The `VectorQuantizerDecoder` object returns the vector quantizer codeword for a given index value. Each column of the `Codebook` property is a codeword.

To obtain the vector quantizer codeword for a given index value:

1. Define and set up your vector quantizer decoder. See "Construction" on page 4-1895.
2. Call step to get the vector quantizer codeword according to the properties of `dsp.VectorQuantizerDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`vqdec = dsp.VectorQuantizerDecoder` creates a vector quantizer decoder System object, `vqdec`, that returns a vector quantizer codeword corresponding to a given, zero-based index value.

`vqdec = dsp.VectorQuantizerDecoder('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a vector quantizer decoder, `vqdec`, with each specified property set to the specified value.

Properties

`CodebookSource`

Source of codebook values

Specify the codebook source as `Property` or `Input port`. When you select `Property`, the object reads the codebook from the `codebook` on page 4-1896 property. When you select `Input port`, the object reads the codebook from the input of the `step` method.

The default is `Property`.

`Codebook`

codebook

Specify quantized output values as a k-by-N matrix, where $k \geq 1$ and $N \geq 1$. Each column of the codebook matrix is a codeword, and each codeword corresponds to an index value. This property applies when you set the `CodebookSource on page 4-1896` property to `Property`. The default is:

$$\begin{bmatrix} 1.5 & 13.3 & 136.4 & 6.8 \\ 2.5 & 14.3 & 137.4 & 7.8 \\ 3.5 & 15.3 & 138.4 & 8.8 \end{bmatrix}$$

This property is tunable.

`OutputDataType`

Data type of codebook and quantized output

Specify the data type of the codebook and quantized output values as: `Same as input`, `double`, `single` or `Custom`. This property applies only when you set `CodebookSource` on page 4-1896 to `Property`. The default is `double`.

Fixed-Point Properties

`CustomOutputDataType`

Output word and fraction lengths

Specify the output fixed-point type as a signed or unsigned `numerictype` object. This property applies only when you set the `OutputDataType` property to `Custom`. The default is `numerictype(true,16)`.

Methods

step	Perform vector quantization decoding

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Determine Vector Quantizer Codeword

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Given index values as an input, determine the corresponding vector quantized codewords for a specified codebook.

```
vqdec = dsp.VectorQuantizerDecoder;
vqdec.Codebook = [1 10 100;2 20 200;3 30 300];
indices = uint8([1 0 2 0]);
qout = vqdec(indices)
```

```
qout =

    10     1   100     1
    20     2   200     2
    30     3   300     3
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the `Vector Quantizer Decoder` block reference page. The object properties correspond to the block parameters, except:
There is no object property that directly corresponds to the **Action for out of range index value** block parameter. The object sets any index values less than `0` to `0` and any index values greater than or equal to N to N-1.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.VectorQuantizerEncoder | dsp.ScalarQuantizerDecoder

Introduced in R2012a

step

System object: dsp.VectorQuantizerDecoder
Package: dsp

Perform vector quantization decoding

Syntax

```
Q = step(vqdec,I)
Q = step(vqdec,I,C)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Q = step(vqdec,I)` returns the quantized output values `Q` corresponding to the input indices `I`. The data type of `I` can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. The `OutputDataType` property determines the data type of `Q`.

`Q = step(vqdec,I,C)` uses input `C` as the `codebook` values when the `CodebookSource` property is `Input port`. The data type of `C` can be `double`, `single`, or `fixed-point`. The output `Q` has the same data type as the `codebook` input `C`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.VectorQuantizerEncoder System object

Package: dsp

Vector quantization encoding

Description

The `VectorQuantizerEncoder` object performs vector quantization encoding. The object finds the nearest codeword by computing a distortion based on Euclidean or weighted Euclidean distance.

To perform vector quantization encoding:

1. Define and set up your vector quantizer encoder. See "Construction" on page 4-1900.
2. Call step to perform the quantization encoding according to the properties of `dsp.VectorQuantizerEncoder`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`vqenc = dsp.VectorQuantizerEncoder` returns a vector quantizer encoder System object, `vqenc`. This object finds a zero-based index of the nearest codeword for each given input column vector.

`vqenc = dsp.VectorQuantizerEncoder('`*`PropertyName`*`',`*`PropertyValue`*`,...)` returns a vector quantizer encoder System object, `vqenc`, with each specified property set to the specified value.

Properties

CodebookSource

Source of codebook values

Specify how to determine the codebook values as `Property` or `Input port`. The default is `Property`.

Codebook

Codebook

Specify the codebook to which the input column vector or matrix is compared, as a *k*-by-*N* matrix. Each column of the codebook matrix is a codeword, and each codeword corresponds to an index value. The codeword vectors must have the same number of rows as the input. The first codeword vector corresponds to an index value of `0`, the second codeword vector corresponds to an index value of `1`, and so on. This property applies when you set the `CodebookSource on page 4-1901` property to `Property`. The default is:

$$\begin{bmatrix} 1.5 & 13.3 & 136.4 & 6.8 \\ 2.5 & 14.3 & 137.4 & 7.8 \\ 3.5 & 15.3 & 138.4 & 8.8 \end{bmatrix}$$

This property is tunable.

DistortionMeasure

Distortion calculation method

Specify how to calculate the distortion as `Squared error` or `Weighted squared error`. If you set this property to `Squared error`, the object calculates the distortion by evaluating the Euclidean distance between the input column vector and each codeword in the codebook. If you set this property to `Weighted squared error`, the object calculates the distortion by evaluating a weighted Euclidean distance using a weighting factor to emphasize or deemphasize certain input values. The default is `Squared error`.

WeightsSource

Source of weighting factor

Specify how to determine weighting factor as `Property` or `Input port`. This property applies when you set the `DistortionMeasure on page 4-1901` property to `Weighted squared error`. The default is `Property`.

Weights

Weighting factor

Specify the weighting factor as a vector of length equal to the number of rows of the input. This property applies when you set the `DistortionMeasure on page 4-1901` property to `Weighted squared error` and `WeightsSource on page 4-1901` property is `Property`. The default is [1 1 1]. This property is tunable.

TiebreakerRule

Behavior when input column vector is equidistant from two codewords.

Specify whether to represent the input column vector by the lower index valued codeword or higher indexed valued codeword when an input column vector is equidistant from two codewords. You can set this property to `Choose the lower index` or `Choose the higher index`. The default is `Choose the lower index`.

CodewordOutputPort

Enable output of codeword value

Set this property to `true` to output the codeword vectors nearest to the input column vectors. The default is `false`.

QuantizationErrorOutputPort

Enable output of quantization error

Set this property to `true` to output the quantization error value that results when the object represents the input column vector by the nearest codeword. The default is `false`.

OutputIndexDataType

Data type of index output

Specify the data type of the index output as: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`. The default is `int32`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest` or `Zero`. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType on page 4-1903` property to `Custom`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Same as input`, or `Custom`. The default is `Same as product`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType on page 4-1903` property to`Custom`.

Methods

step	Perform vector quantization encoding

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Find Indices of Nearest Codewords

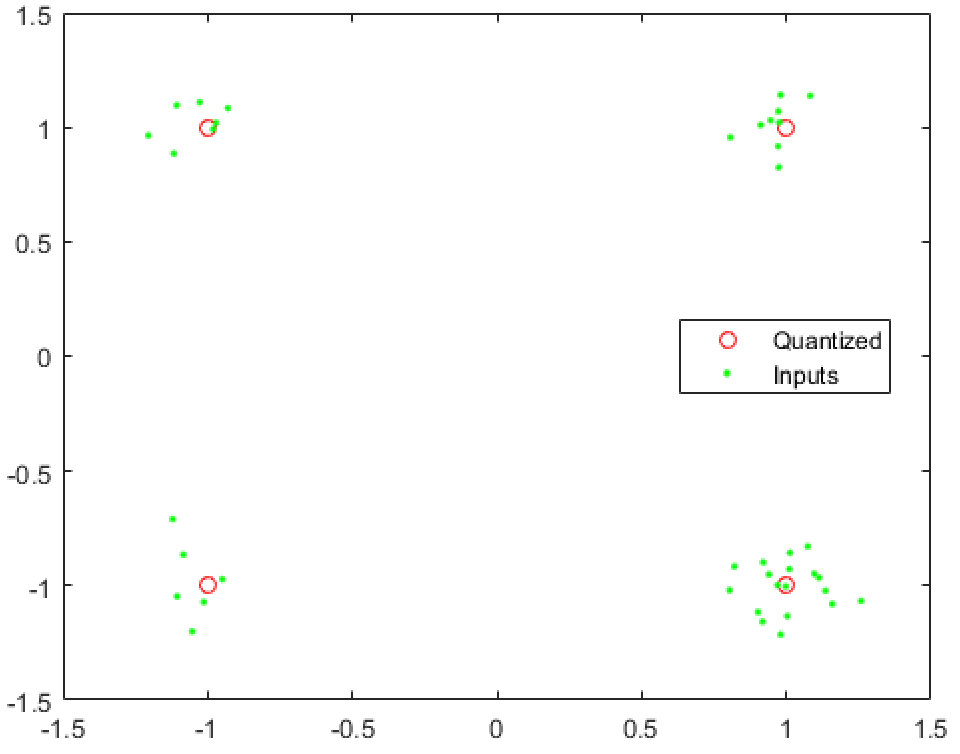
Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Find the indices of nearest codewords based on Euclidean distances.

```
vqenc = dsp.VectorQuantizerEncoder(...
    'Codebook', [-1 -1 1 1;1 -1 -1 1], ...
    'CodewordOutputPort', true, ...
    'QuantizationErrorOutputPort', true, ...
    'OutputIndexDataType', 'uint8');
```

Generate an input signal with some additive noise

```
x = sign(rand(2,40)-0.5) + 0.1*randn(2,40);
[ind, cw, err] = vqenc(x);
plot(cw(1,:), cw(2,:), 'rO', x(1,:), x(2,:), 'g.');
legend('Quantized', 'Inputs', 'location', 'best');
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Vector Quantizer Encoder` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also
dsp.VectorQuantizerDecoder | dsp.ScalarQuantizerEncoder

Introduced in R2012a

step

System object: dsp.VectorQuantizerEncoder
Package: dsp

Perform vector quantization encoding

Syntax

```
INDEX = step(vqenc,INPUT)
INDEX = step(vqenc,...,CODEBOOK)
INDEX = step(vqenc,...,WEIGHTS)
[..., CODEWORD] = step(vqenc, ...)
[..., QERR] = step(vqenc, ...)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`INDEX = step(vqenc,INPUT)` returns `INDEX`, a scalar or column vector representing the quantization region(s) to which `INPUT` belongs. `INPUT` can be a column vector of size k-by-1 or an M multichannel matrix of dimensions k-by-M, where k is the length of each codeword in the codebook. All inputs to the object can be real floating-point or fixed-point values and must be of the same data type. The output index values can be signed or unsigned integers.

`INDEX = step(vqenc,...,CODEBOOK)` uses the codebook given in input `CODEBOOK`, a k-by-N matrix with N codewords each of length k. This option is available when the `CodebookSource` property is `Input port`.

`INDEX = step(vqenc,...,WEIGHTS)` uses the input vector `WEIGHTS` to emphasize or de-emphasize certain input values when calculating the distortion measure. `WEIGHTS` must be a vector of length equal to the number of rows of `INPUT`. This option is available when the `DistortionMeasure` property is `Weighted squared error` and the `WeightsSource` property is `Input port`.

`[..., CODEWORD] = step(vqenc, ...)` outputs the `CODEWORD` values that correspond to each index value when the `CodewordOutputPort` property is `true`.

`[..., QERR] = step(vqenc, ...)` outputs the quantization error `QERR` for each input value when the `QuantizationErrorOutputPort` property is `true`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.Window System object

Package: dsp

Window object

Description

The `Window` object applies a window to an input signal.

To apply a window to an input signal:

1. Define and set up your window. See “Construction” on page 4-1909.
2. Call step to apply the window according to the properties of `dsp.Window`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`win = dsp.Window` returns a window object, `win`, that applies a Hamming window with symmetric sampling.

`win = dsp.Window('`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a window object, `win`, with each property set to the specified value.

`win = dsp.Window(WINDOW,'`*`PropertyName`*`',`*`PropertyValue`*`, ...)` returns a window object, `win`, with the `WindowFunction` property set to `WINDOW` and other properties set to the specified values.

Properties

`WindowFunction`

Type of window

Specify the type of window to apply as `Bartlett`, `Blackman`, `Boxcar`, `Chebyshev`, `Hamming` , `Hann`, `Hanning`, `Kaiser`, `Taylor`, `Triang`. The default is `Hamming`. If you run this object in simulation, this property is tunable. When you generate code from a function or script that contains this object, and run the generated code, this property is not tunable.

`WeightsOutputPort`

Enable the output of window weights

Set this property to `true` to output the window weights. The weights are an M-by-1 vector with M equal to the first dimension of the input. The default is `false`.

`StopbandAttenuation`

Level of stopband attenuation in decibels

Specify the level of stopband attenuation in decibels. This property only applies when the `WindowFunction` on page 4-1910 property is `Chebyshev`. The default is 50. This property is tunable.

`Beta`

Kaiser window parameter

Specify the Kaiser window parameter as a real number. Increasing the absolute value of `Beta` widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. This property only applies when `WindowFunction` on page 4-1910 property is `Kaiser`. The default is 10. This property is tunable.

`NumConstantSidelobes`

Number of constant sidelobes

Specify the number of constant sidelobes as an integer greater than zero. This property only applies when `WindowFunction` on page 4-1910 property is `Taylor`. The default is 4. This property is tunable.

MaximumSidelobeLevel

Maximum sidelobe level relative to mainlobe

Specify, in decibels, the maximum sidelobe level relative to the mainlobe as a real number less than or equal to zero. The default is –30, which produces sidelobes with peaks 30 dB down from the mainlobe peak. This property only applies when `WindowFunction` on page 4-1910 property is `Taylor`. This property is tunable.

Sampling

Window sampling for generalized-cosine windows

Specify the window sampling for generalized-cosine windows as `Symmetric` or `Periodic`. This property only applies when `WindowFunction` on page 4-1910 property is `Blackman`, `Hamming`, `Hann`, or `Hanning`. If you run this object in simulation, this property is tunable. When you generate code from a function or script that contains this object, and run the generated code, this property is not tunable.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects”.

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` or `Saturate`. The default is `Wrap`. This property applies only if the object is not in full precision mode.

WindowDataType

Window word and fraction lengths

Specify the window fixed-point data type as one of `Same word length as input` or `Custom`. The default is `Same word length as input`.

CustomWindowDataType

Window word and fraction lengths

Specify the window fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `WindowDataType` on page 4-1912 property to `Custom`. The default is `numerictype([],16,15)`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of `Full precision`, `Same as input`, or `Custom`. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` on page 4-1912 property to `Custom`. The default is `numerictype([],16,15)`.

OutputDataType

Output data type

Specify the output fixed-point data type as one of `Same as product`, `Same as input`, `Custom`. The default is `Same as product`.

`CustomOutputDataType`

Output word and fraction lengths

Specify the output fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` on page 4-1912 property to `Custom`. The default is `numerictype([],16,15)`.

Methods

step	Multiply input by window

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

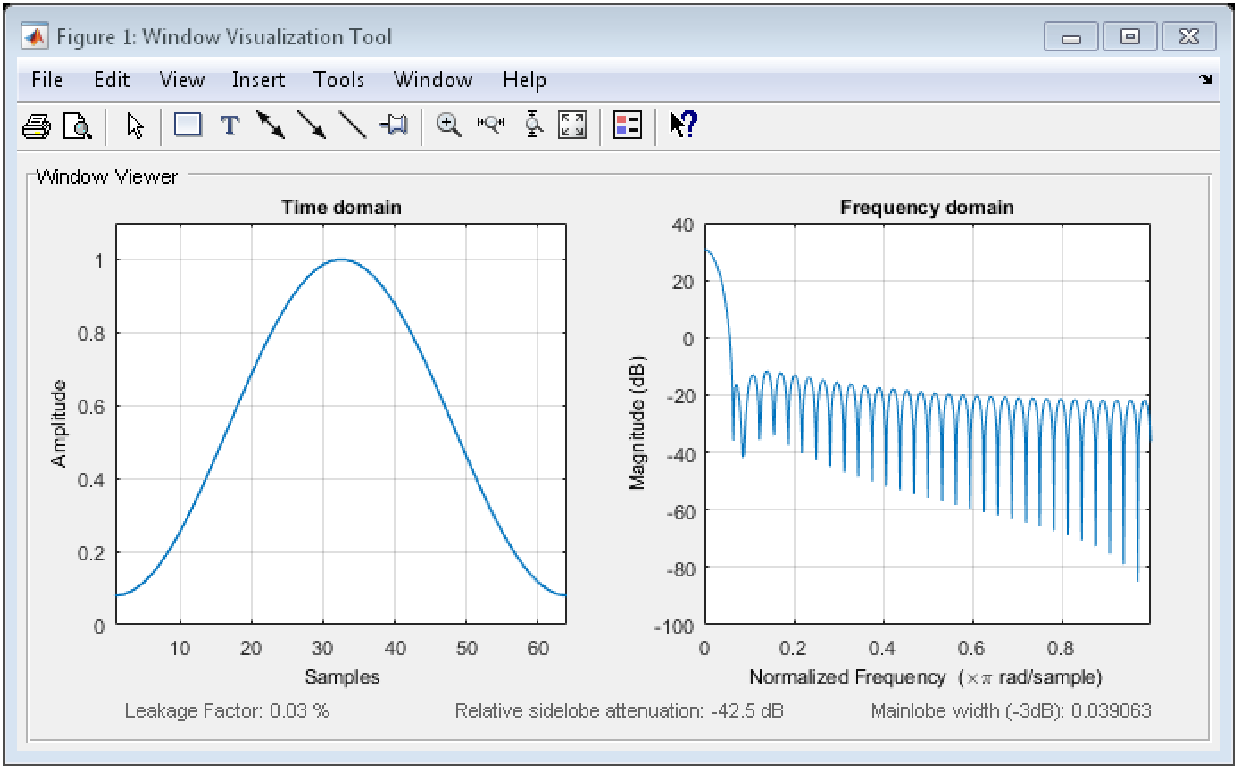
Apply Hamming Window To the Input

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

```
win = dsp.Window( ...
    'WindowFunction', 'Hamming', ...
    'WeightsOutputPort',true);
x = rand(64,1);
[y, w] = win(x);
```

View the window's time and frequency domain responses

```
wvtool(w);
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Window Function` block reference page. The object properties correspond to the block parameters, except:

- **Operation** — The window object does not support the `Generate window` option.
- **Operation** — The `Generate and apply window` option on the block corresponds to the `WeightsOutputPort` on page 4-1910 property set to `true` on the window object.
- The window object only supports frame-based processing.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- This object has no tunable properties for code generation.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

dsp.FFT | sigwin.blackman | sigwin.hamming | sigwin.kaiser | sigwin.triang | sigwin.bartlett | sigwin.chebwin | sigwin.hann | sigwin.taylorwin | `wvtool`

Introduced in R2012a

step

System object: dsp.Window
Package: dsp

Multiply input by window

Syntax

```
Y = step(win,X)
[Y, W] = step(win,X)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(win,X)` generates the windowed output, `Y` , of the input, `X`, using the specified window.

`[Y, W] = step(win,X)` returns the window values `W` when the `WeightsOutputPort` property is `true`.

This object supports only frame-based processing. To see the effect of the window, the data must have a frame size of at least 2 in each channel.

dsp.ZeroCrossingDetector System object

Package: dsp

Zero crossing detector

Description

The `ZeroCrossingDetector` object counts the number of times the signal crosses zero, or changes sign. The zero crossing detector supports both floating-point and fixed-point data types.

To count the number of times a signal crosses zero or changes sign:

1. Define and set up your zero crossing detector. See "Construction" on page 4-1917.
2. Call step to count the number of times according to the properties of `dsp.ZeroCrossingDetector`. The behavior of `step` is specific to each object in the toolbox.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`zcd = dsp.ZeroCrossingDetector` returns a zero crossing detector object, `zcd`, that counts the number of zero crossings in the real-valued,, floating-point, or fixed-point frame-based vector or matrix.

Methods

step	Count zero crossings in input

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

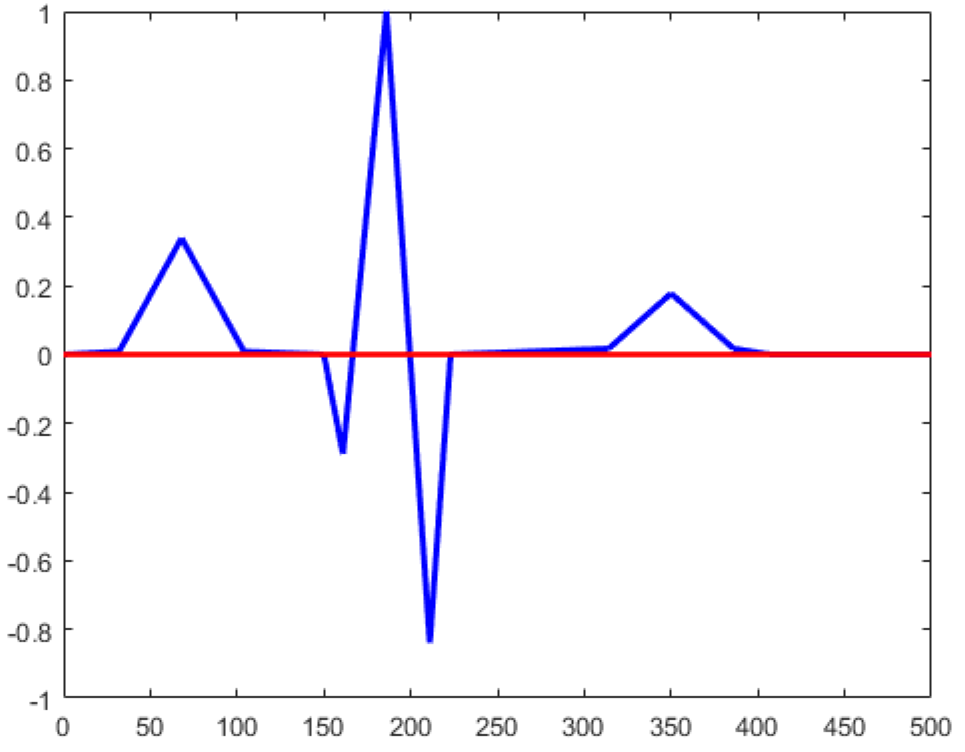
Examples

Determine The Number of Zero Crossings

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Find number of zero crossings in electrocardiogram data.

```
EcgData = ecg(500)';
zcd = dsp.ZeroCrossingDetector;
numZeroCross = zcd(EcgData);  % Equal to 4
plot(1:500,EcgData,'b',[0 500],[0 0],'r','linewidth',2);
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the `Zero Crossing` block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Introduced in R2012a

step

System object: dsp.ZeroCrossingDetector
Package: dsp

Count zero crossings in input

Syntax

`Y = step(zcd,X)`

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(zcd,X)` counts the number of zero crossings, `Y`, in the vector or matrix input `X`.

Note: `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties (MATLAB) and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

spbscopes.SpectrumAnalyzerConfiguration class

Package: spbscopes

Configure Spectrum Analyzer for programmatic access

Description

The `spbscopes.SpectrumAnalyzerConfiguration` object contains the scope configuration information for the Spectrum Analyzer block.

Construction

Call the `get_param` function, specifying a Spectrum Analyzer block.

`htsc = get_param(gcbh,'ScopeConfiguration')` constructs a new Spectrum Analyzer Configuration object.

Properties

`AxesLayout`

Orientation of the spectrum and spectrogram

Specify the layout type as one of `'Horizontal'` or `'Vertical'`. A vertical layout stacks the spectrum above the spectrogram. A horizontal layout puts the two views side-by-side. This property applies when the `ViewType` is set to `'Spectrum and spectrogram'`.

This property is Tunable (Simulink).

Default: `'Vertical'`

`CenterFrequency`

Frequency over which frequency span is centered

Specify as a real scalar the center frequency, in hertz, of the frequency span over which the Spectrum Analyzer computes and plots the spectrum. This property applies when you set the FrequencySpan property to `'Span and center frequency'`. The overall frequency span, defined by the Span and CenterFrequency properties, must fall within the Nyquist frequency interval. When the PlotAsTwoSidedSpectrum property is set to `true`, the interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. When the `PlotAsTwoSidedSpectrum` property is set to `false`, the interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz.

This property is Tunable (Simulink).

Default: `0`

ChannelNames

Channel names

Specify as a cell array of character vectors the input channels names to display in the legend, the Style dialog, and in some Measurements panels. The channel names only appear in the legend when the `ShowLegend` property is `true`. The default is an empty cell array, which labels the channels as `Channel 1`, `Channel 2`, etc.

This property is Tunable (Simulink).

FFTLength

FFT length

Specify as a positive, scalar integer the length of the FFT that the Spectrum Analyzer uses to compute spectral estimates. This property applies only when you set the FrequencyResolutionMethod property to `'WindowLength'` and the FFTLengthSource property to `'Property'`. The `FFTLength` must be greater than or equal to the WindowLength. You cannot control the FFT length when the FrequencyResolutionMethod property is `'RBW'`. In that case, the FFT length is set as the window length required to achieve the specified resolution bandwidth value or 1024, whichever is larger.

This property is Tunable (Simulink).

Default: `128`

FFTLengthSource

Source of the FFT length value

Specify the source of the FFT length value as one of `'Auto'` or `'Property'`.

- When you set this property to `'Auto'`, the Spectrum Analyzer sets the FFT length to the window length specified in the WindowLength property or to 1024, whichever is larger.
- When you set this property to `'Property'`, specify the number of FFT points using the FFTLength property. The `FFTLength` must be greater than the `WindowLength`.

This property applies only when you set the FrequencyResolutionMethod property to `'WindowLength'`.

`FFTLengthSource` property is Tunable (Simulink).

Default: `'Auto'`

FrequencyOffset

Frequency offset

Specify as a real, scalar value to apply the same frequency offset, in Hertz, to all channels. To apply a specific frequency offset for each channel, specify a vector of real, scalar values. The vector length must be equal to number of input channels. The *frequency*-axis values are offset by the values specified in this property. The overall span must fall within the *Nyquist* frequency interval. When the PlotAsTwoSidedSpectrum property is `true`, the Nyquist interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. When you set the `PlotAsTwoSidedSpectrum` property to `false`, the Nyquist interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. You can control the overall span in different ways based on how you set the FrequencySpan property.

This property is Tunable (Simulink).

Default: `0`

FrequencyResolutionMethod

Frequency resolution method

Specify how to control the frequency resolution of the spectrum analyzer as either 'RBW' or 'WindowLength'. When you set this property to 'RBW', the RBWSource and RBW properties control the frequency resolution (in Hz) of the analyzer. When you set this property to 'WindowLength', the WindowLength property controls the frequency resolution.

In this case, the frequency resolution of the analyzer is defined as $\frac{NENBW * Fs}{N_{window}}$ where *NENBW* is the normalized effective noise bandwidth of the window currently specified in the Window property. You can control the number of FFT points only when the FrequencyResolutionMethod property is 'WindowLength'. When the FrequencyResolutionMethod property is 'RBW', the FFT length is the window length that results from achieving the specified RBW value or 1024, whichever is larger.

This property is Tunable (Simulink).

Default: 'RBW'

FrequencyScale

Frequency scale

Specify the frequency scale as either 'Linear' or 'Log'. When you set the FrequencyScale property to 'Log', the Spectrum Analyzer displays the frequencies on the *x*-axis on a logarithmic scale. To use the 'Log' setting, you must also set the PlotAsTwoSidedSpectrum property to false. When the PlotAsTwoSidedSpectrum property is true, you must set this property to 'Linear'.

This property is Tunable (Simulink).

Default: 'Linear'

FrequencySpan

Frequency span mode

Specify the frequency span mode as one of 'Full', 'Span and center frequency', or 'Start and stop frequencies'.

- When you set this property to `'Full'`, the Spectrum Analyzer computes and plots the spectrum over the entire *Nyquist* frequency interval. When the PlotAsTwoSidedSpectrum property is `true`, the Nyquist interval is $\left[-\frac{SampleRate}{2},\frac{SampleRate}{2}\right]+FrequencyOffset$ hertz. If you set the `PlotAsTwoSidedSpectrum` property to `false`, the Nyquist interval is $\left[0,\frac{SampleRate}{2}\right]+FrequencyOffset$ hertz.
- When you set this property to `'Span and center frequency'`, the Spectrum Analyzer computes and plots the spectrum over the interval specified by the Span and CenterFrequency properties.
- When you set this property to `'Start and stop frequencies'`, the Spectrum Analyzer computes and plots the spectrum over the interval specified by the StartFrequency and StopFrequency properties.

This property is Tunable (Simulink).

Default: `'Full'`

`Name`

Caption to display on Spectrum Analyzer window

Specify as a character vector the caption to display on the scope window. This property is Tunable (Simulink).

Default: `'Spectrum Analyzer'`

`NumInputPorts`

Number of input ports

Number of input ports, specified as an integer. Each signal coming through an input port becomes a separate channel in the Spectrum Analyzer.

Default: `1`

`OpenAtSimulationStart`

Open scope when starting simulation

Set this property to `true` to open the scope when the simulation starts. Set this property to `false` to prevent the scope from opening at the start of simulation.

Default: `true`

`OverlapPercent`

Overlap percentage

Specify as a real, scalar value, the percentage overlap between the previous and current buffered data segments. The overlap creates a window segment that is used to compute a spectral estimate. The value must be greater than or equal to zero and less than 100. This property is Tunable (Simulink).

Default: `0`

`PlotAsTwoSidedSpectrum`

Two sided spectrum flag

Set this property to `true` to compute and plot two-sided spectral estimates. Set this property to `false` to compute and plot one-sided spectral estimates. If you set this property to `false`, then the input signal must be real valued. When the input signal is complex valued, you must set this property to `true`.

When this property is `false`, Spectrum Analyzer uses *power folding*. The *y*-axis values are twice the amplitude that they would be if this property were set to `true`, except at 0 and the Nyquist frequency. A one-sided PSD contains the total power of the signal in the frequency interval from DC to half of the Nyquist rate. For more information, see the `pwelch` function reference page.

Default: `true`

`PlotMaxHoldTrace`

Max-hold trace flag

Set this property to `true` to compute and plot the maximum-hold spectrum of each input channel. The maximum-hold spectrum at each frequency bin is computed by keeping the maximum value of all the power spectrum estimates. When you toggle this property, the Spectrum Analyzer resets its maximum-hold computations. This property applies only when you set the ViewType property to `'Spectrum'` or `'Spectrum and spectrogram'`.

This property is Tunable (Simulink).

See also: PlotMinHoldTrace and PlotNormalTrace.

Default: `false`

`PlotMinHoldTrace`

Min-hold trace flag

Set this property to `true` to compute and plot the minimum-hold spectrum of each input channel. The minimum-hold spectrum at each frequency bin is computed by keeping the minimum value of all the power spectrum estimates. When you toggle this property, the Spectrum Analyzer resets its minimum-hold computations. This property applies only when you set the ViewType property to `'Spectrum'` or `'Spectrum and spectrogram'`.

This property is Tunable (Simulink).

See also: PlotMaxHoldTrace andPlotNormalTrace.

Default: `false`

`PlotNormalTrace`

Normal trace flag

Set this property to `false` to remove the display of the normal traces. These traces display the free-running spectral estimates. Note that even when the traces are removed from the display, the Spectrum Analyzer continues its spectral computations. This property applies only when you set the ViewType property to `'Spectrum'` or `'Spectrum and spectrogram'`.

This property is Tunable (Simulink).

See also: PlotMaxHoldTrace and PlotMinHoldTrace.

Default: `true`

`PlotType`

Plot type for normal traces

Specify the type of plot to use for displaying normal traces as either `'Line'` or `'Stem'`. *Normal* traces are traces that display free-running spectral estimates. This property applies only when you set the ViewType property to `'Spectrum'` or `'Spectrum and spectrogram'` and the PlotNormalTrace property to `true`.

This property is Tunable (Simulink).

Default: `'Line'`

Position

Spectrum Analyzer window position in pixels

Specify, in pixels, the size and location of the scope window as a 4-element double vector of the form [left bottom width height]. You can place the scope window in a specific position on your screen by modifying the values to this property. This property is Tunable (Simulink).

Default: The default depends on your screen resolution. By default, the Spectrum Analyzer window appears in the center of your screen with a width of `800` pixels and height of `450` pixels.

RBW

Resolution bandwidth

Specify as a real, positive scalar the resolution bandwidth (RBW), in hertz. RBW controls the spectral resolution of Spectrum Analyzer. This property applies only when you set the FrequencyResolutionMethod property to `'RBW'` and the RBWSource property to `'Property'`. You must specify a value to ensure that there are at least two RBW intervals over the specified frequency span. Thus, the ratio of the overall span to RBW must be greater than two: $\frac{span}{RBW} > 2$. You can specify the overall span in different ways based on how you set the `FrequencySpan` on page 4-1573 property.

This property is Tunable (Simulink).

See also: RBWSource.

Default: `9.76`

RBWSource

Source of resolution bandwidth value

Specify the source of the resolution bandwidth (RBW) as either `'Auto'` or `'Property'`. This property is relevant only when you set the FrequencyResolutionMethod property to `'RBW'`.

- When you set this property to `'Auto'`, the Spectrum Analyzer adjusts the spectral estimation resolution to ensures that there are 1024 RBW intervals over the defined frequency span.
- When you set this property to `'Property'`, specify the resolution bandwidth directly using the RBW property.

This property is Tunable (Simulink).

See also: RBW.

Default: `'Auto'`

`ReducePlotRate`

Reduce plot rate to improve performance

When you set this property to `true`, the scope logs data for later use and updates the display at fixed intervals of time. Data occurring between these fixed intervals might not be plotted. When you set this property to `false`, the scope updates every time it computes the power spectrum. Use the `false` setting when you do not want to miss any spectral updates at the expense of slower simulation speed. The simulation speed is faster when this property is set to `true`. This property is Tunable (Simulink).

Default: `true`

`ReferenceLoad`

Reference load

Specify as a real, positive scalar the load, in ohms, that the Spectrum Analyzer uses as a reference to compute power values.

This property is Tunable (Simulink).

Default: `1`

`SampleRate`

Sample rate of input

Specify the sample rate, in hertz, of the input signals.

The sample rate must be a finite numeric scalar.

Default: `10e3`

`ShowGrid`

Option to enable or disable grid display

When you set this property to `true`, the grid appears. When you set this property to `false`, the grid is hidden. This property is Tunable (Simulink).

Default: `false`

`ShowLegend`

Show or hide legend

When you set this property to `true`, the scope displays a legend with the input channels labels specified in the `ChannelNames` on page 4-1570 property. When you set this property to `false`, the scope does not display a legend.

This property applies only when you set the `ViewType` property to `'Spectrum'` or `'Spectrum and spectrogram'`. This property is Tunable (Simulink).

Default: `false`

`SidelobeAttenuation`

Sidelobe attenuation of window

Specify as a real, positive scalar the window sidelobe attenuation, in decibels (dB). This property applies only when you set the Window property to `'Chebyshev'` or `'Kaiser'`. The value must be greater than or equal to `45`.

This property is Tunable (Simulink).

Default: `60`

`Span`

Frequency span over which spectrum is computed and plotted

Specify as a real, positive scalar the frequency span, in hertz, over which the Spectrum Analyzer computes and plots the spectrum. This property applies only when you set the FrequencySpan property to `'Span and center frequency'`. The overall span, defined by this property and the CenterFrequency property, must fall within the *Nyquist* frequency interval. When the PlotAsTwoSidedSpectrum property is `true`, the Nyquist interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. When you set the `PlotAsTwoSidedSpectrum` property to `false`, the Nyquist interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz.

This property is Tunable (Simulink).

Default: `10e3`

SpectralAverages

Number of spectral averages

Specify as a positive, scalar integer the number of spectral averages. This property applies only when you set the ViewType property to `'Spectrum'`. The Spectrum Analyzer computes the current power spectrum estimate by computing a running average of the last N power spectrum estimates. This property defines the number of spectral averages, N.

This property is Tunable (Simulink).

Default: `1`

SpectralMask

Spectral mask lines for a Power or Power Density plot.

Specify whether to display upper and lower spectral masks lines on a spectrum plot. This property uses `SpectralMaskSpecfication` properties to enable and configure the spectral masks. The `SpectralMaskSpecfication` properties are:

- `EnabledMasks` — Masks to enable, specified as a character vector. Valid values are `'None'`, `'Upper'`, `'Lower'`, or `'Upper and lower'`.

  Default: `'None'`
- `UpperMask` — Upper limit spectral mask, specified as a scalar or two-column matrix. If `UpperMask` is a scalar, the upper limit mask uses the power value of the scalar

for all frequency values applicable to the Spectrum Analyzer. If UpperMask is a matrix, the first column contains the frequency values (Hz), which correspond to the *x*-axis values. The second column contains the power values, which correspond to the associated *y*-axis values. To apply offsets to the power and frequency values, use ReferenceLevel and MaskFrequencyOffset property values, respectively.

Default: Inf

- LowerMask — Lower limit spectral mask, specified as a scalar or two-column matrix. If LowerMask is a scalar, the lower limit mask uses the power value of the scalar for all frequency values applicable to the Spectrum Analyzer. If LowerMask is a matrix, the first column contains the frequency values (Hz), which correspond to the *x*-axis values. The second column contains the power values, which correspond to the associated *y*-axis values. To apply offsets to the power and frequency values, use ReferenceLevel and MaskFrequencyOffset property values, respectively.

  Default: Inf

- ReferenceLevel — Reference level for mask power values, specified as either 'Custom' or 'Spectrum peak'. When ReferenceLevel is 'Custom', the CustomReferenceLevel property value is used as the reference to the power values, in dBr, in the UpperMask and LowerMask properties. When ReferenceLevel is 'Spectrum peak', the peak value of the current spectrum of the SelectedChannel is used.

  Default: 'Custom'

- CustomReferenceLevel — Custom reference level, specified as a real value, in the same units as the power units. The reference level is the value to which to reference the power values in the UpperMask and LowerMask properties. This property applies when ReferenceLevel is set to 'Custom'. This property uses the same units as the PowerUnits property of the Spectrum Analyzer.

  Default: 0

- SelectedChannel — Input channel with peak spectrum to use as the mask reference level, specified as an integer. This property applies when ReferenceLevel is set to 'Spectrum peak'.

  Default: 1

- MaskFrequencyOffset — Frequency offset, specified as a finite, numeric scalar. Frequency offset is the amount of offset to apply to frequency values in the UpperMask and LowerMask properties.

Default: `0`

All `SpectralMaskSpecfication` properties are Tunable (Simulink).

You can check the status of the spectral mask using a method or an event listener.

- To get the status of the spectral masks, call getSpectralMaskStatus.
- To perform an action every time the mask fails, use the `MaskTestFailed` event. To trigger a function when the mask fails, create a listener to the `MaskTestFailed` event and define a callback function to trigger. For more details about using events, see “Events” (MATLAB).

For example spectral masks, refer to the dsp.SpectrumAnalzyer System object or `Spectrum Analyzer` block.

SpectrumType

Type of spectrum to show

Specify the spectrum type as one of `'Power'`, `'Power density'`, or `'RMS'`.

- `'Power'`, the Spectrum Analyzer shows the power spectrum.
- `'Power density'`, the Spectrum Analyzer shows the power spectral density. The power spectral density is the magnitude squared of the spectrum normalized to a bandwidth of 1 hertz.
- `'RMS'`, the Spectrum Analyzer shows the root mean squared. The root mean squared shows the square root of the sum of the squared means. This option is useful when viewing the frequency of voltage or current signals.

This property is Tunable (Simulink).

Default: `'Power'`

SpectrumUnits

Units of the spectrum

Specify the units in which the Spectrum Analyzer displays power values as either `'dBm'`, `'dBW'`, `'Watts'`, `'Vrms'`, or `'dBV'`.

The available spectrum units depends on the value of `SpectrumType`.

`SpectrumType`	Allowed `SpectrumUnits`
`Power` or `Power density`	`'dBm'`, `'dBW'`, `'Watts'`
`RMS`	`'Vrms'`, `'dBV'`

This property is Tunable (Simulink).

Default: `'dBm'`

`StartFrequency`

Start frequency over which spectrum is computed

Specify as a real scalar the start frequency, in hertz, over which the Spectrum Analyzer computes and plots the spectrum. This property applies only when you set the FrequencySpan property to `'Start and stop frequencies'`. The overall span, which is defined by StopFrequency and this property, must fall within the *Nyquist* frequency interval. When the PlotAsTwoSidedSpectrum property is `true`, the Nyquist interval is $\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. If you set the `PlotAsTwoSidedSpectrum` property to `false`, the Nyquist interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz.

This property is Tunable (Simulink).

Default: `-5e3`

`StopFrequency`

Stop frequency over which spectrum is computed

Specify as a real scalar the stop frequency, in hertz, over which the Spectrum Analyzer computes and plots the spectrum. This property applies only when you set the FrequencySpan property to `'Start and stop frequencies'`. The overall span, defined by this property and the StartFrequency property, must fall within the *Nyquist* frequency interval. When the PlotAsTwoSidedSpectrum property is set to `true`, the interval is $\left[-\frac{SampleRate}{2},\frac{SampleRate}{2}\right]+FrequencyOffset$ hertz. When the `PlotAsTwoSidedSpectrum` property is set to `false`, the interval is $\left[0,\frac{SampleRate}{2}\right]+FrequencyOffset$ hertz.

This property is Tunable (Simulink).

Default: `5e3`

`TimeResolution`

Time resolution

Specify the time resolution of each spectrogram line as a positive scalar, expressed in seconds. This property applies when you set the ViewType property to `'Spectrogram'` or `'Spectrum and spectrogram'`, and the TimeResolutionSource property to `'Property'`.

The time resolution value is determined based on frequency resolution method, the RBW setting, and the time resolution setting.

Frequency Resolution	RBW Setting	Time Resolution Setting	Time Resolution
RBW (Hz)	`Auto`	`Auto`	1/RBW s
RBW (Hz)	`Auto`	Manually entered	Time Resolution s
RBW (Hz)	Manually entered	`Auto`	1/RBW s
RBW (Hz)	Manually entered	Manually entered	Must be equal to or greater than the minimum attainable time resolution, 1/RBW s. Several spectral estimates are combined into one spectrogram line to obtain the desired time resolution. Interpolation is used to obtain time resolution values that are not integer multiples of 1/RBW s.

Frequency Resolution	RBW Setting	Time Resolution Setting	Time Resolution
Window length	—	`Auto`	1/RBW s RBW = *(NENBW**Fs)/Window Length, where *NENBW* is the normalized effective noise bandwidth of the specified window.
Window length	—	Manually entered	Must be equal to or greater than the minimum attainable time resolution, *(NENBW**Fs)/ Window Length. Several spectral estimates are combined into one spectrogram line to obtain the desired time resolution. Interpolation is used to obtain time resolution values that are not integer multiples of 1/RBW s.

This property is Tunable (Simulink).

Default: `0.001`

`TimeResolutionSource`

Source of the time resolution value.

Specify the source for the time resolution of each spectrogram line as either `'Auto'` or `'Property'`. This property applies when you set the ViewType property to `'Spectrogram'` or `'Spectrum and spectrogram'`. See the time resolution table at TimeResolution.

This property is Tunable (Simulink).

Default: `'Auto'`

`TimeSpan`

Time span

Specify the time span of the spectrogram display as a positive scalar in seconds. This property applies when you set the ViewType property to `'Spectrogram'` or `'Spectrum` and spectrogram'and the TimeSpanSource property to 'Property'. You must set the time span to be at least twice as large as the duration of the number of samples required for a spectral update.

This property is Tunable (Simulink).

Default: 0.1

TimeSpanSource

Source of time span value

Specify the source for the time span of the spectrogram as either 'Auto' or 'Property'. This property applies when you set the ViewType property to 'Spectrogram' or 'Spectrum and spectrogram'. If you set this property to 'Auto', the spectrogram displays 100 spectrogram lines at any given time. If you set this property to 'Property', the spectrogram uses the time duration you specify in seconds in the TimeSpan property.

This property is Tunable (Simulink).

Default: 'Auto'

Title

Display title

Specify the display title as a character vector. Enter %<SignalLabel> to use the signal labels in the Simulink Model as the axes titles. This property is Tunable (Simulink).

Default: ''

TreatMby1SignalsAsOneChannel

Treat unoriented sample-based input signal as a column vector

Set this property to true to treat M-by-1 and unoriented sample-based inputs as a column vector, or one channel. Set this property to false to treat M-by-1 and unoriented sample-based inputs as a 1-by-M row vector.

Default: true

ViewType

Viewer type

Specify the spectrum type as one of 'Spectrum', 'Spectrogram', or 'Spectrum and spectrogram'.

- 'Spectrum' — the Spectrum Analyzer shows the spectrum.
- 'Spectrogram' — the Spectrum Analyzer opens a spectrogram view, which shows frequency content over time. Each line of the spectrogram is one periodogram. Time scrolls from the bottom to the top of the display. The most recent spectrogram update is at the bottom of the display.
- 'Spectrum and Spectrogram' — the Spectrum Analyzer shows a dual view of a spectrum and spectrogram.

This property is Tunable (Simulink).

Default: 'Spectrum'

Window

Window function

Specify a window function for the spectral estimator as one of the options in the following table. For information on any window function, follow the link to the corresponding function reference in the Signal Processing Toolbox documentation.

Window Option	Corresponding Function in Signal Processing Toolbox
'Rectangular'	rectwin
'Chebyshev'	chebwin
'Flat Top'	flattopwin
'Hamming'	hamming
'Hann'	hann
'Kaiser'	kaiser

This property is Tunable (Simulink).

Default: 'Hann'

WindowLength

The window length

Control the frequency resolution by specifying the window length, in samples used to compute the spectral estimates. The window length must be an integer scalar greater than 2. This property applies only when you set the FrequencyResolutionMethod property to `'WindowLength'`, which controls the frequency resolution based on your window length setting. The resulting resolution bandwidth (RBW) is $\frac{NENBW * Fs}{N_{window}}$ where N_{window} is the `WindowLength` value.

This property is Tunable (Simulink).

See also: Window.

Default: `1024`

YLabel

The label for the *y*-axis

Specify the text for the scope to display to the left of the *y*-axis. Tunable (Simulink)

This property applies only when you set the ViewType property to `'Spectrum'` or `'Spectrum and spectrogram'`. Regardless of this property, Spectrum Analyzer always displays power units as one of `'dBm'`, `'dBW'`, `'Watts'`, `'dBm/Hz'`, `'dBW/Hz'`, `'Watts/Hz'`, `'Vrms'`, `'dBV'`.

See also: SpectrumUnits.

Default: `''`

YLimits

The limits for the *y*-axis

Specify the *y*-axis limits as a 2-element numeric vector, `[ymin ymax]`. This property is Tunable (Simulink).

This property applies only when you set the ViewType property to `'Spectrum'` or `'Spectrum and spectrogram'`. The units directly depend upon the SpectrumUnits property.

Default: `[-80, 20]`

Examples

Construct a Spectrum Analyzer Configuration Object

Create a new Simulink® model with a randomly-generated name.

```
sysname=char(randi(26,1,7)+96);
new_system(sysname);
```

Add a new Spectrum Analyzer block to the model.

```
add_block('built-in/SpectrumAnalyzer',[sysname,'/SpectrumAnalyzer'])
```

Call the `get_param (Simulink)` function to retrieve the default Spectrum Analyzer block configuration properties.

```
hsac = get_param([sysname,'/SpectrumAnalyzer'],'ScopeConfiguration')
```

```
hsac =

  SpectrumAnalyzerConfiguration with properties:

                          Name: 'SpectrumAnalyzer'
                 NumInputPorts: '1'
         OpenAtSimulationStart: 1
                       Visible: 0
                      Position: [240 287 800 450]
              SampleRateSource: 'Inherited'
                    SampleRate: '10e3'
                  SpectrumType: 'Power'
                      ViewType: 'Spectrum'
                 FrequencySpan: 'Full'
                          Span: '10e3'
               CenterFrequency: '0'
                StartFrequency: '-5e3'
                 StopFrequency: '5e3'
     FrequencyResolutionMethod: 'RBW'
                     RBWSource: 'Auto'
                           RBW: '9.76'
                  WindowLength: '1024'
```

```
            FFTLengthSource: 'Auto'
                  FFTLength: '1024'
             OverlapPercent: '0'
                     Window: 'Hann'
       SidelobeAttenuation: '60'
      TimeResolutionSource: 'Auto'
             TimeResolution: '1e-3'
             TimeSpanSource: 'Auto'
                   TimeSpan: '100e-3'
           SpectralAverages: '1'
              SpectrumUnits: 'dBm'
              ReferenceLoad: '1'
               SpectralMask: [1×1 SpectralMaskSpecification]
           PlotMaxHoldTrace: 0
           PlotMinHoldTrace: 0
            PlotNormalTrace: 1
                   PlotType: 'Line'
     PlotAsTwoSidedSpectrum: 1
             FrequencyScale: 'Linear'
            FrequencyOffset: '0'
                      Title: ''
                    YLimits: [-80 20]
                     YLabel: ''
                 ShowLegend: 0
               ChannelNames: {''}
                   ShowGrid: 1
             ReducePlotRate: 1
TreatMby1SignalsAsOneChannel: 1
                 AxesLayout: 'Vertical'
```

See Also

See Also
dsp.SpectrumAnalyzer | `Spectrum Analyzer`

Topics
“Control Scopes Programmatically” (Simulink)

Functions — Alphabetical List

adaptfilt

Adaptive filter

Compatibility

`adaptfilt` has been removed. Use dsp.LMSFilter, dsp.BlockLMSFilter, dsp.RLSFilter, dsp.FastTransversalFilter, dsp.AffineProjectionFilter, dsp.AdaptiveLatticeFilter, dsp.FilteredXLMSFilter, dsp.FrequencyDomainAdaptiveFilter, or dsp.AffineProjectionFilter instead.

Syntax

```
ha = adaptfilt.algorithm('input1',input2,...)
```

Description

`ha = adaptfilt.`*`algorithm`*`('input1',input2,...)` returns the adaptive filter object `ha` that uses the adaptive filtering technique specified by *`algorithm`*. When you construct an adaptive filter object, include an *`algorithm`* specifier to implement a specific adaptive filter. Note that you do not enclose the algorithm option in single quotation marks. To construct an adaptive filter object you must supply an *`algorithm`* name — there is no default algorithm, although every constructor creates a default adaptive filter when you do not provide input arguments such as `input1` or `input2` in the calling syntax.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Algorithms

For adaptive filter (`adaptfilt`) objects, the *`algorithm`* name determines which adaptive filter algorithm your `adaptfilt` object implements. Each available algorithm entry appears in one of the tables along with a brief description of the algorithm. Click on the algorithm in the first column to get more information about the associated adaptive filter technique.

- "Least Mean Squares (LMS) Based FIR Adaptive Filters" on page 5-3
- "Recursive Least Squares (RLS) Based FIR Adaptive Filters" on page 5-3
- "Affine Projection (AP) FIR Adaptive Filters" on page 5-4
- "FIR Adaptive Filters in the Frequency Domain (FD)" on page 5-4
- "Lattice Based (L) FIR Adaptive Filters" on page 5-5

Least Mean Squares (LMS) Based FIR Adaptive Filters

adaptfilt.algorithm	Algorithm Used to Generate Filter Coefficients
`adaptfilt.adjlms`	Use the Adjoint LMS FIR adaptive filter algorithm
`adaptfilt.blms`	Use the Block LMS FIR adaptive filter algorithm
`adaptfilt.blmsfft`	Use the FFT-based Block LMS FIR adaptive filter algorithm
`adaptfilt.dlms`	Use the delayed LMS FIR adaptive filter algorithm
`adaptfilt.filtxlms`	Use the filtered-x LMS FIR adaptive filter algorithm
`adaptfilt.lms`	Use the LMS FIR adaptive filter algorithm
`adaptfilt.nlms`	Use the normalized LMS FIR adaptive filter algorithm
`adaptfilt.sd`	Use the sign-data LMS FIR adaptive filter algorithm
`adaptfilt.se`	Use the sign-error LMS FIR adaptive filter algorithm
`adaptfilt.ss`	Use the sign-sign LMS FIR adaptive filter algorithm

For further information about an adapting algorithm, refer to the reference page for the algorithm.

Recursive Least Squares (RLS) Based FIR Adaptive Filters

adaptfilt.algorithm	Algorithm Used to Generate Filter Coefficients
`adaptfilt.ftf`	Use the fast transversal least squares adaptation algorithm
`adaptfilt.qrdrls`	Use the QR-decomposition RLS adaptation algorithm
`adaptfilt.hrls`	Use the householder RLS adaptation algorithm

adaptfilt.algorithm	Algorithm Used to Generate Filter Coefficients
`adaptfilt.hswrls`	Use the householder SWRLS adaptation algorithm
`adaptfilt.rls`	Use the recursive-least squares (RLS) adaptation algorithm
`adaptfilt.swrls`	Use the sliding window (SW) RLS adaptation algorithm
`adaptfilt.swftf`	Use the sliding window FTF adaptation algorithm

For more complete information about an adapting algorithm, refer to the reference page for the algorithm.

Affine Projection (AP) FIR Adaptive Filters

adaptfilt.algorithm	Algorithm Used to Generate Filter Coefficients
`adaptfilt.ap`	Use the affine projection algorithm that uses direct matrix inversion
`adaptfilt.apru`	Use the affine projection algorithm that uses recursive matrix updating
`adaptfilt.bap`	Use the block affine projection adaptation algorithm

To find more information about an adapting algorithm, refer to the reference page for the algorithm.

FIR Adaptive Filters in the Frequency Domain (FD)

adaptfilt.algorithm	Algorithm Used to Generate Filter Coefficients
`adaptfilt.fdaf`	Use the frequency domain adaptation algorithm
`adaptfilt.pbfdaf`	Use the partition block version of the FDAF algorithm
`adaptfilt.pbufdaf`	Use the partition block unconstrained version of the FDAF algorithm
`adaptfilt.tdafdct`	Use the transform domain adaptation algorithm using DCT
`adaptfilt.tdafdft`	Use the transform domain adaptation algorithm using DFT
`adaptfilt.ufdaf`	Use the unconstrained FDAF algorithm for adaptation

For more information about an adapting algorithm, refer to the reference page for the algorithm.

Lattice Based (L) FIR Adaptive Filters

adaptfilt.algorithm	Algorithm Used to Generate Filter Coefficients
`adaptfilt.gal`	Use the gradient adaptive lattice filter adaptation algorithm
`adaptfilt.lsl`	Use the least squares lattice adaptation algorithm
`adaptfilt.qrdlsl`	Use the QR decomposition least squares lattice adaptation algorithm

For more information about an adapting algorithm, refer to the reference page for the algorithm.

Properties for All Adaptive Filter Objects

Each reference page for an algorithm and `adaptfilt.algorithm` object specifies which properties apply to the adapting algorithm and how to use them.

Methods for Adaptive Filter Objects

As is true with all objects, methods enable you to perform various operations on `adaptfilt` objects. To use the methods, you apply them to the object handle that you assigned when you constructed the `adaptfilt` object.

Most of the analysis methods that apply to `dfilt` objects also work with `adaptfilt` objects. Methods like `freqz` rely on the filter coefficients in the `adaptfilt` object. Since the coefficients change each time the filter adapts to data, you should view the results of using a method as an analysis of the filter at a moment in time for the object. Use caution when you apply an analysis method to your adaptive filter objects — always check that your result approached your expectation.

In particular, the Filter Visualization Tool (FVTool) supports all of the `adaptfilt` objects. Analyzing and viewing your `adaptfilt` objects is straightforward — use the `fvtool` method with the name of your object

```
fvtool(objectname)
```

to launch FVTool and work with your object.

Some methods share their names with functions in Signal Processing Toolbox software, or even functions in this toolbox. Functions that share names with methods behave in a similar way. Using the same name for more than one function or method is called *overloading* and is common in many toolboxes.

Method	Description
`adaptfilt/coefficients`	Return the instantaneous adaptive filter coefficients
`adaptfilt/filter`	Apply an `adaptfilt` object to your signal
`adaptfilt/freqz`	Plot the instantaneous adaptive filter frequency response
`adaptfilt/grpdelay`	Plot the instantaneous adaptive filter group delay
`adaptfilt/impz`	Plot the instantaneous adaptive filter impulse response.
`adaptfilt/info`	Return the adaptive filter information.
`adaptfilt/isfir`	Test whether an adaptive filter is an finite impulse response (FIR) filters.
`adaptfilt/islinphase`	Test whether an adaptive filter is linear phase
`adaptfilt/ismaxphase`	Test whether an adaptive filter is maximum phase
`adaptfilt/isminphase`	Test whether an adaptive filter is minimum phase
`adaptfilt/isreal`	True whether an adaptive filter has real coefficients
`adaptfilt/isstable`	Test whether an adaptive filter is stable
`adaptfilt/maxstep`	Return the maximum step size for an adaptive filter
`adaptfilt/msepred`	Return the predicted mean square error
`adaptfilt/msesim`	Return the measured mean square error via simulation.
`adaptfilt/phasez`	Plot the instantaneous adaptive filter phase response
`adaptfilt/reset`	Reset an adaptive filter to initial conditions

Method	Description
`adaptfilt/stepz`	Plot the instantaneous adaptive filter step response
`adaptfilt/tf`	Return the instantaneous adaptive filter transfer function
`adaptfilt/zerophase`	Plot the instantaneous adaptive filter zerophase response
`adaptfilt/zpk`	Return a matrix containing the instantaneous adaptive filter zero, pole, and gain values
`adaptfilt/zplane`	Plot the instantaneous adaptive filter in the Z-plane

Working with Adaptive Filter Objects

The next sections cover viewing and changing the properties of `adaptfilt` objects.

Viewing Object Properties

As with any object, you can use `get` to view a `adaptfilt` object's properties. To see a specific property, use

```
get(ha,'property')
```

To see all properties for an object, use

```
get(ha)
```

Changing Object Properties

To set specific properties, use

```
set(ha,'property1',value1,'property2',value2,...)
```

You must use single quotation marks around the property name so MATLAB treats them as character vectors.

Copying an Object

To create a copy of an object, use `copy`.

```
ha2 = copy(ha)
```

Note Using the syntax `ha2 = ha` copies only the object handle and does not create a new object — `ha` and `ha2` are not independent. When you change the characteristics of `ha2`, those of `ha` change as well.

Using Filter States

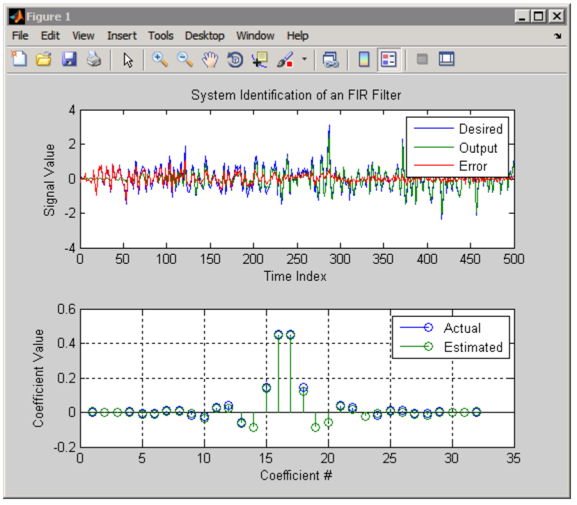
Two properties control your adaptive filter states.

- `States` — stores the current states of the filter. Before the filter is applied, the states correspond to the initial conditions and after the filter is applied, the states correspond to the final conditions.
- `PersistentMemory` — resets the filter before filtering. The default value is `false` which causes the properties that are modified by the filter, such as `coefficients` and `states`, to be reset to the value you specified when you constructed the object, before you use the object to filter data. Setting `PersistentMemory` to `true` allows the object to retain its current properties between filtering operations, rather than resetting the filter to its property values at construction.

Examples

Construct an LMS adaptive filter object and use it to identify an unknown system. For this example, use 500 iteration of the adapting process to determine the unknown filter coefficients. Using the LMS algorithm represents one of the most straightforward technique for adaptive filters.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
mu = 0.008;            % LMS step size.
ha = adaptfilt.lms(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');  grid on;
```



See Also

`dfilt` | `filter`

Introduced in R2011a

adaptfilt.adjlms

FIR adaptive filter that uses adjoint LMS algorithm

Compatibility

`adaptfilt.adjlms` has been removed. Use dsp.LMSFilter, dsp.BlockLMSFilter, dsp.RLSFilter, dsp.FastTransversalFilter, dsp.AffineProjectionFilter, dsp.AdaptiveLatticeFilter, dsp.FilteredXLMSFilter, dsp.FrequencyDomainAdaptiveFilter, or dsp.AffineProjectionFilter instead.

Syntax

```
ha = adaptfilt.adjlms(l,step,leakage,pathcoeffs,pathest,...
errstates,pstates,coeffs,states)
```

Note: `adaptfilt.adjlms` will be removed in a future release.

Description

`ha = adaptfilt.adjlms(l,step,leakage,pathcoeffs,pathest,... errstates,pstates,coeffs,states)` constructs object `ha`, an FIR adjoint LMS adaptive filter. `l` is the adaptive filter length (the number of coefficients or taps) and must be a positive integer. `l` defaults to 10 when you omit the argument. `step` is the adjoint LMS step size. It must be a nonnegative scalar. When you omit the `step` argument, `step` defaults to 0.1.

`leakage` is the adjoint LMS leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, you implement a leaky version of the `adjlms` algorithm to determine the filter coefficients. `leakage` defaults to 1 specifying no leakage in the algorithm.

`pathcoeffs` is the secondary path filter model. This vector should contain the coefficient values of the secondary path from the output actuator to the error sensor.

pathest is the estimate of the secondary path filter model. pathest defaults to the values in pathcoeffs.

errstates is a vector of error states of the adaptive filter. It must have a length equal to the filter order of the secondary path model estimate. errstates defaults to a vector of zeros of appropriate length. pstates contains the secondary path FIR filter states. It must be a vector of length equal to the filter order of the secondary path model. pstates defaults to a vector of zeros of appropriate length. The initial filter coefficients for the secondary path filter compose vector coeffs. It must be a length l vector. coeffs defaults to a length l vector of zeros. states is a vector containing the initial filter states. It must be a vector of length l+ne-1, where ne is the length of errstates. When you omit states, it defaults to an appropriate length vector of zeros.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for filter.

Properties

In the syntax for creating the adaptfilt object, the input options are properties of the object created. This table lists the properties for the adjoint LMS object, their default values, and a brief description of the property.

Property	Default Value	Description
Algorithm	None	Specifies the adaptive filter algorithm the object uses during adaptation
Coefficients	Length l vector with zeros for all elements	Adjoint LMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need l entries in coefficients. Updated filter coefficients are returned in coefficients when you use s as an output argument.
ErrorStates	[0,...,0]	A vector of the error states for your adaptive filter, with length equal to the order of your secondary path filter.
FilterLength	10	The number of coefficients in your adaptive filter.

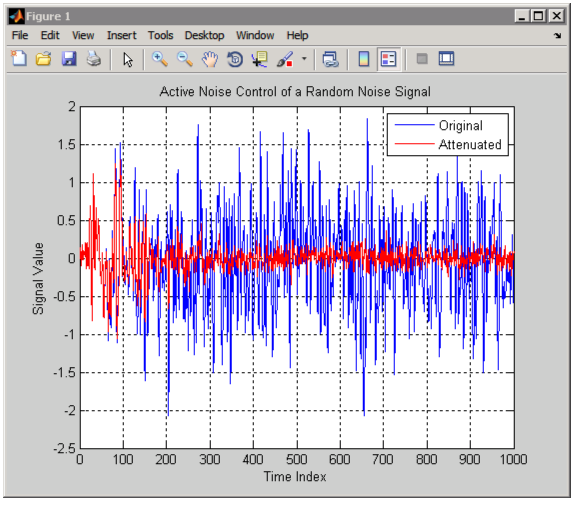
Property	Default Value	Description
`Leakage`	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1.
`SecondaryPathCoeffs`	No default	A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor.
`SecondaryPathEstimate`	`pathcoeffs` values	An estimate of the secondary path filter model.
`SecondaryPathStates`	Length of the secondary path filter. All elements are zeros.	The states of the secondary path filter, the unknown system
`States`	l+ne+1, where ne is `length(errstates)`	Contains the initial conditions for your adaptive filter and returns the states of the FIR filter after adaptation. If omitted, it defaults to a zero vector of length equal to l +ne+1. When you use `adaptfilt.adjlms` in a loop structure, use this element to specify the initial filter states for the adapting FIR filter.
`Stepsize`	0.1	Sets the adjoint LMS algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.

Examples

Demonstrate active noise control of a random noise signal that runs for 1000 samples.

```
x  = randn(1,1000);      % Noise source
g  = fir1(47,0.4);       % FIR primary path system model
n  = 0.1*randn(1,1000);  % Observation noise signal
d  = filter(g,1,x)+n;    % Signal to be canceled (desired)
b  = fir1(31,0.5);       % FIR secondary path system model
mu = 0.008;              % Adjoint LMS step size
ha = adaptfilt.adjlms(32,mu,1,b);
[y,e] = filter(ha,x,d);
plot(1:1000,d,'b',1:1000,e,'r');
title('Active Noise Control of a Random Noise Signal');
legend('Original','Attenuated');
xlabel('Time Index'); ylabel('Signal Value');  grid on;
```

Reviewing the figure shows that the adaptive filter attenuates the original noise signal as you expect.



References

Wan, Eric., "Adjoint LMS: An Alternative to Filtered-X LMS and Multiple Error LMS," Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP), pp. 1841-1845, 1997

See Also

`adaptfilt.dlms` | `adaptfilt.filtxlms`

Introduced in R2011a

adaptfilt.ap

FIR adaptive filter that uses direct matrix inversion

Compatibility

`adaptfilt.ap` has been removed. Use dsp.AffineProjectionFilter instead.

Syntax

```
ha = adaptfilt.ap(l,step,projectord,offset,coeffs,states,...
errstates,epsstates)
```

Description

`ha = adaptfilt.ap(l,step,projectord,offset,coeffs,states,...
errstates,epsstates)` constructs an affine projection FIR adaptive filter `ha` using direct matrix inversion.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.ap`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	Affine projection step size. This is a scalar that should be a value between zero and one. Setting `step` equal to one provides the fastest convergence during adaptation. `step` defaults to 1.

Input Argument	Description
projectord	Projection order of the affine projection algorithm. projectord defines the size of the input signal covariance matrix and defaults to two.
offset	Offset for the input signal covariance matrix. You should initialize the covariance matrix to a diagonal matrix whose diagonal entries are equal to the offset you specify. offset should be positive. offset defaults to one.
coeffs	Vector containing the initial filter coefficients. It must be a length l vector, the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
states	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).
errstates	Vector of the adaptive filter error states. errstates defaults to a zero vector with length equal to (projectord - 1).
epsstates	Vector of the epsilon values of the adaptive filter. epsstates defaults to a vector of zeros with (projectord - 1) elements.

Properties

Since your adaptfilt.ap filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.ap objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
ProjectionOrder	1 to as large as needed.	Projection order of the affine projection algorithm. ProjectionOrder defines the size of the input signal covariance matrix and defaults to two.
OffsetCov	Matrix of values	Contains the offset covariance matrix

Name	Range	Description
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector, the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).
`ErrorStates`	Vector of elements	Vector of the adaptive filter error states. `errstates` defaults to a zero vector with length equal to (`projectord` - 1).
`EpsilonStates`	Vector of elements	Vector of the epsilon values of the adaptive filter. `epsstates` defaults to a vector of zeros with (`projectord` - 1) elements.
`StepSize`	Any scalar from zero to one, inclusive	Specifies the step size taken between filter coefficient updates
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `true`.

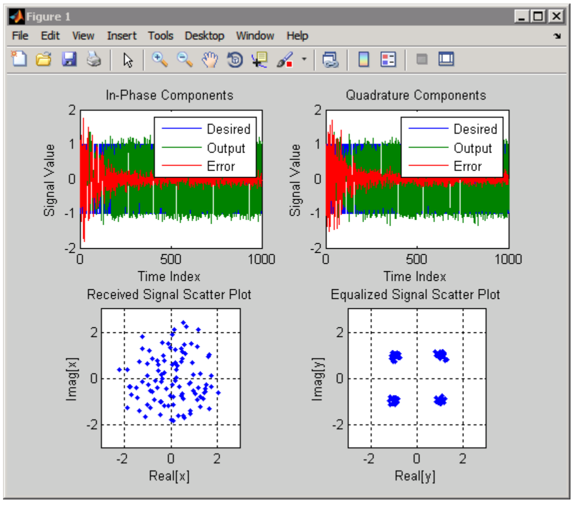
Examples

Quadrature phase shift keying (QPSK) adaptive equalization using a 32-coefficient FIR filter. Run the adaptation for 1000 iterations.

```
D  = 16;                      % Number of samples of delay
b  = exp(j*pi/4)*[-0.7 1];    % Numerator coefficients of channel
a  = [1 -0.7];                % Denominator coefficients of channel
ntr= 1000;                    % Number of iterations
s  = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D));% Baseband Signal
n  = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D));      % Noise signal
r  = filter(b,a,s)+n;         % Received signal
```

```
x  = r(1+D:ntr+D);     % Input signal (received signal)
d  = s(1:ntr);         % Desired signal (delayed QPSK signal)
mu = 0.1;              % Step size
po = 4;                % Projection order
offset = 0.05;         % Offset for covariance matrix
ha = adaptfilt.ap(32,mu,po,offset);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e])); title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.');
axis([-3 3 -3 3]); title('Received Signal Scatter Plot');
axis('square');  xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot');
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

The four plots shown reveal the QPSK process at work.



References

[1] Ozeki, K. and Umeda, T., “An Adaptive Filtering Algorithm Using an Orthogonal Projection to an Affine Subspace and Its Properties,” Electronics and Communications in Japan, vol.67-A, no. 5, pp. 19-27, May 1984

[2] Maruyama, Y., "A Fast Method of Projection Algorithm," Proc. 1990 IEICE Spring Conf., B-744

See Also

`msesim`

Introduced in R2011a

adaptfilt.apru

FIR adaptive filter that uses recursive matrix updating

Compatibility

`adaptfilt.apru` has been removed. Use dsp.AffineProjectionFilter instead.

Syntax

```
ha = adaptfilt.apru(l,step,projectord,offset,coeffs,states,
...errstates,epsstates)
```

Description

`ha = adaptfilt.apru(l,step,projectord,offset,coeffs,states,`
`...errstates,epsstates)` constructs an affine projection FIR adaptive filter `ha` using recursive matrix updating.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.apru`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps). It must be a positive integer. `l` defaults to 10.
`step`	Affine projection step size. This is a scalar that should be a value between zero and one. Setting `step` equal to one provides the fastest convergence during adaptation. `step` defaults to 1.
`projectord`	Projection order of the affine projection algorithm. `projectord` defines the size of the input signal covariance matrix and defaults to two.

Input Argument	Description
`offset`	Offset for the input signal covariance matrix. You should initialize the covariance matrix to a diagonal matrix whose diagonal entries are equal to the offset you specify. `offset` should be positive. `offset` defaults to one.
`coeffs`	Vector containing the initial filter coefficients. It must be a length `l` vector, the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`states`	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).
`errstates`	Vector of the adaptive filter error states. `errstates` defaults to a zero vector with length equal to (`projectord` - 1).
`epsstates`	Vector of the epsilon values of the adaptive filter. `epsstates` defaults to a vector of zeros with (`projectord` - 1) elements.

Properties

Since your `adaptfilt.apru` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.apru` objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`ProjectionOrder`	1 to as large as needed.	Projection order of the affine projection algorithm. `ProjectionOrder` defines the size of the input signal covariance matrix and defaults to two.
`OffsetCov`	Matrix of values	Contains the offset covariance matrix
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector,

Name	Range	Description
		the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).
`ErrorStates`	Vector of elements	Vector of the adaptive filter error states. `errstates` defaults to a zero vector with length equal to (`projectord` - 1).
`EpsilonStates`	Vector of elements	Vector of the epsilon values of the adaptive filter. `epsstates` defaults to a vector of zeros with (`projectord` - 1) elements.
`StepSize`	Any scalar from zero to one, inclusive	Specifies the step size taken between filter coefficient updates
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `true`.

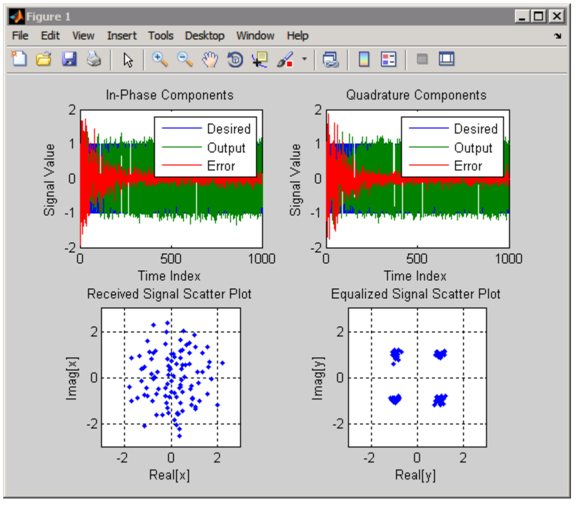
Examples

Demonstrate quadrature phase shift keying (QPSK) adaptive equalization using a 32-coefficient FIR filter. This example runs the adaptation process for 1000 iterations.

```
D  = 16;                        % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                  % Denominator coefficients of channel
ntr= 1000;                      % Number of iterations
```

```
s  = sign(randn(1,ntr+D)) + 1j*sign(randn(1,ntr+D)); % Baseband
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));  % Noise signal
r  = filter(b,a,s)+n;     % Received signal
x  = r(1+D:ntr+D); % Input signal (received signal)
d  = s(1:ntr);     % Desired signal (delayed QPSK signal)
mu = 0.1;                       % Step size
po = 4;                         % Projection order
offset = 0.05;                     % Offset
ha = adaptfilt.apru(32,mu,po,offset); [y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e])); title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e])); title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot');
axis('square');  xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot');
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

In the following component and scatter plots, you see the results of QPSK equalization.



References

[1] Ozeki. K., T. Omeda, “An Adaptive Filtering Algorithm Using an Orthogonal Projection to an Affine Subspace and Its Properties,” Electronics and Communications in Japan, vol. 67-A, no. 5, pp. 19-27, May 1984

[2] Maruyama, Y, "A Fast Method of Projection Algorithm," Proceedings 1990 IEICE Spring Conference, B-744

See Also

`adaptfilt` | `adaptfilt.ap` | `adaptfilt.bap`

Introduced in R2011a

adaptfilt.bap

FIR adaptive filter that uses block affine projection

Compatibility

`adaptfilt.bap` has been removed. Use dsp.AffineProjectionFilter instead.

Syntax

```
ha = adaptfilt.bap(l,step,projectord,offset,coeffs,states)
```

Description

`ha = adaptfilt.bap(l,step,projectord,offset,coeffs,states)` constructs a block affine projection FIR adaptive filter `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.bap`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	Affine projection step size. This is a scalar that should be a value between zero and one. Setting `step` equal to one provides the fastest convergence during adaptation. `step` defaults to 1.
`projectord`	Projection order of the affine projection algorithm. `projectord` defines the size of the input signal covariance matrix and defaults to two.

Input Argument	Description
offset	Offset for the input signal covariance matrix. You should initialize the covariance matrix to a diagonal matrix whose diagonal entries are equal to the offset you specify. offset should be positive. offset defaults to one.
coeffs	Vector containing the initial filter coefficients. It must be a length l vector, the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
states	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).

Properties

Since your adaptfilt.bap filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.bap objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
ProjectionOrder	1 to as large as needed.	Projection order of the affine projection algorithm. ProjectionOrder defines the size of the input signal covariance matrix and defaults to two.
OffsetCov	Matrix of values	Contains the offset covariance matrix
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector, the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.

Name	Range	Description
`States`	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).
`StepSize`	Any scalar from zero to one, inclusive	Specifies the step size taken between filter coefficient updates
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `true`.

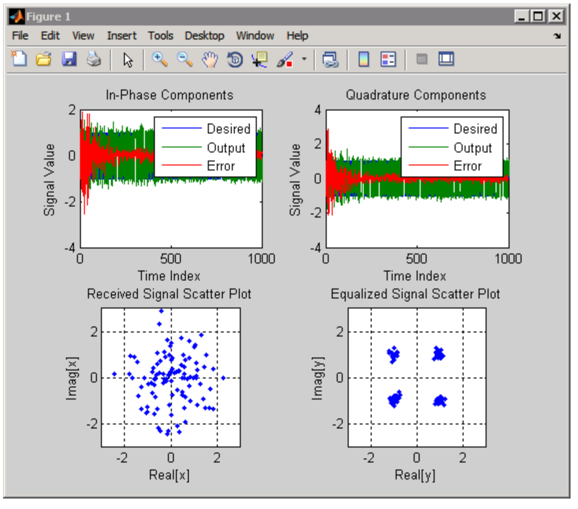
Examples

Show an example of quadrature phase shift keying (QPSK) adaptive equalization using a 32-coefficient FIR filter.

```
D  = 16;                              % delay
b  = exp(1j*pi/4)*[-0.7 1];           % Numerator coefficients
a  = [1 -0.7];                       % Denominator coefficients
ntr= 1000;                           % Number of iterations
s  = sign(randn(1,ntr+D))+1j*sign(randn(1,ntr+D));% Baseband signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));   % Noise signal
r  = filter(b,a,s)+n;                          % Received signal
x  = r(1+D:ntr+D);                    % Input signal (received signal)
d  = s(1:ntr);                   % Desired signal (delayed QPSK signal)
mu = 0.5;                        % Step size
po = 4;                          % Projection order
offset = 1.0;                    % Offset for covariance matrix
ha = adaptfilt.bap(32,mu,po,offset);
[y,e] = filter(ha,x,d); subplot(2,2,1);
plot(1:ntr,real([d;y;e]));
title('In-Phase Components'); legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
```

```
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

Using the block affine projection object in QPSK results in the plots shown here.



References

[1] Ozeki, K. and T. Omeda, "An Adaptive Filtering Algorithm Using an Orthogonal Projection to an Affine Subspace and Its Properties," Electronics and Communications in Japan, vol. 67-A, no. 5, pp. 19-27, May 1984

[2] Montazeri, M. and Duhamel, P, "A Set of Algorithms Linking NLMS and Block RLS Algorithms," IEEE Transactions Signal Processing, vol. 43, no. 2, pp, 444-453, February 1995

See Also

`adaptfilt` | `adaptfilt.ap` | `adaptfilt.apru`

Introduced in R2011a

adaptfilt.blms

FIR adaptive filter that uses BLMS

Compatibility

`adaptfilt.blms` has been removed. Usedsp.BlockLMSFilter instead.

Syntax

```
ha = adaptfilt.blms(l,step,leakage,blocklen,coeffs,states)
```

Description

`ha = adaptfilt.blms(l,step,leakage,blocklen,coeffs,states)` constructs an FIR block LMS adaptive filter `ha`, where `l` is the adaptive filter length (the number of coefficients or taps) and must be a positive integer. `l` defaults to 10.

`step` is the block LMS step size. You must set step to a nonnegative scalar. You can use function `maxstep` to determine a reasonable range of step size values for the signals being processed. When unspecified, `step` defaults to 0.

`leakage` is the block LMS leakage factor. It must be a scalar between 0 and 1. If you set leakage to be less than one, you implement the leaky block LMS algorithm. `leakage` defaults to 1 specifying no leakage in the adapting algorithm.

`blocklen` is the block length used. It must be a positive integer and the signal vectors `d` and `x` should be divisible by `blocklen`. Larger block lengths result in faster per-sample execution times but with poor adaptation characteristics. When you choose `blocklen` such that `blocklen + length(coeffs)` is a power of 2, use `adaptfilt.blmsfft`. `blocklen` defaults to `l`.

`coeffs` is a vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length `l` vector of zeros.

`states` contains a vector of your initial filter states. It must be a length `l` vector and defaults to a length `l` vector of zeros when you do not include it in your calling function.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object created. This table lists the properties for the adjoint LMS object, their default values, and a brief description of the property.

Property	Default Value	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`States`	Vector of elements	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to `l`
`Leakage`		Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1.
`BlockLength`	Vector of length `l`	Size of the blocks of data processed in each iteration
`StepSize`	0.1	Sets the block LMS algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely

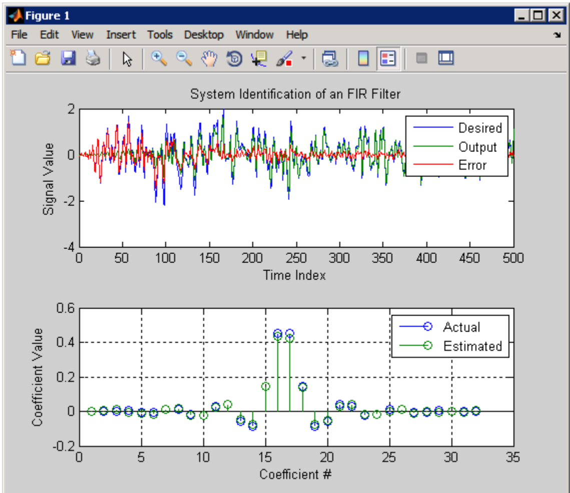
Property	Default Value	Description
		the adaptive filter converges to the filter solution. Use `maxstep` to determine the maximum usable step size.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.

Examples

Use an adaptive filter to identify an unknown 32nd-order FIR filter. In this example 500 input samples result in 500 iterations of the adaptation process. You see in the plot that follows the example code that the adaptive filter has determined the coefficients of the unknown system under test.

```
x  = randn(1,500);        % Input to the filter
b  = fir1(31,0.5);        % FIR system to be identified
no  = 0.1*randn(1,500);   % Observation noise signal
d  = filter(b,1,x)+no;    % Desired signal
mu = 0.008;               % Block LMS step size
n  = 5;                   % Block length
ha = adaptfilt.blms(32,mu,1,n);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');
grid on;
```

Based on looking at the figures here, the adaptive filter correctly identified the unknown system after 500 iterations, or fewer. In the lower plot, you see the comparison between the actual filter coefficients and those determined by the adaptation process.



References

Shynk, J.J.,“Frequency-Domain and Multirate Adaptive Filtering,” IEEE Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992.

See Also

`adaptfilt.blmsfft` | `adaptfilt.fdaf` | `adaptfilt.lms`

Introduced in R2011a

adaptfilt.blmsfft

FIR adaptive filter that uses FFT-based BLMS

Compatibility

`adaptfilt.blmsfft` has been removed. Use dsp.LMSFilter, dsp.BlockLMSFilter, dsp.RLSFilter, dsp.FastTransversalFilter, dsp.AffineProjectionFilter, dsp.AdaptiveLatticeFilter, dsp.FilteredXLMSFilter, dsp.FrequencyDomainAdaptiveFilter, or dsp.AffineProjectionFilter instead.

Syntax

```
ha = adaptfilt.blmsfft(l,step,leakage,blocklen,coeffs,
states)
```

Description

`ha = adaptfilt.blmsfft(l,step,leakage,blocklen,coeffs, states)` constructs an FIR block LMS adaptive filter object `ha` where `l` is the adaptive filter length (the number of coefficients or taps) and must be a positive integer. `l` defaults to 10. `step` is the block LMS step size. It must be a nonnegative scalar. The function `maxstep` may be helpful to determine a reasonable range of step size values for the signals you are processing. `step` defaults to 0.

`leakage` is the block LMS leakage factor. It must also be a scalar between 0 and 1. When `leakage` is less than one, the `adaptfilt.blmsfft` implements a leaky block LMS algorithm. `leakage` defaults to 1 (no leakage). `blocklen` is the block length used. It must be a positive integer such that

```
blocklen + length(coeffs)
```

is a power of two; otherwise, an `adaptfilt.blms` algorithm is used for adapting. Larger block lengths result in faster execution times, with poor adaptation characteristics as the cost of the speed gained. `blocklen` defaults to `l`. Enter your initial filter coefficients in `coeffs`, a vector of length `l`. When omitted, `coeffs` defaults to a length `l` vector of all zeros. `states` contains a vector of initial filter states; it must be a length `l` vector. `states` defaults to a length `l` vector of all zeros when you omit the `states` argument in the calling syntax.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object you create. This table lists the properties for the block LMS object, their default values, and a brief description of the property.

Property	Default Value	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coefficients` defaults to length `l` vector of zeros when you do not provide the argument for input.
`States`	Vector of elements of length `l`	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to `l`
`Leakage`	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1.
`BlockLength`	Vector of length `l`	Size of the blocks of data processed in each iteration
`StepSize`	0.1	Sets the block LMS algorithm step size used for each iteration of the adapting algorithm.

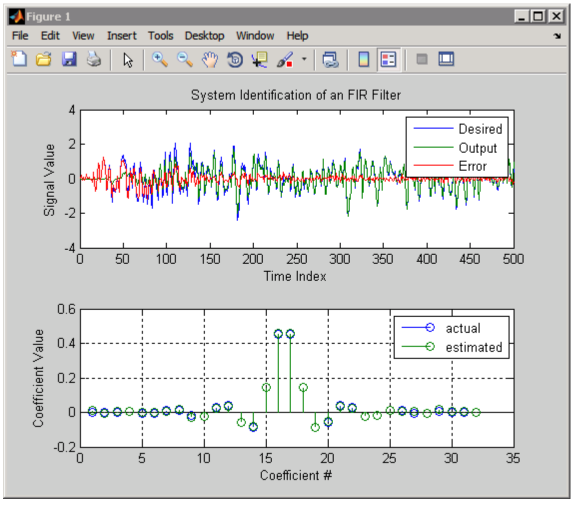
Property	Default Value	Description
		Determines both how quickly and how closely the adaptive filter converges to the filter solution. Use `maxstep` to determine the maximum usable step size.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.

Examples

Identify an unknown FIR filter with 32 coefficients using 512 iterations of the adapting algorithm.

```
x  = randn(1,512);       % Input to the filter
b  = fir1(31,0.5);       % FIR system to be identified
no  = 0.1*randn(1,512);  % Observation noise signal
d  = filter(b,1,x)+no;   % Desired signal
mu = 0.008;              % Step size
n  = 16;                 % Block length
ha = adaptfilt.blmsfft(32,mu,1,n);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d(1:500);y(1:500);e(1:500)]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error'); xlabel('Time Index');
ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('actual','estimated'); grid on;
xlabel('Coefficient #'); ylabel('Coefficient Value');
```

As a result of running the adaptation process, filter object `ha` now matches the unknown system FIR filter `b`, based on comparing the filter coefficients derived during adaptation.



References

Shynk, J.J., "Frequency-Domain and Multirate Adaptive Filtering," IEEE Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992.

See Also

`adaptfilt.blms` | `adaptfilt.fdaf` | `adaptfilt.lms` | `filter`

Introduced in R2011a

adaptfilt.dlms

FIR adaptive filter that uses delayed LMS

Compatibility

`adaptfilt.dlms` has been removed. Use dsp.LMSFilter, dsp.BlockLMSFilter, dsp.RLSFilter, dsp.FastTransversalFilter, dsp.AffineProjectionFilter, dsp.AdaptiveLatticeFilter, dsp.FilteredXLMSFilter, dsp.FrequencyDomainAdaptiveFilter, or dsp.AffineProjectionFilter instead.

Syntax

```
ha = adaptfilt.dlms(l,step,leakage,delay,errstates,coeffs,
...states)
```

Description

`ha = adaptfilt.dlms(l,step,leakage,delay,errstates,coeffs, ...states)` constructs an FIR delayed LMS adaptive filter `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.dlms`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	LMS step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.
`leakage`	Your LMS leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, `adaptfilt.lms` implements a

Input Argument	Description
	leaky LMS algorithm. When you omit the `leakage` property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.
`delay`	Update delay given in time samples. This scalar should be a positive integer — negative delays do not work. `delay` defaults to 1.
`errstates`	Vector of the error states of your adaptive filter. It must have a length equal to the update delay (`delay`) in samples. `errstates` defaults to an appropriate length vector of zeros.
`coeffs`	Vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length `l` vector with elements equal to zero.
`states`	Vector of initial filter states for the adaptive filter. It must be a length `l`-1 vector. `states` defaults to a length l-1 vector of zeros.

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object you create. This table lists the properties for the block LMS object, their default values, and a brief description of the property.

Property	Default Value	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input. LMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need `l` entries in `coeffs`.
`Delay`	1	Specifies the update delay for the adaptive algorithm.

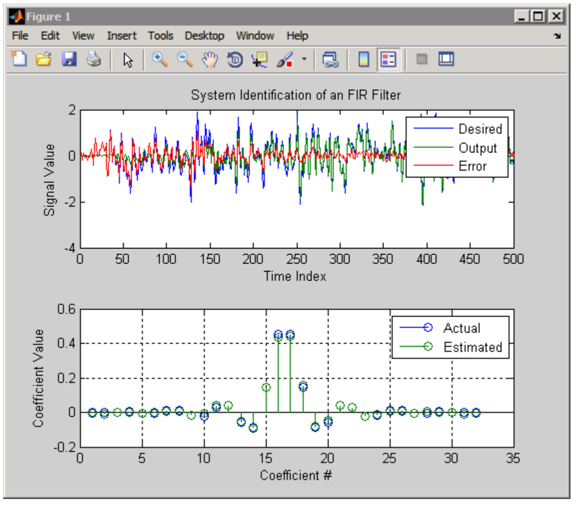
Property	Default Value	Description
ErrorStates	Vector of zeros with the number of elements equal to delay	A vector comprising the error states for the adaptive filter.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
Leakage	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
StepSize	0.1	Sets the LMS algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).

Examples

System identification of a 32-coefficient FIR filter. Refer to the figure that follows to see the results of the adapting filter process.

```
x  = randn(1,500);      % Input to the filter
b  = fir1(31,0.5);      % FIR system to be identified
n  = 0.1*randn(1,500);  % Observation noise signal
d  = filter(b,1,x)+n;   % Desired signal
mu = 0.008;             % LMS step size.
delay = 1;              % Update delay
ha = adaptfilt.dlms(32,mu,1,delay);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('Actual','Estimated'); grid on;
xlabel('Coefficient #'); ylabel('Coefficient Value');
```

Using a delayed LMS adaptive filter in the process to identify an unknown filter appears to work as planned, as shown in this figure.



References

Shynk, J.J.,“Frequency-Domain and Multirate Adaptive Filtering,” IEEE Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992.

See Also

`adaptfilt.adjlms` | `adaptfilt.filtxlms` | `adaptfilt.lms`

Introduced in R2011a

adaptfilt.fdaf

FIR adaptive filter that uses frequency-domain with bin step size normalization

Compatibility

`adaptfilt.fdaf` has been removed. Use dsp.FrequencyDomainAdaptiveFilter instead.

Syntax

```
ha = adaptfilt.fdaf(l,step,leakage,delta,lambda,blocklen,
offset,...coeffs,states)
```

Description

`ha = adaptfilt.fdaf(l,step,leakage,delta,lambda,blocklen, offset,...coeffs,states)` constructs a frequency-domain FIR adaptive filter `ha` with bin step size normalization. If you omit all the input arguments you create a default object with l = 10 and `step` = 1.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.fdaf`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps). l must be a positive integer; it defaults to 10 when you omit the argument.
`step`	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. `step` defaults to 1.
`leakage`	Leakage parameter of the adaptive filter. If this parameter is set to a value between zero and one, you implement a leaky FDAF algorithm. `leakage` defaults to 1 — no leakage provided in the algorithm.

Input Argument	Description
delta	Initial common value of all of the FFT input signal powers. Its initial value should be positive. delta defaults to 1.
lambda	Specifies the averaging factor used to compute the exponentially-windowed FFT input signal powers for the coefficient updates. lambda should lie in the range (0,1]. lambda defaults to 0.9.
blocklen	Block length for the coefficient updates. This must be a positive integer. For faster execution, (blocklen + l) should be a power of two. blocklen defaults to l.
offset	Offset for the normalization terms in the coefficient updates. Use this to avoid divide by zeros or by very small numbers when any of the FFT input signal powers become very small. offset defaults to zero.
coeffs	Initial time-domain coefficients of the adaptive filter. coeff should be a length l vector. The adaptive filter object uses these coefficients to compute the initial frequency-domain filter coefficients via an FFT computed after zero-padding the time-domain vector by the blocklen.
states	The adaptive filter states. states defaults to a zero vector that has length equal to l.

Properties

Since your adaptfilt.fdaf filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.fdaf objects. To show you the properties that apply, this table lists and describes each property for the adaptfilt.fdaf filter object.

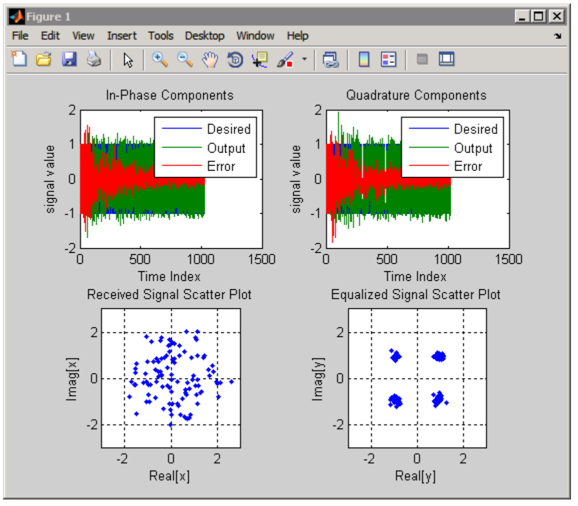
Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation.
AvgFactor	(0, 1]	Specifies the averaging factor used to compute the exponentially-windowed FFT input signal powers for the coefficient updates. Same as the input argument lambda.

Name	Range	Description
BlockLength	Any integer	Block length for the coefficient updates. This must be a positive integer. For faster execution, (blocklen + l) should be a power of two. blocklen defaults to l.
FFTCoefficients		Stores the discrete Fourier transform of the filter coefficients in coeffs.
FFTStates		States for the FFT operation.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
Leakage		Leakage parameter of the adaptive filter. if this parameter is set to a value between zero and one, you implement a leaky FDAF algorithm. leakage defaults to 1 — no leakage provided in the algorithm.
Offset	Any positive real value	Offset for the normalization terms in the coefficient updates. Use this to avoid dividing by zero or by very small numbers when any of the FFT input signal powers become very small. offset defaults to zero.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
Power		A vector of 2*l elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.
StepSize	Any scalar from zero to one, inclusive	Specifies the step size taken between filter coefficient updates

Examples

Quadrature Phase Shift Keying (QPSK) adaptive equalization using 1024 iterations of a 32-coefficient FIR filter. After this example code, a figure demonstrates the equalization results.

```
D = 16;                          % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                   % Denominator coefficients of channel
ntr= 1024;                       % Number of iterations
s  = sign(randn(1,ntr+D))+1j*sign(randn(1,ntr+D));  %QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));       % Noise signal
r  = filter(b,a,s)+n;            % Received signal
x  = r(1+D:ntr+D);               % Input signal (received signal)
d  = s(1:ntr);                   % Desired signal (delayed QPSK signal)
del = 1;                         % Initial FFT input powers
mu  = 0.1;                       % Step size
lam = 0.9;                       % Averaging factor
ha = adaptfilt.fdaf(32,mu,1,del,lam);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e])); title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('signal value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e])); title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('signal value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```



References

Shynk, J.J.,"Frequency-Domain and Multirate Adaptive Filtering," IEEE Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992

See Also

`adaptfilt.ufdaf` | `adaptfilt.pbfdaf` | `adaptfilt.blms` | `adaptfilt.blmsfft`

Introduced in R2011a

adaptfilt.filtxlms

FIR adaptive filter that uses filtered-x LMS

Compatibility

`adaptfilt.filtxlms` has been removed. Use dsp.FilteredXLMSFilter instead.

Syntax

```
ha = adaptfilt.filtxlms(l,step,leakage,pathcoeffs,
pathest,...errstates,pstates,coeffs,states)
```

Description

`ha = adaptfilt.filtxlms(l,step,leakage,pathcoeffs,
pathest,...errstates,pstates,coeffs,states)` constructs an filtered-x LMS adaptive filter `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.filtxlms`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	Filtered LMS step size. it must be a nonnegative scalar. `step` defaults to 0.1.
`leakage`	is the filtered-x LMS leakage factor. it must be a scalar between 0 and 1. If it is less than one, a leaky version of `adaptfilt.filtxlms` is implemented. `leakage` defaults to 1 (no leakage).

Input Argument	Description
`pathcoeffs`	is the secondary path filter model. this vector should contain the coefficient values of the secondary path from the output actuator to the error sensor.
`pathest`	is the estimate of the secondary path filter model. `pathest` defaults to the values in `pathcoeffs`.
`fstates`	is a vector of filtered input states of the adaptive filter. `fstates` defaults to a zero vector of length equal to (`l` - 1).
`pstates`	are the secondary path FIR filter states. it must be a vector of length equal to the (`length`(`pathcoeffs`) - 1). `pstates` defaults to a vector of zeros of appropriate length.
`coeffs`	is a vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length l vector of zeros.
`states`	Vector of initial filter states. `states` defaults to a zero vector of length equal to the larger of (`length`(`pathcoeffs`) - 1) and (`length`(`pathest`) - 1).

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object created. This table lists the properties for the adjoint LMS object, their default values, and a brief description of the property.

Property	Default Value	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.

Property	Default Value	Description
`FilteredInputStates`	`l-1`	Vector of filtered input states with length equal to `l` - 1.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`States`	Vector of elements	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2)
`SecondaryPathCoeffs`	No default	A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor
`SecondaryPathEstimate`	`pathcoeffs` values	An estimate of the secondary path filter model
`SecondaryPathStates`	Vector of size (`length` (`pathcoeffs`) -1) with all elements equal to zero.	The states of the secondary path FIR filter — the unknown system
`StepSize`	0.1	Sets the filtered-x algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.

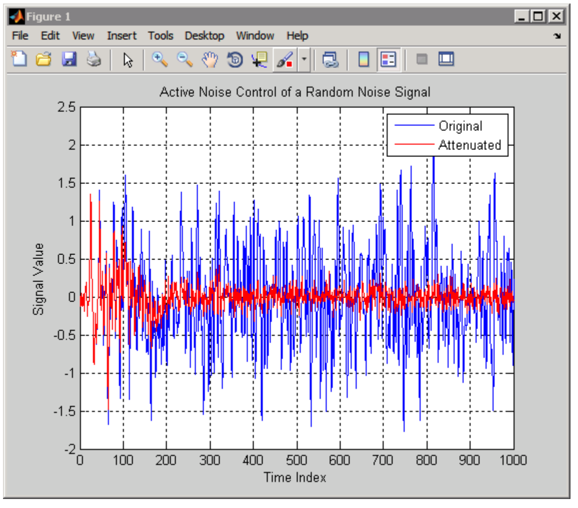
Examples

Demonstrate active noise control of a random noise signal over 1000 iterations.

As the figure that follows this code demonstrates, the filtered-x LMS filter successfully controls random noise in this context.

```
x  = randn(1,1000);      % Noise source
g  = fir1(47,0.4);       % FIR primary path system model
n  = 0.1*randn(1,1000);  % Observation noise signal
```

```
d  = filter(g,1,x)+n;     % Signal to be cancelled
b  = fir1(31,0.5);        % FIR secondary path system model
mu = 0.008;               % Filtered-X LMS step size
ha = adaptfilt.filtxlms(32,mu,1,b);
[y,e] = filter(ha,x,d); plot(1:1000,d,'b',1:1000,e,'r');
title('Active Noise Control of a Random Noise Signal');
legend('Original','Attenuated');
xlabel('Time Index'); ylabel('Signal Value');  grid on;
```



See also

`adaptfilt.dlms`, `adaptfilt.lms`

References

Kuo, S.M., and Morgan, D.R. *Active Noise Control Systems: Algorithms and DSP Implementations*, New York, N.Y: John Wiley & Sons, 1996.

Widrow, B., and Stearns, S.D. *Adaptive Signal Processing*, Upper Saddle River, N.J: Prentice Hall, 1985.

Introduced in R2011a

adaptfilt.ftf

Fast transversal LMS adaptive filter

Compatibility

`adaptfilt.ftf` has been removed. Use dsp.FastTransversalFilter instead.

Syntax

```
ha = adaptfilt.ftf(l,lambda,delta,gamma,gstates,coeffs,
states)
```

Description

`ha = adaptfilt.ftf(l,lambda,delta,gamma,gstates,coeffs, states)` constructs a fast transversal least squares adaptive filter object `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.ftf`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`lambda`	RLS forgetting factor. This is a scalar that should lie in the range (1-0.5/`l`, 1]. `lambda` defaults to 1.
`delta`	Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. `delta` defaults to one.
`gamma`	Conversion factor. `gamma` defaults to one specifying soft-constrained initialization.

Input Argument	Description
gstates	States of the Kalman gain updates. gstates defaults to a zero vector of length l.
coeffs	Length l vector of initial filter coefficients. coeffs defaults to a length l vector of zeros.
states	Vector of initial filter States. states defaults to a zero vector of length (l-1).

Properties

Since your adaptfilt.ftf filter is an object, it has properties that define its operating behavior. Note that many of the properties are also input arguments for creating adaptfilt.ftf objects. To show you the properties that apply, this table lists and describes each property for the fast transversal least squares filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
BkwdPrediction		Returns the predicted samples generated during adaptation. See “References — Adaptive Filters” for details about linear prediction.
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
ConversionFactor		Conversion factor. Called gamma when it is an input argument, it defaults to the matrix [1 -1] that specifies soft-constrained initialization.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps

Name	Range	Description
`ForgettingFactor`		RLS forgetting factor. This is a scalar that should lie in the range (1-0.5/`l`, 1]. `lambda` defaults to 1.
`FwdPrediction`		Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.
`InitFactor`		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. `delta` defaults to one.
`KalmanGain`		Empty when you construct the object, this gets populated after you run the filter.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).

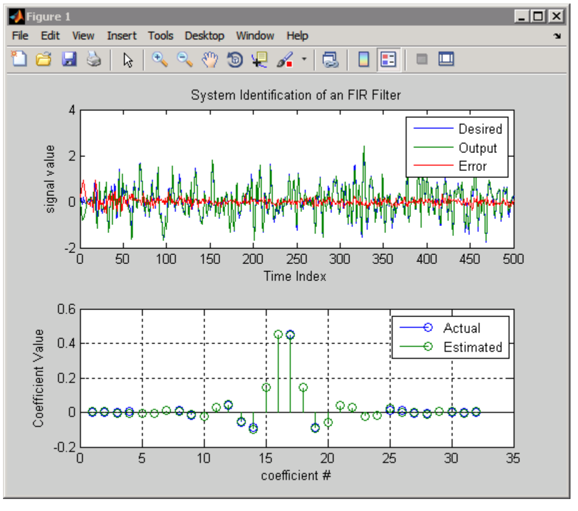
Examples

System Identification of a 32-coefficient FIR filter by running the identification process for 500 iterations.

```
x  = randn(1,500);    % Input to the filter
b  = fir1(31,0.5);    % FIR system to be identified
n  = 0.1*randn(1,500);% Observation noise signal
d  = filter(b,1,x)+n; % Desired signal
```

```
N   = 31;               % Adaptive filter order
lam = 0.99;             % RLS forgetting factor
del = 0.1;              % Soft-constrained initialization factor
ha = adaptfilt.ftf(32,lam,del);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('signal value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated'); grid on;
xlabel('coefficient #'); ylabel('Coefficient Value');
```

For this example of identifying an unknown system, the figure shows that the adaptation process identifies the filter coefficients for the unknown FIR filter within the first 150 iterations.



References

D.T.M. Slock and Kailath, T., “Numerically Stable Fast Transversal Filters for Recursive Least Squares Adaptive Filtering,” IEEE Trans. Signal Processing, vol. 38, no. 1, pp. 92-114.

See Also

`adaptfilt.swftf` | `adaptfilt.rls` | `adaptfilt.lsl`

Introduced in R2011a

adaptfilt.gal

FIR adaptive filter that uses gradient lattice

Compatibility

`adaptfilt.gal` has been removed. Use dsp.AdaptiveLatticeFilter instead.

Syntax

```
ha = adaptfilt.gal(l,step,leakage,offset,rstep,delta,
lambda,...rcoeffs,coeffs,states)
```

Description

`ha = adaptfilt.gal(l,step,leakage,offset,rstep,delta, lambda,...rcoeffs,coeffs,states)` constructs a gradient adaptive lattice FIR filter `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.gal`.

Input Argument	Description
`l`	Length of the joint process filter coefficients. It must be a positive integer and must be equal to the length of the reflection coefficients plus one. `l` defaults to 10.
`step`	Joint process step size of the adaptive filter. This scalar should be a value between zero and one. `step` defaults to 0.
`leakage`	Leakage factor of the adaptive filter. It must be a scalar between 0 and 1. Setting leakage less than one implements a leaky algorithm to estimate both the reflection and the joint process coefficients. `leakage` defaults to 1 (no leakage).

Input Argument	Description
offset	Specifies an optional offset for the denominator of the step size normalization term. It must be a scalar greater or equal to zero. A non-zero offset is useful to avoid divide-by-near-zero conditions when the input signal amplitude becomes very small. offset defaults to 1.
rstep	Reflection process step size of the adaptive filter. This scalar should be a value between zero and one. rstep defaults to step.
delta	Initial common value of the forward and backward prediction error powers. It should be a positive value. 0.1 is the default value for delta.
lambda	Specifies the averaging factor used to compute the exponentially windowed forward and backward prediction error powers for the coefficient updates. lambda should lie in the range (0, 1]. lambda defaults to the value (1 - step).
rcoeffs	Vector of initial reflection coefficients. It should be a length (l-1) vector. rcoeffs defaults to a zero vector of length (l-1).
coeffs	Vector of initial joint process filter coefficients. It must be a length l vector. coeffs defaults to a length l vector of zeros.
states	Vector of the backward prediction error states of the adaptive filter. states defaults to a zero vector of length (l-1).

Properties

Since your adaptfilt.gal filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.gal objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
AvgFactor		Specifies the averaging factor used to compute the exponentially-windowed forward and backward prediction error

Name	Range	Description
		powers for the coefficient updates. Same as the input argument `lambda`.
`BkwdPredErrorPower`		Returns the minimum mean-squared prediction error. See "References — Adaptive Filters" for details about linear prediction.
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`FwdPredErrorPower`		Returns the minimum mean-squared prediction error in the forward direction. See "References — Adaptive Filters" for details about linear prediction.
`Leakage`	0 to 1	Leakage parameter of the adaptive filter. If this parameter is set to a value between zero and one, you implement a leaky GAL algorithm. `leakage` defaults to 1 — no leakage provided in the algorithm.
`Offset`		Offset for the normalization terms in the coefficient updates. Use this to avoid dividing by zero or by very small numbers when input signal amplitude becomes very small. `offset` defaults to one.

Name	Range	Description
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
ReflectionCoeffs		Coefficients determined for the reflection portion of the filter during adaptation.
ReflectionCoeffsStep		Size of the steps used to determine the reflection coefficients.
States	Vector of elements	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).
StepSize	0 to 1	Specifies the step size taken between filter coefficient updates

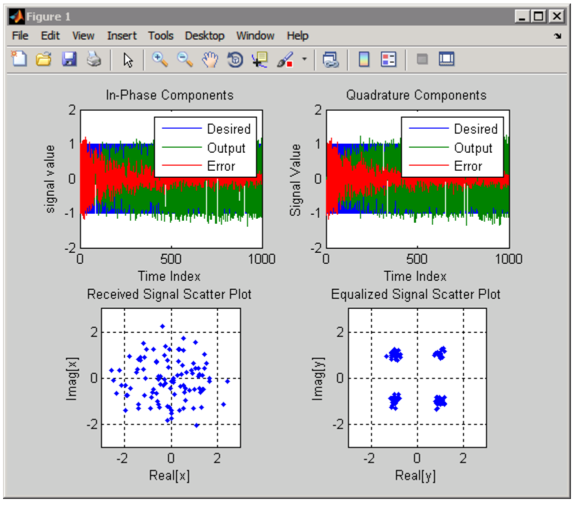
Examples

Perform a Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient adaptive filter over 1000 iterations.

```
D  = 16;                     % Number of delay samples
b  = exp(1j*pi/4)*[-0.7 1];   % Numerator coefficients
a  = [1 -0.7];               % Denominator coefficients
ntr= 1000;                   % Number of iterations
s  = sign(randn(1,ntr+D)) + 1j*sign(randn(1,ntr+D)); % QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));   % Noise signal
r  = filter(b,a,s)+n;        % Received signal
x  = r(1+D:ntr+D);           % Input signal (received signal)
d  = s(1:ntr);               % Desired signal (delayed QPSK signal)
L = 32;                      % filter length
mu = 0.007;                  % Step size
ha = adaptfilt.gal(L,mu);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
```

```
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('signal value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.');  axis([-3 3 -3 3]);
title('Received Signal Scatter Plot');  axis('square');
xlabel('Real[x]'); ylabel('Imag[x]');  grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.');
axis([-3 3 -3 3]); title('Equalized Signal Scatter Plot');
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]');  grid on;
```

To see the results, look at this figure.



References

Griffiths, L.J. “A Continuously Adaptive Filter Implemented as a Lattice Structure,” Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, Hartford, CT, pp. 683-686, 1977

Haykin, S.,*Adaptive Filter Theory*, 3rd Ed., Upper Saddle River, NJ, Prentice Hall, 1996

See Also

`adaptfilt.qrdlsl` | `adaptfilt.lsl` | `adaptfilt.tdafdft`

Introduced in R2011a

adaptfilt.hrls

FIR adaptive filter that uses householder (RLS)

Compatibility

`adaptfilt.hrls` has been removed. Use dsp.RLSFilter instead.

Syntax

```
ha = adaptfilt.hrls(l,lambda,sqrtinvcov,coeffs,states)
```

Description

`ha = adaptfilt.hrls(l,lambda,sqrtinvcov,coeffs,states)` constructs an FIR householder RLS adaptive filter `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.hrls`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`lambda`	RLS forgetting factor. This is a scalar and should lie in the range (0, 1]. `lambda` defaults to 1 meaning the adaptation process retains infinite memory.
`sqrtinvcov`	Square-root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.

Input Argument	Description
`coeffs`	Vector of initial filter coefficients. It must be a length `l` vector. `coeffs` defaults to being a length `l` vector of zeros.
`states`	Vector of initial filter states. It must be a length `l`-1 vector. `states` defaults to a length `l`-1 vector of zeros.

Properties

Since your `adaptfilt.hrls` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.hrls` objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

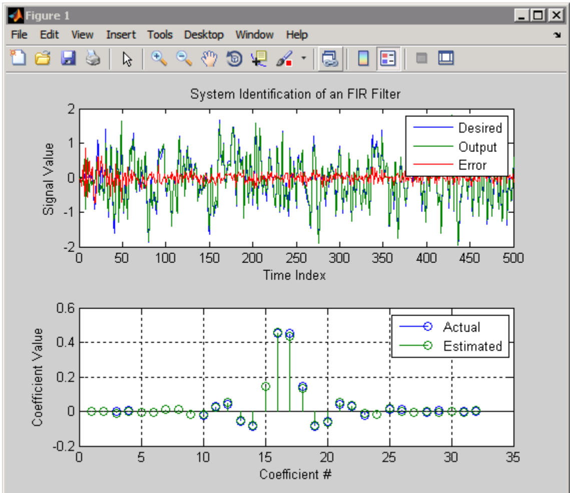
Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`ForgettingFactor`	Scalar	RLS forgetting factor. This is a scalar and should lie in the range (0, 1]. Same as input argument `lambda`. It defaults to 1 meaning the adaptation process retains infinite memory.
`KalmanGain`	Vector of size (`l`,1)	Empty when you construct the object, this gets populated after you run the filter.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when

Name	Range	Description
		you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. Defaults to false.
SqrtInvCov	Matrix of doubles	Square root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l - 1).

Examples

Use 500 iterations of an adaptive filter object to identify a 32-coefficient FIR filter system. Both the example code and the resulting figure show the successful filter identification through adaptive filter processing.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
G0 = sqrt(10)*eye(32); % Initial sqrt correlation matrix inverse
lam = 0.99;            % RLS forgetting factor
ha = adaptfilt.hrls(32,lam,G0);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated'); grid on;
 xlabel('Coefficient #'); ylabel('Coefficient Value');
```



See Also

adaptfilt.hswrls | adaptfilt.qrdrls | adaptfilt.rls

Introduced in R2011a

adaptfilt.hswrls

FIR adaptive filter that uses householder sliding window RLS

Compatibility

`adaptfilt.hswrls` has been removed in a future release. Use dsp.RLSFilter instead.

Syntax

```
ha = adaptfilt.hswrls(l,lambda,sqrtinvcov,swblocklen,
dstates,coeffs,states)
```

Description

`ha = adaptfilt.hswrls(l,lambda,sqrtinvcov,swblocklen, dstates,coeffs,states)` constructs an FIR householder sliding window recursive-least-square adaptive filter `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.hswrls`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`lambda`	Recursive least square (RLS) forgetting factor. This is a scalar and should lie in the range (0, 1]. `lambda` defaults to 1 meaning the adaptation process retains infinite memory.

Input Argument	Description
sqrtinvcov	Square-root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.
swblocklen	Block length of the sliding window. This integer must be at least as large as the filter length. swblocklen defaults to 16.
dstates	Desired signal states of the adaptive filter. dstates defaults to a zero vector with length equal to (swblocklen - 1).
coeffs	Vector of initial filter coefficients. It must be a length l vector. coeffs defaults to being a length l vector of zeros.
states	Vector of initial filter states. It must be a length (l + swblocklen - 2) vector. states defaults to a length (l + swblocklen - 2) vector of zeros.

Properties

Since your adaptfilt.hswrls filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.hswrls objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
DesiredSignalStates	Vector	Desired signal states of the adaptive filter. dstates defaults to a zero vector with length equal to (swblocklen - 1).
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps

Name	Range	Description
ForgettingFactor	Scalar	Root-least-square (RLS) forgetting factor. This is a scalar and should lie in the range (0, 1]. Same as input argument lambda. It defaults to 1 meaning the adaptation process retains infinite memory.
KalmanGain	(l,1) vector	Empty when you construct the object, this gets populated after you run the filter.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. Defaults to false.
SqrtInvCov	l-by-l Matrix	Square-root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).
SwBlockLength	Integer	Block length of the sliding window. This integer must be at least as large as the filter length. swblocklen defaults to 16.

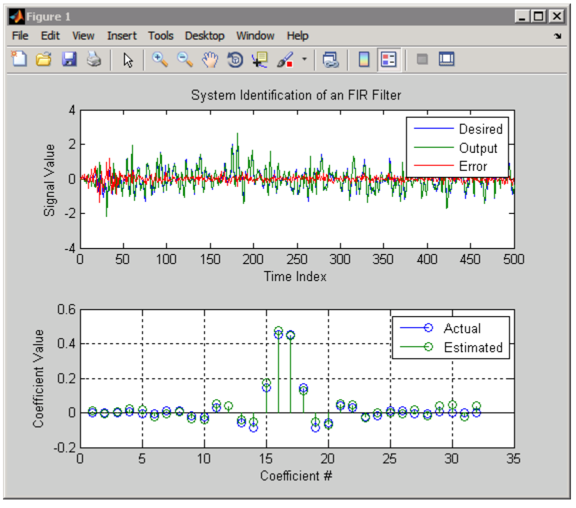
Examples

System Identification of a 32-coefficient FIR filter.

```
x  = randn(1,500);     % Input to the filter
```

```
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
G0 = sqrt(10)*eye(32); % Initial sqrt correlation matrix inverse
lam = 0.99;            % RLS forgetting factor
N  = 64;               % block length
ha = adaptfilt.hswrls(32,lam,G0,N);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated'); grid on;
xlabel('Coefficient #'); ylabel('Coefficient Value');
```

In the pair of plots shown in the figure you see the comparison of the desired and actual output for the adapting filter and the coefficients of both filters, the unknown and the adapted.



See Also

`adaptfilt.hrls` | `adaptfilt.qrdrls` | `adaptfilt.rls`

Introduced in R2011a

adaptfilt.lms

FIR adaptive filter that uses LMS

Compatibility

`adaptfilt.lms` has been removed. Use dsp.LMSFilter instead.

Syntax

```
ha = adaptfilt.lms(l,step,leakage,coeffs,states)
```

Description

`ha = adaptfilt.lms(l,step,leakage,coeffs,states)` constructs an FIR LMS adaptive filter object `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.lms`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	LMS step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.1.
`leakage`	Your LMS leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, `adaptfilt.lms` implements a

Input Argument	Description
	leaky LMS algorithm. When you omit the `leakage` property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.
`coeffs`	Vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length `l` vector with elements equal to zero.
`states`	Vector of initial filter states for the adaptive filter. It must be a length `l`-1 vector. `states` defaults to a length l-1 vector of zeros.

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object created. This table lists the properties for the `adaptfilt.lms` object, their default values, and a brief description of the property.

Property	Range	Property Description
`Algorithm`	None	Reports the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to a length `l` vector of zeros when you do not provide the vector as an input argument.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`Leakage`	0 to 1	LMS leakage factor. It must be a scalar between zero and one. When it is less than one, a leaky NLMS algorithm results. `leakage` defaults to 1 (no leakage).
`PersistentMemory`	`false` or `true`	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values

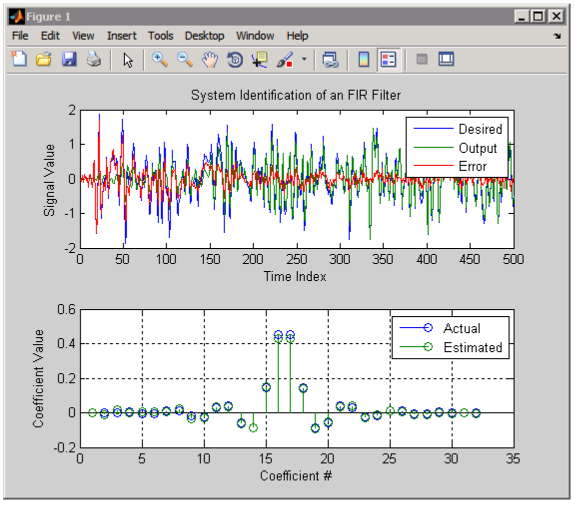
Property	Range	Property Description
		in place when you create the filter. `PersistentMemory` returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to `false`.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` - 1).
`StepSize`	0 to 1	LMS step size. It must be a scalar between zero and one. Setting this step size value to one provides the fastest convergence. `step` defaults to 0.1.

Examples

Use 500 iterations of an adapting filter system to identify and unknown 32nd-order FIR filter.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
mu = 0.008;            % LMS step size.
ha = adaptfilt.lms(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');  grid on;
```

Using LMS filters in an adaptive filter architecture is a time honored means for identifying an unknown filter. By running the example code provided you can demonstrate one process to identify an unknown FIR filter.



References

Shynk J.J., “Frequency-Domain and Multirate Adaptive Filtering,” IEEE Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992.

See Also

`adaptfilt.blms` | `adaptfilt.blmsfft` | `adaptfilt.dlms` | `adaptfilt.nlms` | `adaptfilt.tdafdft` | `adaptfilt.sd` | `adaptfilt.se` | `adaptfilt.ss`

Introduced in R2011a

adaptfilt.lsl

Least squares lattice adaptive filter

Compatibility

`adaptfilt.lsl` has been removed. Use dsp.AdaptiveLatticeFilter instead.

Syntax

```
ha = adaptfilt.lsl(l,lambda,delta,coeffs,states)
```

Description

`ha = adaptfilt.lsl(l,lambda,delta,coeffs,states)` constructs a least squares lattice adaptive filter ha.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.lsl`.

Input Argument	Description
`l`	Length of the joint process filter coefficients. It must be a positive integer and must be equal to the length of the prediction coefficients plus one. L defaults to 10.
`lambda`	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. `lambda` defaults to 1. `lambda` = 1 denotes infinite memory while adapting to find the new filter.
`delta`	Soft-constrained initialization factor in the least squares lattice algorithm. It should be positive. `delta` defaults to 1.
`coeffs`	Vector of initial joint process filter coefficients. It must be a length `l` vector. `coeffs` defaults to a length `l` vector of all zeros.

Input Argument	Description
states	Vector of the backward prediction error states of the adaptive filter. states defaults to a length l vector of all zeros, specifying soft-constrained initialization for the algorithm.

Properties

Since your adaptfilt.lsl filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.lsl objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation.
BkwdPrediction		Returns the predicted samples generated during adaptation. See "References — Adaptive Filters" for details about linear prediction.
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
ForgettingFactor		Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting forgetting factor = 1 denotes infinite memory while adapting to find the new filter. Note that this is the lambda input argument.
FwdPrediction		Contains the predicted values for samples during adaptation. Compare

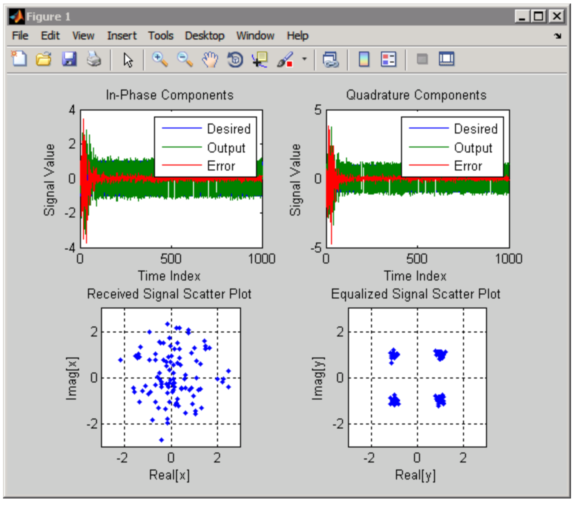
Name	Range	Description
		these to the actual samples to get the error and power.
`InitFactor`		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. `delta` defaults to one.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to `l`.

Examples

Demonstrate Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient adaptive filter running for 1000 iterations. After you review the example code, the figure shows the results of running the example to use QPSK adaptive equalization with a 32nd-order FIR filter. The error between the in-phase and quadrature components, as shown by the errors plotted in the upper plots, falls to near zero. Also, the equalized signal shows the clear quadrature nature.

```
D  = 16;                      % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1];    % Numerator coefficients of channel
a  = [1 -0.7];                % Denominator coefficients of channel
ntr= 1000;                    % Number of iterations
s  = sign(randn(1,ntr+D))+1j*sign(randn(1,ntr+D));  % QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));      % Noise signal
r  = filter(b,a,s)+n;         % Received signal
x  = r(1+D:ntr+D);            % Input signal (received signal)
```

```
d  = s(1:ntr);                  % Desired signal (delayed QPSK signal)
lam = 0.995;                    % Forgetting factor
del = 1;                        % initialization factor
ha = adaptfilt.lsl(32,lam,del);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.');  axis([-3 3 -3 3]);
title('Received Signal Scatter Plot');  axis('square');
xlabel('Real[x]'); ylabel('Imag[x]');  grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.');  axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot');  grid on;
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]');
```



References

Haykin, S., *Adaptive Filter Theory*, 2nd Edition, Prentice Hall, N.J., 1991

See Also

adaptfilt.qrdlsl | adaptfilt.gal | adaptfilt.ftf | adaptfilt.rls

Introduced in R2011a

adaptfilt.nlms

Normalized least mean squares adaptive filter

Compatibility

`adaptfilt.nlms` has been removed. Use dsp.LMSFilter instead.

Syntax

```
ha = adaptfilt.nlms(l,step,leakage,offset,coeffs,states)
```

Description

`ha = adaptfilt.nlms(l,step,leakage,offset,coeffs,states)` constructs a normalized least-mean squares (NLMS) FIR adaptive filter object named `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.nlms`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	NLMS step size. It must be a scalar between 0 and 2. Setting this step size value to one provides the fastest convergence. `step` defaults to 1.
`leakage`	NLMS leakage factor. It must be a scalar between zero and one. When it is less than one, a leaky NLMS algorithm results. `leakage` defaults to 1 (no leakage).
`offset`	Specifies an optional offset for the denominator of the step size normalization term. You must specify offset to be a scalar

Input Argument	Description
	greater than or equal to zero. Nonzero offsets can help avoid a divide-by-near-zero condition that causes errors. Use this to avoid dividing by zero (or by very small numbers) when the square of the input data norm becomes very small (when the input signal amplitude becomes very small). When you omit it, `offset` defaults to zero.
`coeffs`	Vector composed of your initial filter coefficients. Enter a length `l` vector. `coeffs` defaults to a vector of zeros with length equal to the filter order.
`states`	Your initial adaptive filter states appear in the `states` vector. It must be a vector of length `l`-1. `states` defaults to a length `l`-1 vector with zeros for all of the elements.

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object you create. This table lists the properties for normalized LMS objects, their default values, and a brief description of the property.

Property	Range	Property Description
`Algorithm`	None	Reports the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`Leakage`	0 to 1	NLMS leakage factor. It must be a scalar between zero and one. When it is less than one, a leaky NLMS algorithm results. `leakage` defaults to 1 (no leakage).

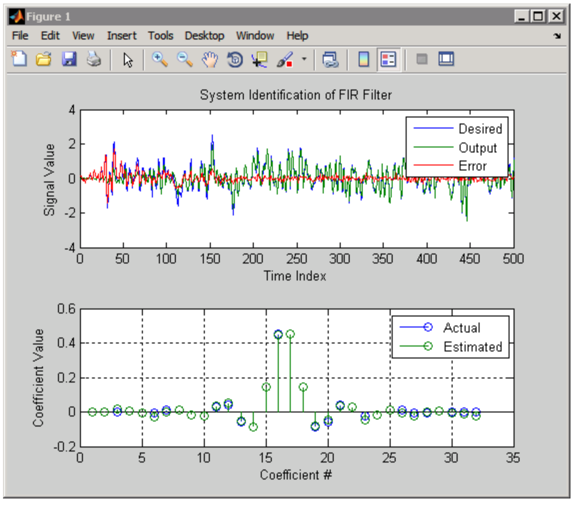
Property	Range	Property Description
`Offset`	0 or greater	Specifies an optional offset for the denominator of the step size normalization term. You must specify offset to be a scalar greater than or equal to zero. Nonzero offsets can help avoid a divide-by-near-zero condition that causes errors. Use this to avoid dividing by zero (or by very small numbers) when the square of the input data norm becomes very small (when the input signal amplitude becomes very small). When you omit it, `offset` defaults to zero.
`PersistentMemory`	`false` or `true`	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to `false`.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` - 1).
`StepSize`	0 to 1	NLMS step size. It must be a scalar between zero and one. Setting this step size value to one provides the fastest convergence. `step` defaults to one.

Examples

To help you compare this algorithm's performance to other LMS-based algorithms, such as BLMS or LMS, this example demonstrates the NLMS adaptive filter in use to identify the coefficients of an unknown FIR filter of order equal to 32 — an example used in other adaptive filter examples.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
mu = 1;                % NLMS step size
offset = 50;           % NLMS offset
ha = adaptfilt.nlms(32,mu,1,offset);
[y,e] = filter(ha,x,d);
subplot(2,1,1);
plot(1:500,[d;y;e]);
legend('Desired','Output','Error');
title('System Identification of FIR Filter');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2);
stem([b', ha.coefficients']);
legend('Actual','Estimated'); grid on;
xlabel('Coefficient #'); ylabel('Coefficient Value');
```

As you see from the figure, the `nlms` variant again closely matches the actual filter coefficients in the unknown FIR filter.



See Also

adaptfilt.ap | adaptfilt.apru | adaptfilt.lms | adaptfilt.rls | adaptfilt.swrls

Introduced in R2011a

adaptfilt.pbfdaf

Partitioned block frequency-domain FIR adaptive filter

Compatibility

`adaptfilt.pbfdaf` has been removed. Use dsp.LMSFilter, dsp.BlockLMSFilter, dsp.RLSFilter, dsp.FastTransversalFilter, dsp.AffineProjectionFilter, dsp.AdaptiveLatticeFilter, dsp.FilteredXLMSFilter, dsp.FrequencyDomainAdaptiveFilter, or dsp.AffineProjectionFilter instead.

Syntax

```
ha = adaptfilt.pbfdaf(l,step,leakage,delta,lambda,blocklen,
offset,coeffs,states)
```

Description

`ha = adaptfilt.pbfdaf(l,step,leakage,delta,lambda,blocklen,offset,coeffs,states)` constructs a partitioned block frequency-domain FIR adaptive filter `ha` that uses bin step size normalization during adaptation.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.pbfdaf`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. L defaults to 10.
`step`	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. `step` defaults to 1.

Input Argument	Description
leakage	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, a leaky version of the PBFDAF algorithm is implemented. leakage defaults to 1— no leakage.
delta	Initial common value of all of the FFT input signal powers. Its initial value should be positive. delta defaults to 1.
lambda	Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates. lambda should lie in the range (0,1]. lambda defaults to 0.9.
blocklen	Block length for the coefficient updates. This must be a positive integer such that (l/blocklen) is also an integer. For faster execution, blocklen should be a power of two. blocklen defaults to two.
offset	Offset for the normalization terms in the coefficient updates. This can be useful to avoid divide by zeros conditions, or dividing by very small numbers, if any of the FFT input signal powers become very small. offset defaults to zero.
coeffs	Initial time-domain coefficients of the adaptive filter. It should be a vector of length l. The PBFDAF algorithm uses these coefficients to compute the initial frequency-domain filter coefficient matrix via FFTs.
states	Specifies the filter initial conditions. states defaults to a zero vector of length l.

Properties

Since your adaptfilt.pbfdaf filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.pbfdaf objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation.

Name	Range	Description
AvgFactor		Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates. AvgFactor should lie in the range (0,1]. AvgFactor defaults to 0.9. Called lambda as an input argument.
BlockLength		Block length for the coefficient updates. This must be a positive integer such that (l/blocklen) is also an integer. For faster execution, blocklen should be a power of two. blocklen defaults to two.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
FFTCoefficients		Stores the discrete Fourier transform of the filter coefficients in coeffs.
FFTStates		States for the FFT operation.
Leakage	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, a leaky version of the PBFDAF algorithm is implemented. leakage defaults to 1 — no leakage.
Offset		Offset for the normalization terms in the coefficient updates. This can be useful to avoid divide by zeros conditions, or dividing by very small numbers, if any of the FFT input signal powers become very small. offset defaults to zero.

Name	Range	Description
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
Power		A vector of 2*l elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.
StepSize	0 to 1	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. step defaults to 1.

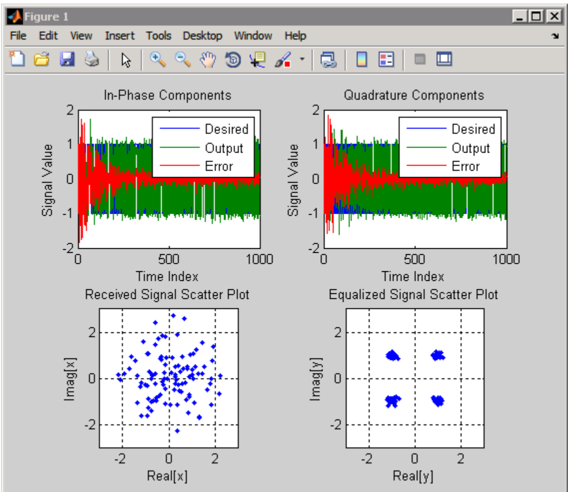
Examples

An example of Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient FIR filter.

```
D  = 16;                        % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                  % Denominator coefficients of channel
ntr = 1000;                     % Number of iterations
s  = sign(randn(1,ntr+D))+1j*sign(randn(1,ntr+D)); % Baseband QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));     % Noise signal
r  = filter(b,a,s)+n;    % Received signal
x  = r(1+D:ntr+D);              % Input signal (received signal)
d  = s(1:ntr);                   % Desired signal (delayed QPSK signal)
del = 1;                         % Initial FFT input powers
mu  = 0.1;                       % Step size
lam = 0.9;                       % Averaging factor
N   = 8;                         % Block size
ha = adaptfilt.pbfdaf(32,mu,1,del,lam,N);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e])); title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
```

```
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.');  axis([-3 3 -3 3]);
 title('Received Signal Scatter Plot');
axis('square');  xlabel('Real[x]'); ylabel('Imag[x]');  grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.');  axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot');
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]');  grid on;subplot(2,2,4); plot(y(ntr-100:ntr),'.');  ax
title('Equalized Signal Scatter Plot');
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]');  grid on;
```

In the figure shown, the four subplots provide the details of the results of the QPSK process used in the equalization for this example.



References

So, J.S. and K.K. Pang, “Multidelay Block Frequency Domain Adaptive Filter,” IEEE Trans. Acoustics, Speech, and Signal Processing, vol. 38, no. 2, pp. 373-376, February 1990

Paez Borrallo, J.M.and M.G. Otero, “On The Implementation of a Partitioned Block Frequency Domain Adaptive Filter (PBFDAF) For Long Acoustic Echo Cancellation,” Signal Processing, vol. 27, no. 3, pp. 301-315, June 1992

See Also

`adaptfilt.fdaf` | `adaptfilt.pbufdaf` | `adaptfilt.blmsfft`

Introduced in R2011a

adaptfilt.pbufdaf

Partitioned block unconstrained frequency-domain adaptive filter

Compatibility

`adaptfilt.pbufdaf` has been removed. Use dsp.LMSFilter, dsp.BlockLMSFilter, dsp.RLSFilter, dsp.FastTransversalFilter, dsp.AffineProjectionFilter, dsp.AdaptiveLatticeFilter, dsp.FilteredXLMSFilter, dsp.FrequencyDomainAdaptiveFilter, or dsp.AffineProjectionFilter instead.

Syntax

```
ha = adaptfilt.pbufdaf(l,step,leakage,delta,lambda,
blocklen,...offset,coeffs,states)
```

Description

`ha = adaptfilt.pbufdaf(l,step,leakage,delta,lambda, blocklen,...offset,coeffs,states)` constructs a partitioned block unconstrained frequency-domain FIR adaptive filter `ha` with bin step size normalization.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.pbufdaf`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. L defaults to 10.
`step`	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. `step` defaults to 1.

Input Argument	Description
leakage	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, a leaky version of the PBFDAF algorithm is implemented. leakage defaults to 1 — no leakage.
delta	Initial common value of all of the FFT input signal powers. Its initial value should be positive. delta defaults to 1.
lambda	Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates. lambda should lie in the range (0,1]. lambda defaults to 0.9.
blocklen	Block length for the coefficient updates. This must be a positive integer such that (l/blocklen) is also an integer. For faster execution, blocklen should be a power of two. blocklen defaults to two.
offset	Offset for the normalization terms in the coefficient updates. This can be useful to avoid divide by zeros conditions, or dividing by very small numbers, if any of the FFT input signal powers become very small. offset defaults to zero.
coeffs	Initial time-domain coefficients of the adaptive filter. It should be a vector of length l. The PBFDAF algorithm uses these coefficients to compute the initial frequency-domain filter coefficient matrix via FFTs.
states	Specifies the filter initial conditions. states defaults to a zero vector of length l.

Properties

Since your adaptfilt.pbufdaf filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.pbufdaf objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation

Name	Range	Description
`AvgFactor`		Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates. `AvgFactor` should lie in the range (0,1]. `AvgFactor` defaults to 0.9. Called `lambda` as an input argument.
`BlockLength`		Block length for the coefficient updates. This must be a positive integer such that (`l/blocklen`) is also an integer. For faster execution, `blocklen` should be a power of two. `blocklen` defaults to two.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`FFTCoefficients`		Stores the discrete Fourier transform of the filter coefficients in `coeffs`.
`FFTStates`		States for the FFT operation.
`Leakage`	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, a leaky version of the PBFDAF algorithm is implemented. `leakage` defaults to 1 — no leakage.
`Offset`		Offset for the normalization terms in the coefficient updates. This can be useful to avoid divide by zeros conditions, or dividing by very small numbers, if any of the FFT input signal powers become very small.v`offset` defaults to zero.

Name	Range	Description
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
Power	2*l element vector	A vector of 2*l elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.
StepSize	0 to 1	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. step defaults to 1.

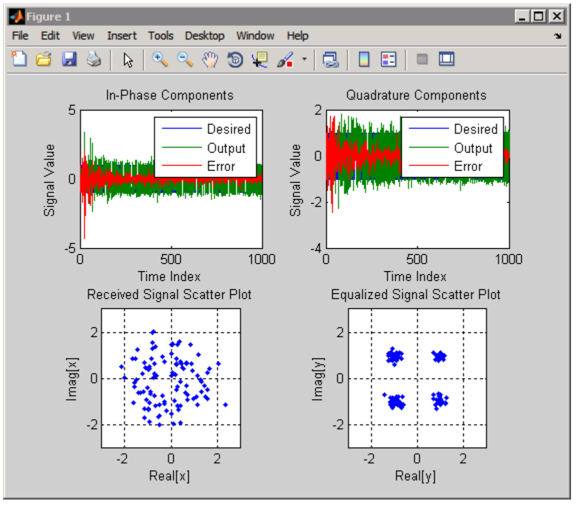
Examples

Demonstrating Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient FIR filter. To perform the equalization, this example runs for 1000 iterations.

```
D  = 16;      % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1]; % Numerator coefficients of channel
a  = [1 -0.7];             % Denominator coefficients of channel
ntr= 1000;                 % Number of iterations
s  = sign(randn(1,ntr+D))+1j*sign(randn(1,ntr+D)); % Baseband QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D)); % Noise signal
r  = filter(b,a,s)+n;          % Received signal
x  = r(1+D:ntr+D);             % Input signal (received signal)
d  = s(1:ntr);                 % Desired signal (delayed QPSK signal)
del = 1;                       % Initial FFT input powers
mu  = 0.1;                     % Step size
lam = 0.9;                     % Averaging factor
N   = 8;                       % Block size
ha = adaptfilt.pbufdaf(32,mu,1,del,lam,N);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components'); legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
```

```
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

You can compare this algorithm to another, such as the `pbfdaf` version. Use the same example of QPSK adaptation. The following figure shows the results.



References

So, J.S. and K.K. Pang, "Multidelay Block Frequency Domain Adaptive Filter," IEEE Trans. Acoustics, Speech, and Signal Processing, vol. 38, no. 2, pp. 373-376, February 1990

Paez Borrallo, J.M. and M.G. Otero, "On The Implementation of a Partitioned Block Frequency Domain Adaptive Filter (PBFDAF) for Long Acoustic Echo Cancellation," Signal Processing, vol. 27, no. 3, pp. 301-315, June 1992

See Also

`adaptfilt.ufdaf` | `adaptfilt.pbfdaf` | `adaptfilt.blmsfft`

Introduced in R2011a

adaptfilt.qrdlsl

QR-decomposition-based least-squares lattice adaptive filter

Compatibility

`adaptfilt.qrdlsl` has been removed. Use dsp.AdaptiveLatticeFilter instead.

Syntax

```
ha = adaptfilt.qrdlsl(l,lambda,delta,coeffs,states)
```

Description

`ha = adaptfilt.qrdlsl(l,lambda,delta,coeffs,states)` returns a QR-decomposition-based least squares lattice adaptive filter `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.qrdlsl`.

Input Argument	Description
`l`	Length of the joint process filter coefficients. It must be a positive integer and must be equal to the length of the prediction coefficients plus one. L defaults to 10.
`lambda`	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. `lambda` defaults to 1. `lambda` = 1 denotes infinite memory while adapting to find the new filter.
`delta`	Soft-constrained initialization factor in the least squares lattice algorithm. It should be positive. `delta` defaults to 1.

Input Argument	Description
`coeffs`	Vector of initial joint process filter coefficients. It must be a length `l` vector. `coeffs` defaults to a length `l` vector of all zeros.
`states`	Vector of the angle normalized backward prediction error states of the adaptive filter

Properties

Since your `adaptfilt.qrdlsl` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.qrdlsl` objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`BkwdPrediction`		Returns the predicted samples generated during adaptation. See “References — Adaptive Filters” for details about linear prediction.
`Coefficients`	Vector of elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`ForgettingFactor`		Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting `forgetting factor =` 1 denotes infinite memory while adapting to find the new filter. Note that this is the `lambda` input argument.

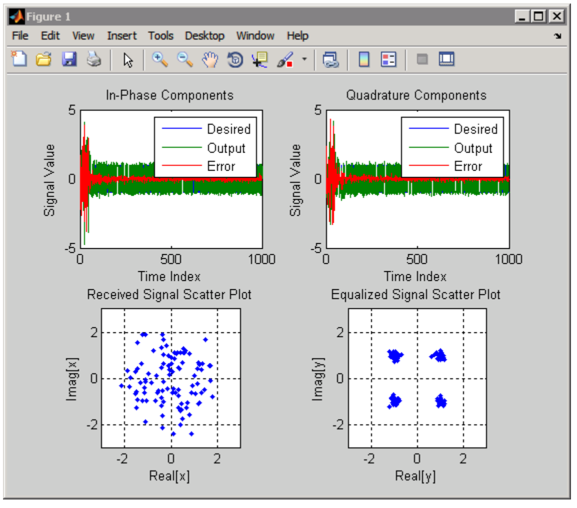
Name	Range	Description
`FwdPrediction`		Returns the predicted samples generated during adaptation in the forward direction. See "References — Adaptive Filters" for details about linear prediction.
`InitFactor`		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. `delta` defaults to one.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to `l -1`

Examples

Implement Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient adaptive filter. To see the results of the equalization process in this example, look at the figure that follows the example code.

```
D  = 16;                        % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                  % Denominator coefficients of channel
ntr= 1000;                      % Number of iterations
s  = sign(randn(1,ntr+D))+1j*sign(randn(1,ntr+D)); % Baseband  QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));   % Noise signal
r  = filter(b,a,s)+n;       % Received signal
x  = r(1+D:ntr+D);          % Input signal (received signal)
d  = s(1:ntr);              % Desired signal (delayed QPSK signal)
```

```
lam = 0.995;              % Forgetting factor
del = 1;                  % Soft-constrained initialization factor
ha = adaptfilt.qrdlsl(32,lam,del);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components'); legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components'); legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```



References

Haykin, S.,*Adaptive Filter Theory*, 2nd Edition, Prentice Hall, N.J., 1991

See Also

adaptfilt.qrdrls | adaptfilt.gal | adaptfilt.ftf | adaptfilt.lsl

Introduced in R2011a

adaptfilt.qrdrls

FIR adaptive filter that uses QR-decomposition-based RLS

Compatibility

`adaptfilt.qrdrls` has been removed. Use dsp.RLSFilter instead.

Syntax

```
ha = adaptfilt.qrdrls(l,lambda,sqrtcov,coeffs,states)
```

Description

`ha = adaptfilt.qrdrls(l,lambda,sqrtcov,coeffs,states)` constructs an FIR QR-decomposition-based recursive-least squares (RLS) adaptive filter object `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.qrdrls`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer.`l` defaults to 10.
`lambda`	RLS forgetting factor. This is a scalar and should lie within the range (0, 1]. `lambda` defaults to 1.
`sqrtcov`	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.
`coeffs`	Vector of initial filter coefficients. It must be a length `l` vector. `coeffs` defaults to length `l` vector whose elements are zeros.

Input Argument	Description
`states`	Vector of initial filter states. It must be a length `l`-1 vector. `states` defaults to a length `l`-1 vector of zeros.

Properties

Since your `adaptfilt.qrdrls` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.qrdrls` objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Vector of length `l`	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`ForgettingFactor`	Scalar	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting `forgetting factor` = 1 denotes infinite memory while adapting to find the new filter. Note that this is the `lambda` input argument.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you

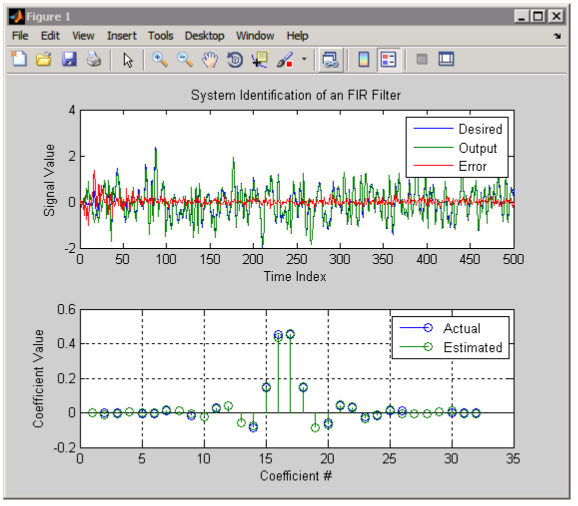
Name	Range	Description
		constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`SqrtCov`	Square matrix with each dimension equal to the filter length `l`	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.
`States`	Vector of elements	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).

Examples

System Identification of a 32-coefficient FIR filter (500 iterations).

```
x  = randn(1,500);       % Input to the filter
b  = fir1(31,0.5);       % FIR system to be identified
n  = 0.1*randn(1,500);   % Observation noise signal
d  = filter(b,1,x)+n;    % Desired signal
G0 = sqrt(.1)*eye(32);   % Initial sqrt correlation matrix
lam = 0.99;              % RLS forgetting factor
ha = adaptfilt.qrdrls(32,lam,G0);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated'); grid on;
xlabel('Coefficient #'); ylabel('Coefficient Value');
```

Using this variant of the RLS algorithm successfully identifies the unknown FIR filter, as shown here.



See Also

`adaptfilt.rls` | `adaptfilt.hrls` | `adaptfilt.hswrls` | `adaptfilt.swrls`

Introduced in R2011a

adaptfilt.rls

Recursive least-squares FIR adaptive filter

Compatibility

`adaptfilt.rls` has been removed. Use dsp.RLSFilter instead.

Syntax

```
ha = adaptfilt.rls(l,lambda,invcov,coeffs,states)
```

Description

`ha = adaptfilt.rls(l,lambda,invcov,coeffs,states)` constructs an FIR direct form RLS adaptive filter `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.rls`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`lambda`	RLS forgetting factor. This is a scalar and should lie in the range (0, 1]. `lambda` defaults to 1.
`invcov`	Inverse of the input signal covariance matrix. For best performance, you should initialize this matrix to be a positive definite matrix.
`coeffs`	Vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length `l` vector with elements equal to zero.

Input Argument	Description
states	Vector of initial filter states for the adaptive filter. It must be a length l-1 vector. states defaults to a length l-1 vector of zeros.

Properties

Since your adaptfilt.rls filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.rls objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation.
Coefficients	Vector containing l elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps. Remember that filter length is filter order + 1.
ForgettingFactor	Scalar	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting forgetting factor = 1 denotes infinite memory while adapting to find the new filter. Note that this is the lambda input argument.
InvCov	Matrix of size l-by-l	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.
KalmanGain	Vector of size (l,1)	Empty when you construct the object, this gets populated after you run the filter.

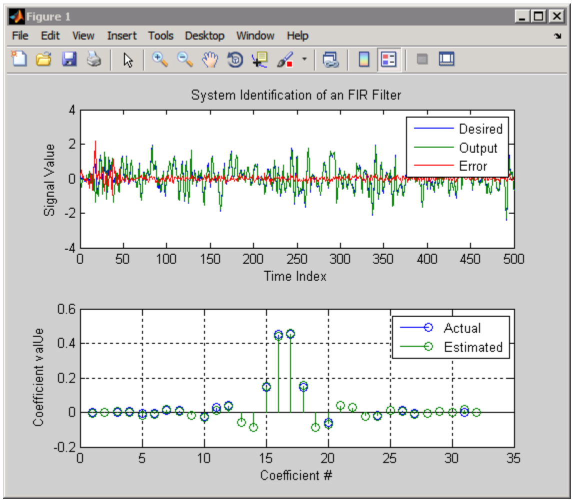
Name	Range	Description
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. Defaults to `false`.
`States`	Double array	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).

Examples

System Identification of a 32-coefficient FIR filter over 500 adaptation iterations.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
P0 = 10*eye(32); % Initial sqrt correlation matrix inverse
lam = 0.99;            % RLS forgetting factor
ha = adaptfilt.rls(32,lam,P0);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient valUe');  grid on;
```

In this example of adaptive filtering using the RLS algorithm to update the filter coefficients for each iteration, the figure shown reveals the fidelity of the derived filter after adaptation.



See Also

`adaptfilt.hrls` | `adaptfilt.hswrls` | `adaptfilt.qrdrls`

Introduced in R2011a

adaptfilt.sd

Sign-data FIR adaptive filter

Compatibility

`adaptfilt.sd` has been removed. Use dsp.LMSFilter instead.

Syntax

```
ha = adaptfilt.sd(l,step,leakage,coeffs,states)
```

Description

`ha = adaptfilt.sd(l,step,leakage,coeffs,states)` constructs an FIR sign-data adaptive filter object `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.sd`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	SD step size. It must be a nonnegative scalar. `step` defaults to 0.1
`leakage`	Your SD leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, `adaptfilt.sd` implements a leaky SD algorithm. When you omit the `leakage` property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.

Input Argument	Description
coeffs	Vector of initial filter coefficients. it must be a length l vector. coeffs defaults to length l vector with elements equal to zero.
states	Vector of initial filter states for the adaptive filter. It must be a length l-1 vector. states defaults to a length l-1 vector of zeros.

Properties

In the syntax for creating the adaptfilt object, the input options are properties of the object you create. This table lists the properties for sign-data objects, their default values, and a brief description of the property.

Property	Default Value	Description
Al gorithm	Sign-data	Defines the adaptive filter algorithm the object uses during adaptation.
Coefficients	zeros(1,l)	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need l entries in coefficients.
FilterLength	10	Reports the length of the filter, the number of coefficients or taps.
Leakage	0	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1. Defaults to 0.
PersistentMemory	false or true	Determine whether the filter states and coefficients get restored to their starting

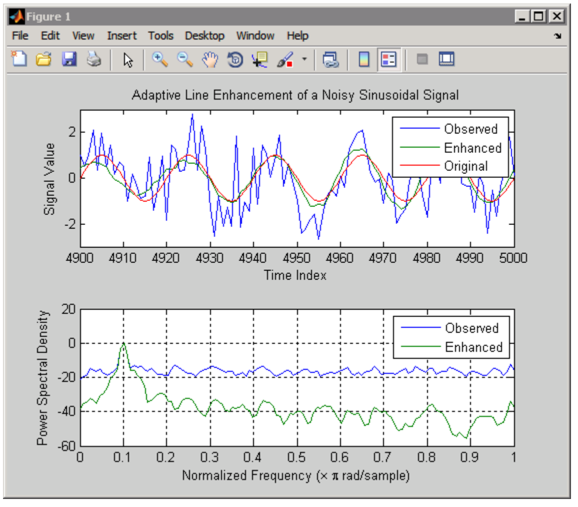
Property	Default Value	Description
		values for each filtering operation. The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to `false`.
`States`	`zeros(l-1,1)`	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` - 1).
`StepSize`	0.1	Sets the SD algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.

Examples

Adaptive line enhancement using a 32-coefficient FIR filter to perform the enhancement. This example runs for 5000 iterations, as you see in property `iter`.

```
d  = 1;                         % Number of samples of delay
ntr= 5000;                      % Number of iterations
v  = sin(2*pi*0.05*[1:ntr+d]);  % Sinusoidal signal
n  = randn(1,ntr+d);            % Noise signal
x  = v(1:ntr)+n(1:ntr);         % Input signal
d  = v(1+d:ntr+d)+n(1+d:ntr+d); % Desired signal
mu = 0.0001;                    % Sign-data step size.
ha = adaptfilt.sd(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:ntr,[d;y;v(1:end-1)]);
axis([ntr-100 ntr -3 3]);
title('Adaptive Line Enhancement of a Noisy Sinusoidal Signal');
legend('Observed','Enhanced','Original');
 xlabel('Time Index'); ylabel('Signal Value');
[pxx,om] = pwelch(x(ntr-1000:ntr));
 pyy = pwelch(y(ntr-1000:ntr));
subplot(2,1,2);
plot(om/pi,10*log10([pxx/max(pxx),pyy/max(pyy)]));
axis([0 1 -60 20]); legend('Observed','Enhanced');
xlabel('Normalized Frequency (\times \pi rad/sample)');
ylabel('Power Spectral Density'); grid on;
```

Each of the variants — sign-data, sign-error, and sign-sign — uses the same example. You can compare the results by viewing the figure shown for each adaptive filter method — `adaptfilt.sd`, `adaptfilt.se`, and `adaptfilt.ss`.



References

Moschner, J.L., “Adaptive Filter with Clipped Input Data,” Ph.D. thesis, Stanford Univ., Stanford, CA, June 1970.

Hayes, M., *Statistical Digital Signal Processing and Modeling,* New York Wiley, 1996.

See Also

`adaptfilt.lms` | `adaptfilt.se` | `adaptfilt.ss`

Introduced in R2011a

adaptfilt.se

FIR adaptive filter that uses sign-error algorithm

Compatibility

`adaptfilt.se` has been removed. Use dsp.LMSFilter instead.

Syntax

```
ha = adaptfilt.se(l,step,leakage,coeffs,states)
```

Description

`ha = adaptfilt.se(l,step,leakage,coeffs,states)` constructs an FIR sign-error adaptive filter `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.se`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	SE step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.1
`leakage`	Your SE leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, `adaptfilt.se` implements a leaky SE algorithm. When you omit the `leakage` property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.

Input Argument	Description
coeffs	Vector of initial filter coefficients. it must be a length l vector. coeffs defaults to length l vector with elements equal to zero.
states	Vector of initial filter states for the adaptive filter. It must be a length l-1 vector. states defaults to a length l-1 vector of zeros.

Properties

In the syntax for creating the adaptfilt object, the input options are properties of the object you create. This table lists the properties for the sign-error SD object, their default values, and a brief description of the property.

Property	Default Value	Description
Algorithm	Sign-error	Defines the adaptive filter algorithm the object uses during adaptation
Coefficients	zeros(1,l)	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input. Should be initialized with the initial coefficients for the FIR filter prior to adapting.
FilterLength	10	Reports the length of the filter, the number of coefficients or taps
Leakage	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1. Defaults to one if omitted.
PersistentMemory	false or true	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation.

Property	Default Value	Description
		The starting values are the values in place when you create the filter. `PersistentMemory` returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to `false`.
`States`	`zeros(l-1,1)`	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` -1).
`StepSize`	0.1	Sets the SE algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.

Use `inspect(ha)` to view or change the object properties graphically using the MATLAB Property Inspector.

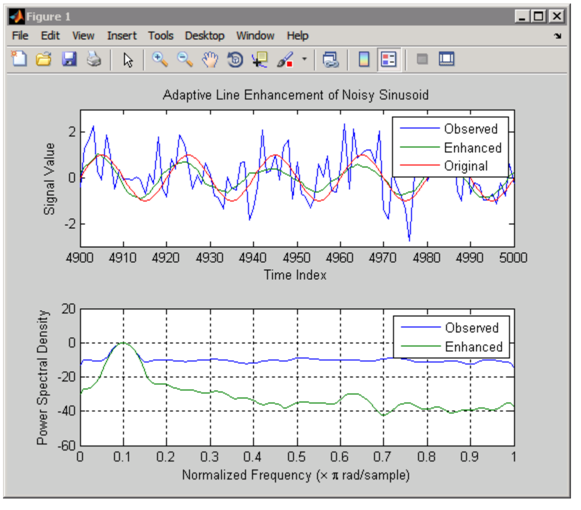
Examples

Adaptive line enhancement using a 32-coefficient FIR filter running over 5000 iterations.

```
d = 1;                        % Number of samples of delay
ntr= 5000;                    % Number of iterations
v = sin(2*pi*0.05*(1:ntr+d));    % Sinusoidal signal
n = randn(1,ntr+d);           % Noise signal
x = v(1:ntr)+n(1:ntr);        % Input signal --(delayed desired signal)
d = v(1+d:ntr+d)+n(1+d:ntr+d);  % Desired signal
mu = 0.0001;                    % Sign-error step size
ha = adaptfilt.se(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1);
plot(1:ntr,[d;y;v(1:end-1)]);
axis([ntr-100 ntr -3 3]);
title('Adaptive Line Enhancement of Noisy Sinusoid');
legend('Observed','Enhanced','Original');
xlabel('Time Index'); ylabel('Signal Value');
HWelch = spectrum.welch;
InputPsd = psd(HWelch,x(ntr-1000:ntr));
OutputPsd = psd(HWelch,y(ntr-1000:ntr));
CompPsdEst = [InputPsd.Data/max(InputPsd.Data), OutputPsd.Data/max(OutputPsd.Data)];
subplot(2,1,2); plot(InputPsd.Frequencies/pi,10*log10(CompPsdEst));
```

```
axis([0 1 -60 20]); legend('Observed','Enhanced');
xlabel('Normalized Frequency (\times \pi rad/sample)');
ylabel('Power Spectral Density'); grid on;
```

Compare the figure shown here to the ones for `adaptfilt.sd` and `adaptfilt.ss` to see how the variants perform on the same example.



References

Gersho, A, “Adaptive Filtering With Binary Reinforcement,” IEEE Trans. Information Theory, vol. IT-30, pp. 191-199, March 1984.

Hayes, M, *Statistical Digital Signal Processing and Modeling*, New York, Wiley, 1996.

See Also

`adaptfilt.sd` | `adaptfilt.ss` | `adaptfilt.lms`

Introduced in R2011a

adaptfilt.ss

FIR adaptive filter that uses sign-sign algorithm

Compatibility

`adaptfilt.ss` has been removed. Use dsp.LMSFilter instead.

Syntax

```
ha = adaptfilt.ss(l,step,leakage,coeffs,states)
```

Description

`ha = adaptfilt.ss(l,step,leakage,coeffs,states)` constructs an FIR sign-error adaptive filter `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.ss`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	SS step size. It must be a nonnegative scalar. `step` defaults to 0.1.
`leakage`	Your SS leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, `adaptfilt.lms` implements a leaky SS algorithm. When you omit the `leakage` property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.

Input Argument	Description
coeffs	Vector of initial filter coefficients. it must be a length l vector. coeffs defaults to length l vector with elements equal to zero.
states	Vector of initial filter states for the adaptive filter. It must be a length l -1 vector. states defaults to a length l-1 vector of zeros.

adaptfilt.ss can be called for a block of data, when x and d are vectors, or in "sample by sample mode" using a For-loop with the method filter:

```
for n = 1:length(x)
ha = adaptfilt.ss(25,0.9);
[y(n),e(n)] = filter(ha,(x(n),d(n),s));
end
```

Properties

In the syntax for creating the adaptfilt object, most of the input options are properties of the object you create. This table lists the properties for sign-sign objects, their default values, and a brief description of the property.

Property	Default Value	Description
Algorithm	Sign-sign	Defines the adaptive filter algorithm the object uses during adaptation
Coefficients	zeros(1,l)	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input. Should be initialized with the initial coefficients for the FIR filter prior to adapting.
FilterLength	10	Reports the length of the filter, the number of coefficients or taps
Leakage	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even

Property	Default Value	Description
		after it reaches a minimum value. Ranges between 0 and 1. 1 is the default value.
PersistentMemory	false or true	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to false.
States	zeros(l-1,1)	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l-1).
StepSize	0.1	Sets the SE algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.

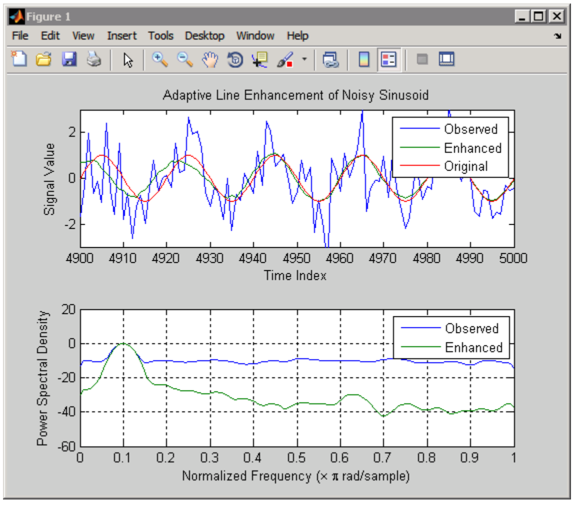
Examples

Demonstrating adaptive line enhancement using a 32-coefficient FIR filter provides a good introduction to the sign-sign algorithm.

```
d = 1;                        % Number of samples of delay
ntr= 5000;                    % Number of iterations
v = sin(2*pi*0.05*(1:ntr+d));    % Sinusoidal signal
n = randn(1,ntr+d);           % Noise signal
x = v(1:ntr)+n(1:ntr);        % Input signal --(delayed desired signal)
d = v(1+d:ntr+d)+n(1+d:ntr+d);  % Desired signal
mu = 0.0001;                    % Sign-error step size
ha = adaptfilt.ss(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1);
plot(1:ntr,[d;y;v(1:end-1)]);
axis([ntr-100 ntr -3 3]);
title('Adaptive Line Enhancement of Noisy Sinusoid');
legend('Observed','Enhanced','Original');
xlabel('Time Index'); ylabel('Signal Value');
```

```
[InputPsd,wi] = pwelch(x(ntr-1000:ntr));
[OutputPsd,wo] = pwelch(y(ntr-1000:ntr));
CompPsdEst = [InputPsd/max(InputPsd), OutputPsd/max(OutputPsd)];
subplot(2,1,2); plot(wi/pi,10*log10(CompPsdEst));
axis([0 1 -60 20]); legend('Observed','Enhanced');
xlabel('Normalized Frequency (\times \pi rad/sample)');
ylabel('Power Spectral Density'); grid on;
```

This example is the same as the ones used for the sign-data and sign-error examples. Comparing the figures shown for each of the others lets you assess the performance of each for the same task.



References

Lucky, R.W, "Techniques For Adaptive Equalization of Digital Communication Systems," Bell Systems Technical Journal, vol. 45, pp. 255-286, Feb. 1966

Hayes, M., *Statistical Digital Signal Processing and Modeling*, New York, Wiley, 1996.

See Also

adaptfilt.se | adaptfilt.sd | adaptfilt.lms

Introduced in R2011a

adaptfilt.swftf

FIR adaptive filter that uses sliding window fast transversal least squares

Compatibility

`adaptfilt.swftf` has been removed. Use dsp.FastTransversalFilter instead.

Syntax

```
ha = adaptfilt.swftf(l,delta,blocklen,gamma,gstates,
dstates,...coeffs,states)
```

Description

`ha = adaptfilt.swftf(l,delta,blocklen,gamma,gstates, dstates,...coeffs,states)` constructs a sliding window fast transversal least squares adaptive filter `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.swftf`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`delta`	Soft-constrained initialization factor. This scalar should be positive and sufficiently large to maintain stability. `delta` defaults to 1.
`blocklen`	Block length of the sliding window. This must be an integer at least as large as the filter length `l`, which is the default value.
`gamma`	Conversion factor. `gamma` defaults to the matrix [1 -1] that specifies soft-constrained initialization.

Input Argument	Description
gstates	States of the Kalman gain updates. gstates defaults to a zero vector of length (l + blocklen - 1).
dstates	Desired signal states of the adaptive filter. dstates defaults to a zero vector of length equal to (blocklen - 1). For a default object, dstates is (l-1).
coeffs	Vector of initial filter coefficients. It must be a length l vector. coeffs defaults to length l vector of all zeros.
states	Vector of initial filter states. states defaults to a zero vector of length equal to (l + blocklen - 2).

Properties

Since your adaptfilt.swftf filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.swftf objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
BkwdPredictions		Returns the predicted samples generated during adaptation. See “References — Adaptive Filters” for details about linear prediction.
BlockLength		Block length of the sliding window. This must be an integer at least as large as the filter length l, which is the default value.
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.

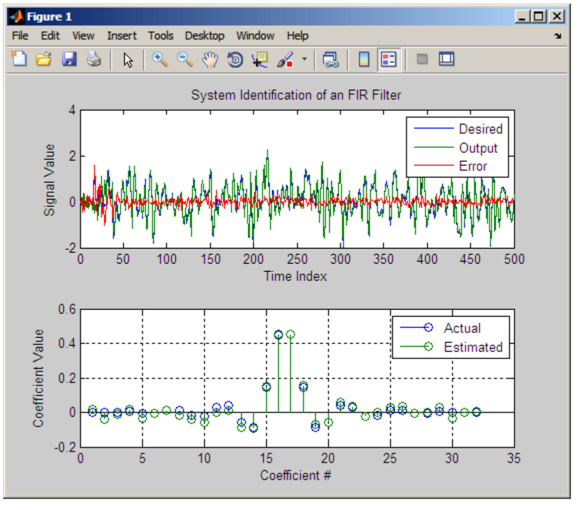
Name	Range	Description
`ConversionFactor`		Conversion factor. Called `gamma` when it is an input argument, it defaults to the matrix [1 -1] that specifies soft-constrained initialization.
`DesiredSignal States`		Desired signal states of the adaptive filter. `dstates` defaults to a zero vector with length equal to (`blocklen` - 1).
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`FwdPrediction`		Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.
`InitFactor`		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. `delta` defaults to one.
`KalmanGain`		Empty when you construct the object, this gets populated after you run the filter.
`KalmanGainStates`		Contains the states of the Kalman gains for the adaptive algorithm. Initialized to a vector of double data type entries.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false`.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).

Examples

Over 500 iterations, perform a system identification of a 32-coefficient FIR filter.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
L  = 32;               % Adaptive filter length
del = 0.1;             % Soft-constrained initialization factor
N  = 64;               % block length
ha = adaptfilt.swftf(L,del,N);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');  grid on;
```

Review the figure for the results of the example. When you evaluate the example you should get the same results, within the differences in the random noise signal you use.



References

Slock, D.T.M., and T. Kailath, “A Modular Prewindowing Framework for Covariance FTF RLS Algorithms,” Signal Processing, vol. 28, pp. 47-61, 1992

Slock, D.T.M., and T. Kailath, “A Modular Multichannel Multi-Experiment Fast Transversal Filter RLS Algorithm,” Signal Processing, vol. 28, pp. 25-45, 1992

See Also

`adaptfilt.ftf` | `adaptfilt.swrls` | `adaptfilt.ap` | `adaptfilt.apru`

Introduced in R2011a

adaptfilt.swrls

FIR adaptive filter that uses window recursive least squares (RLS)

Compatibility

`adaptfilt.swrls` has been removed. Use dsp.RLSFilter instead.

Syntax

```
ha = adaptfilt.swrls(l,lambda,invcov,swblocklen,
dstates,...coeffs,states)
```

Description

`ha = adaptfilt.swrls(l,lambda,invcov,swblocklen, dstates,...coeffs,states)` constructs an FIR sliding window RLS adaptive filter `ha`.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.swrls`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps). It must be a positive integer. `l` defaults to 10.
`lambda`	RLS forgetting factor. This is a scalar and should lie within the range (0, 1]. `lambda` defaults to 1.
`invcov`	Inverse of the input signal covariance matrix. You should initialize `invcov` to a positive definite matrix.

Input Argument	Description
`swblocklen`	Block length of the sliding window. This integer must be at least as large as the filter length. `swblocklen` defaults to 16.
`dstates`	Desired signal states of the adaptive filter. `dstates` defaults to a zero vector with length equal to (`swblocklen` - 1).
`coeffs`	Vector of initial filter coefficients. It must be a length `l` vector. `coeffs` defaults to length `l` vector of all zeros.
`states`	Vector of initial filter states. `states` defaults to a zero vector of length equal to (l + `swblocklen` - 2).

Properties

Since your `adaptfilt.swrls` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.swrls` objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
`Algorithm`	None	Defines the adaptive filter algorithm the object uses during adaptation
`Coefficients`	Any vector of `l` elements	Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.
`DesiredSignalStates`	Vector	Desired signal states of the adaptive filter. `dstates` defaults to a zero vector with length equal to (`swblocklen` - 1).
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`ForgettingFactor`	Scalar	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1.

Name	Range	Description
		Setting `forgetting factor` = 1 denotes infinite memory while adapting to find the new filter. Note that this is the `lambda` input argument.
`InvCov`	Matrix	Square matrix with each dimension equal to the filter length `l`.
`KalmanGain`	Vector with dimensions (`l`,1)	Empty when you construct the object, this gets populated after you run the filter.
`PersistentMemory`	`false` or `true`	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. Defaults to `false`.
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` + `swblocklen` - 2)
`SwBlockLength`	Integer	Block length of the sliding window. This integer must be at least as large as the filter length. `swblocklen` defaults to 16.

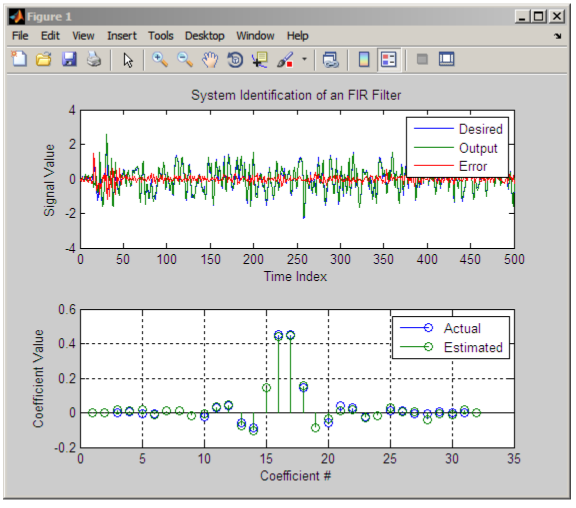
Examples

System Identification of a 32-coefficient FIR filter. Use 500 iterations to adapt to the unknown filter. After the example code, you see a figure that plots the results of the running the code.

```
x  = randn(1,500);     % Input to the filter
```

```
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
P0 = 10*eye(32);       % Initial correlation matrix inverse
lam = 0.99;            % RLS forgetting factor
N  = 64;               % Block length
ha = adaptfilt.swrls(32,lam,P0,N);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
 title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');  grid on;
```

In the figure you see clearly that the adaptive filter process successfully identified the coefficients of the unknown FIR filter. You knew it had to or many things that you take for granted, such as modems on computers, would not work.



See Also

adaptfilt.rls | adaptfilt.qrdrls | adaptfilt.hswrls

Introduced in R2011a

adaptfilt.tdafdct

Adaptive filter that uses discrete cosine transform

Compatibility

`adaptfilt.tdafdct` has been removed. Use dsp.LMSFilter, dsp.BlockLMSFilter, dsp.RLSFilter, dsp.FastTransversalFilter, dsp.AffineProjectionFilter, dsp.AdaptiveLatticeFilter, dsp.FilteredXLMSFilter, dsp.FrequencyDomainAdaptiveFilter, or dsp.AffineProjectionFilter instead.

Syntax

```
ha = adaptfilt.tdafdct(l,step,leakage,offset,delta,lambda,
coeffs,states)
```

Description

`ha = adaptfilt.tdafdct(l,step,leakage,offset,delta,lambda, coeffs,states)` constructs a transform-domain adaptive filter ha object that uses the discrete cosine transform to perform filter adaptation.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.tdafdct`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	Adaptive filter step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.

Input Argument	Description
leakage	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the TDAFDCT algorithm. leakage defaults to 1 — no leakage.
offset	Offset for the normalization terms in the coefficient updates. You can use this argument to avoid dividing by zero or by very small numbers when any of the FFT input signal powers become very small. offset defaults to zero.
delta	Initial common value of all of the transform domain powers. Its initial value should be positive. delta defaults to 5.
lambda	Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. lambda should lie between zero and one. For default filter objects, lambda equals (1 - step).
coeffs	Initial time domain coefficients of the adaptive filter. Set it to be a length l vector. coeffs defaults to a zero vector of length l.
states	Initial conditions of the adaptive filter. states defaults to a zero vector with length equal to (l - 1).

Properties

Since your adaptfilt.tdafdct filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.tdafdct objects. To show you the properties that apply, this table lists and describes each property for the transform domain filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation.
AvgFactor		Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. AvgFactor should lie between zero and one. For

Name	Range	Description
		default filter objects, AvgFactor equals (1 - step). lambda is the input argument that represent AvgFactor.
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
Leakage	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the TDAFDFT algorithm. leakage defaults to 1 — no leakage.
Offset		Offset for the normalization terms in the coefficient updates. You can use this argument to avoid dividing by zeros or by very small numbers when any of the FFT input signal powers become very small. offset defaults to zero.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
Power	2*l element vector	A vector of 2*l elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.

Name	Range	Description
`States`	Vector of elements, data type double	Vector of the adaptive filter states. `states` defaults to a vector of zeros which has length equal to (`l` + `projectord` - 2).
`StepSize`	0 to 1	Step size. It must be a nonnegative scalar, greater than zero and less than or equal to 1. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.

For checking the values of properties for an adaptive filter object, use `get(ha)` or enter the object name, without a trailing semicolon, at the MATLAB prompt.

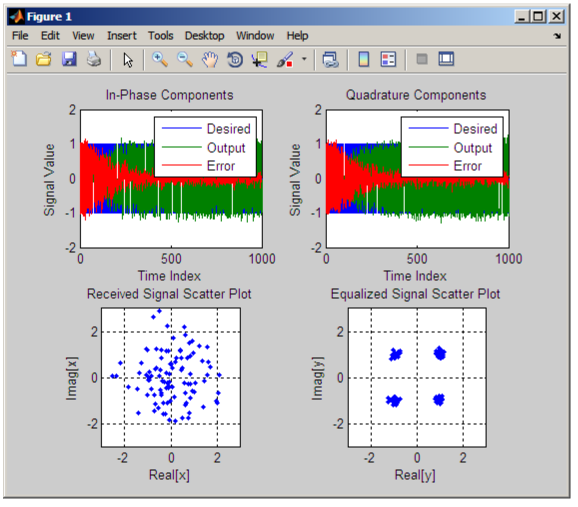
Examples

Using 1000 iterations, perform a Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient FIR filter.

```
D  = 16;                        % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                  % Denominator coefficients of channel
ntr= 1000;                      % Number of iterations
s  = sign(randn(1,ntr+D)) + 1j*sign(randn(1,ntr+D)); %QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D)); % Noise signal
r  = filter(b,a,s)+n;           % Received signal
x  = r(1+D:ntr+D);              % Input signal (received signal)
d  = s(1:ntr);                  % Desired signal (delayed QPSK signal)
L  = 32;                        % filter length
mu = 0.01;                      % Step size
ha = adaptfilt.tdafdct(L,mu);
[y,e] = filter(ha,x,d);
subplot(2,2,1);
plot(1:ntr,real([d;y;e])); title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.');
axis([-3 3 -3 3]); title('Received Signal Scatter Plot');
axis('square');  xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
```

```
title('Equalized Signal Scatter Plot'); grid on;
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]');
```

Compare the plots shown in this figure to those in the other time domain filter variations. The comparison should help you select and understand how the variants differ.



References

Haykin, S.,*Adaptive Filter Theory*, 3rd Edition, Prentice Hall, N.J., 1996.

See Also

`adaptfilt.tdafdft` | `adaptfilt.fdaf` | `adaptfilt.blms`

Introduced in R2011a

adaptfilt.tdafdft

Adaptive filter that uses discrete Fourier transform

Compatibility

`adaptfilt.tdafdft` has been removed. Use dsp.LMSFilter, dsp.BlockLMSFilter, dsp.RLSFilter, dsp.FastTransversalFilter, dsp.AffineProjectionFilter, dsp.AdaptiveLatticeFilter, dsp.FilteredXLMSFilter, dsp.FrequencyDomainAdaptiveFilter, or dsp.AffineProjectionFilter instead.

Syntax

```
ha = adaptfilt.tdafdft(l,step,leakage,offset,
delta,lambda,...coeffs,states)
```

Description

`ha = adaptfilt.tdafdft(l,step,leakage,offset, delta,lambda,...coeffs,states)` constructs a transform-domain adaptive filter object `ha` using a discrete Fourier transform.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.tdafdft`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	Adaptive filter step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.

Input Argument	Description
leakage	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the TDAFDFT algorithm. leakage defaults to 1 — no leakage.
offset	Offset for the normalization terms in the coefficient updates. You can use this argument to avoid dividing by zeros or by very small numbers when any of the FFT input signal powers become very small. offset defaults to zero.
delta	Initial common value of all of the transform domain powers. Its initial value should be positive. delta defaults to 5.
lambda	Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. lambda should lie between zero and one. For default filter objects, LAMBDA equals (1 - step).
coeffs	Initial time domain coefficients of the adaptive filter. Set it to be a length l vector. coeffs defaults to a zero vector of length l.
states	Initial conditions of the adaptive filter. states defaults to a zero vector with length equal to (l - 1).

Properties

Since your adaptfilt.tdafdft filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.tdafdft objects. To show you the properties that apply, this table lists and describes each property for the transform domain filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
AvgFactor		Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. AvgFactor should lie between zero and one. For

Name	Range	Description
		default filter objects, AvgFactor equals (1 - step). lambda is the input argument that represent AvgFactor.
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
Leakage	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the TDAFDFT algorithm. leakage defaults to 1 — no leakage.
Offset		Offset for the normalization terms in the coefficient updates. You can use this argument to avoid dividing by zeros or by very small numbers when any of the FFT input signal powers become very small. offset defaults to zero.
PersistentMemory	false or true	Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
Power	2*l element vector	A vector of 2*l elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.

Name	Range	Description
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).
StepSize	0 to 1	Step size. It must be a nonnegative scalar, greater than zero and less than or equal to 1. step defaults to 0.

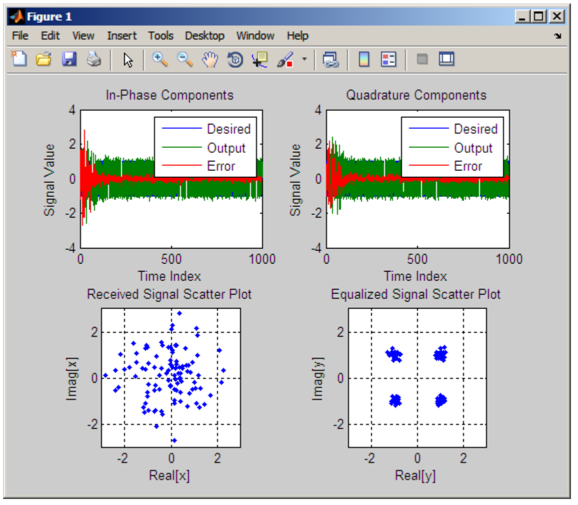
Examples

Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient FIR filter (1000 iterations).

```
D  = 16;                        % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                  % Denominator coefficients of channel
ntr= 1000;                      % Number of iterations
s  = sign(randn(1,ntr+D)) + 1j*sign(randn(1,ntr+D));% Baseband QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));      % Noise signal
r  = filter(b,a,s)+n;            % Received signal
x  = r(1+D:ntr+D);               % Input signal (received signal)
d  = s(1:ntr);                   % Desired signal (delayed QPSK signal)
L  = 32;                        % filter length
mu = 0.01;                      % Step size
ha = adaptfilt.tdafdft(L,mu);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.');
 axis([-3 3 -3 3]); title('Received Signal Scatter Plot');
axis('square');  xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.');
axis([-3 3 -3 3]); title('Equalized Signal Scatter Plot');
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

All of the time domain adaptive filter reference pages use this QPSK example. By comparing the results for each variation you get an idea of the differences in the way each one performs.

This figure demonstrates the results of running the example code shown.



References

Haykin, S.,*Adaptive Filter Theory*, 3rd Edition, Prentice Hall, N.J., 1996

See Also

`adaptfilt.tdafdct` | `adaptfilt.fdaf` | `adaptfilt.blms`

Introduced in R2011a

adaptfilt.ufdaf

FIR adaptive filter that uses unconstrained frequency-domain with quantized step size normalization

Compatibility

`adaptfilt.ufdaf` has been removed. Use dsp.FrequencyDomainAdaptiveFilter instead.

Syntax

```
ha = adaptfilt.ufdaf(l,step,leakage,delta,lambda,blocklen,
offset,coeffs,states)
```

Description

`ha = adaptfilt.ufdaf(l,step,leakage,delta,lambda,blocklen, offset,coeffs,states)` constructs an unconstrained frequency-domain FIR adaptive filter `ha` with quantized step size normalization.

For information on how to run data through your adaptive filter object, see the Adaptive Filter Syntaxes section of the reference page for `filter`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.ufdaf`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	Adaptive filter step size. It must be a nonnegative scalar. `step` defaults to 0.
`leakage`	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the UFDAF algorithm. `leakage` defaults to 1 — no leakage.

Input Argument	Description
delta	Initial common value of all of the FFT input signal powers. the initial value of delta should be positive, and it defaults to 1.
lambda	Specifies the averaging factor used to compute the exponentially-windowed FFT input signal powers for the coefficient updates. lambda should lie in the range (0,1]. For default UFDAF filter objects, lambda defaults to 0.9.
blocklen	Block length for the coefficient updates. This must be a positive integer. For faster execution, (blocklen l) should be a power of two. blocklen defaults to l.
offset	Offset for the normalization terms in the coefficient updates. This can help you avoid divide by zero conditions, or divide by very small numbers conditions, when any of the FFT input signal powers become very small. Default value is zero.
coeffs	Initial time-domain coefficients of the adaptive filter. It should be a length l vector. The filter object uses these coefficients to compute the initial frequency-domain filter coefficients via an FFT computed after zero-padding the time-domain vector by blocklen.
states	Adaptive filter states. states defaults to a zero vector with length equal to l.

Properties

Since your adaptfilt.ufdaf filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.ufdaf objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
AvgFactor		Specifies the averaging factor used to compute the exponentially-windowed FFT input signal powers for the

Name	Range	Description
		coefficient updates. `AvgFactor` should lie in the range (0,1]. For default UFDAF filter objects, `AvgFactor` defaults to 0.9. Note that AvgFactor and lambda are the same thing — lambda is an input argument and AvgFactor a property of the object.
`BlockLength`		Block length for the coefficient updates. This must be a positive integer. For faster execution, (`blocklen` + `1`) should be a power of two. `blocklen` defaults to `1`.
`FFTCoefficients`		Stores the discrete Fourier transform of the filter coefficients in `coeffs`.
`FFTStates`		States for the FFT operation.
`FilterLength`	Any positive integer	Reports the length of the filter, the number of coefficients or taps
`Leakage`	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the UFDAF algorithm. `leakage` defaults to 1 — no leakage.
`Offset`		Offset for the normalization terms in the coefficient updates. This can help you avoid divide by zero conditions, or divide by very small numbers conditions, when any of the FFT input signal powers become very small. Default value is zero.

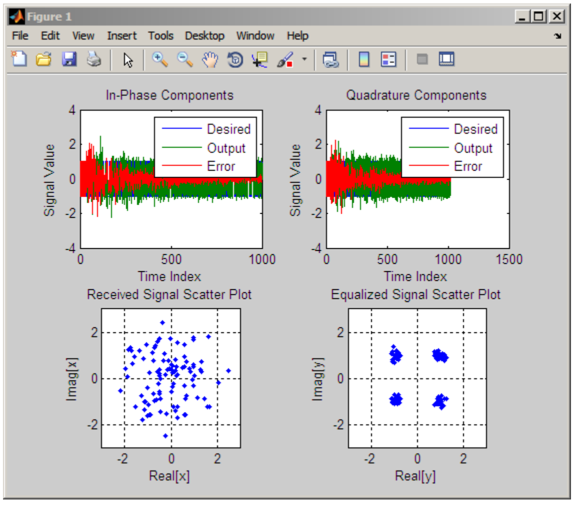
Name	Range	Description
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
Power	2*l element vector	A vector of 2*l elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.
StepSize	0 to 1	Adaptive filter step size. It must be a nonnegative scalar. You can use maxstep to determine a reasonable range of step size values for the signals being processed. step defaults to 0.

Examples

Show an example of Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient adaptive filter. For fidelity, use 1024 iterations. The figure that follows the code provides the information you need to assess the performance of the equalization process.

```
D  = 16;                         % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1];       % Numerator coefficients of channel
a  = [1 -0.7];                   % Denominator coefficients of channel
ntr= 1024;                       % Number of iterations
s = sign(randn(1,ntr+D))+1j*sign(randn(1,ntr+D));% Baseband QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));
r  = filter(b,a,s)+n;% Received signal
x  = r(1+D:ntr+D);   % Input signal (received signal)
d  = s(1:ntr);       % Desired signal (delayed QPSK signal)
del = 1;            %Initial FFT input powers
mu  = 0.1;           % Step size
lam = 0.9;          % Averaging factor
ha = adaptfilt.ufdaf(32,mu,1,del,lam);
```

```
[y,e] = filter(ha,x,d);
subplot(2,2,1);  plot(1:1000,real([d(1:1000);y(1:1000);e(1:1000)]));
title('In-Phase Components'); legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components'); legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```



References

Shynk, J.J.,“Frequency-domain and Multirate Adaptive Filtering,” IEEE Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992

See Also

`adaptfilt.fdaf` | `adaptfilt.pbufdaf` | `adaptfilt.blms` | `adaptfilt.blmsfft`

Introduced in R2011a

allpass2wdf

Allpass to Wave Digital Filter coefficient transformation

Syntax

```
w = allpass2wdf(a)
W = allpass2wdf(A)
```

Description

`w = allpass2wdf(a)` accepts a vector of real allpass polynomial filter coefficients `a`, and returns the transformed coefficient `w`. `w` can be used with allpass filter objects such as `dsp.AllpassFilter`, and `dsp.CoupledAllpassFilter`, with Structure set to `'Wave Digital Filter'`.

`W = allpass2wdf(A)` accepts the cell array of allpass polynomial coefficient vectors `A`. Each cell of `A` holds the coefficients of a section of a cascade allpass filter. `W` is also a cell array, and each cell of `W` contains the transformed version of the coefficients in the corresponding cell of `A`. W can be used with allpass filter objects such as dsp.AllpassFilter and dsp.CoupledAllpassFilter, with structure set to `'Wave Digital Filter'`.

Examples

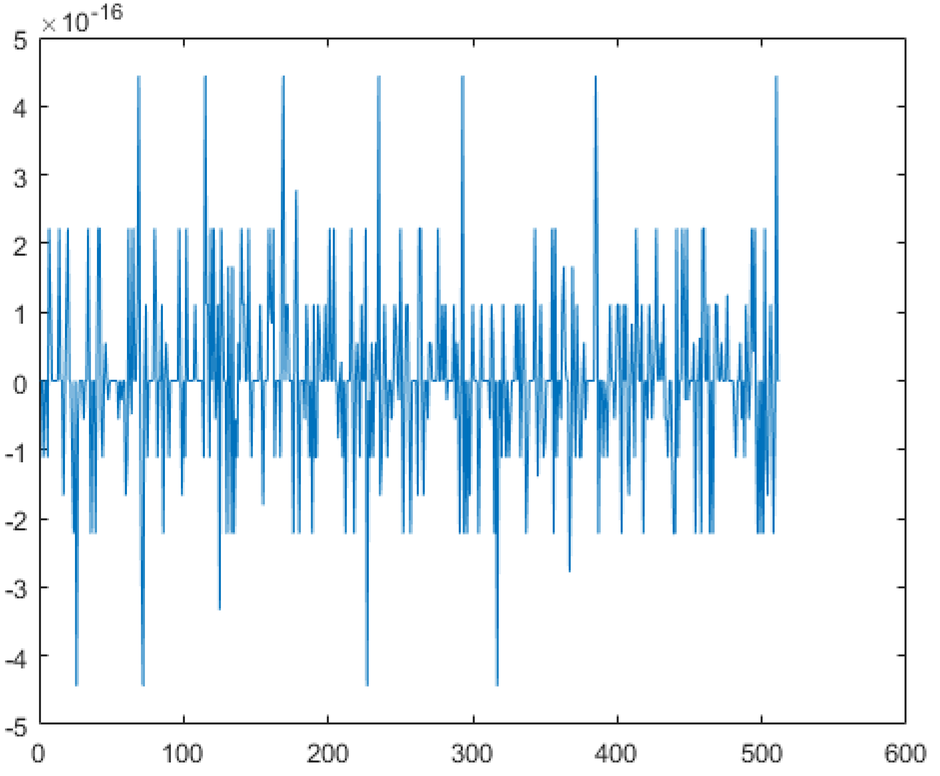
Convert Allpass coefficients to Wave Digital Filter Coefficients

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create a second order allpass filter with coefficients a = [0 0.5]. Convert these coefficients into wave digital filter form using `allpass2wdf`. Assign the transformed coefficients to an allpass filter using the wave digital filter structure. Pass a random input to both these filters and compare the outputs.

```
a = [0 0.5];
allpass = dsp.AllpassFilter('AllpassCoefficients', a);
w = allpass2wdf(a);
```

```
allpasswdf = dsp.AllpassFilter('Structure', 'Wave Digital Filter',...
    'WDFCoefficients', w);
in = randn(512, 1);
outputAllpass = allpass(in);
outputAllpasswdf = allpasswdf(in);
plot(outputAllpass-outputAllpasswdf)
```



The difference between the two outputs is very small.

Input Arguments

a — allpass filter coefficients

(default) | vector of real numbers

Numeric vector of allpass filter coefficients, specified as real numbers. `a` can have length only equal to 1,2, and 4. When the length is 4, the first and third components must both be zero. `a` can be a row or a column vector.

Example: `0.7`

Data Types: `double` | `single`

A — allpass filter coefficients
(default) | vector of cells

Cascade of allpass filter coefficients, specified as a cell vector. Every cell of `A` must contain a real vector of length 1,2, or 4. When the length is 4, the first and third components must both be zero. `A` can be a row or column vector of cells.

Example: `{0.7, [0.1, 0.2]}`

Data Types: `double` | `single`

Output Arguments

w — transformed version of the coefficients a
(default) | vector of real numbers

Numeric vector of transformed coefficients, determined as a real number, to use with single-section allpass filter objects having `Structure` set to `'Wave Digital Filter'`. `w` is always returned as a numeric row vector.

Example: `0.7`

Data Types: `double` | `single`

W — transformed version of the coefficients cell array A
(default) | vector of cell

Cascade of transformed allpass filter coefficients, determined as a cell array, to use with multi-section allpass filter objects having `Structure` set to `'Wave Digital Filter'`. `W` is always returned as a column of cells.

Example: `{0.7;[0.2,-0.0833]}`

Data Types: `double` | `single`

Algorithms

In the more general case, the input coefficients A define a cascade or multisection allpass filter. `allpass2wdf` applies separately to each section of the same transformation used in the single-section case. In the single-section case, the numeric coefficients vector a contains a standard polynomial representation of an allpass filter of order 1, 2, or 4. For example, in the first order case,

$$a = [a_1]$$

represents the first order transfer function:

$$H_1(z) = \frac{z^{-1} + a_1}{1 + a_1 z^{-1}}$$

and in the second order case,

$$a = [a_1, a_2]$$

represents the second order transfer function:

$$H_2(z) = \frac{z^{-2} + a_1 z^{-1} + a_2}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

.

The allpass transfer functions H_1 and H_2 can also have the following alternative representations, using decoupled coefficients in vector w_1 or w_2 respectively.

$$\tilde{H}1(z) = \frac{z^{-1} + w_1}{1 + w_1 z^{-1}}$$

$$\tilde{H}2(z) = \frac{z^{-2} + w_2(1 + w_1)z^{-1} + w_1}{1 + w_2(1 + w_1)z^{-1} + w_1 z^{-2}}$$

For allpass coefficients, w is often used to derive adaptor multipliers for Wave Digital Filter structures, and it is required by a number of allpass based filters in DSP System Toolbox when `Structure` is set to `'Wave Digital Filter'` (e.g. dsp.AllpassFilter, and dsp.CoupledAllpassFilter).

For a given vector of section coefficients a, `allpass2wdf` computes the corresponding vector w such that

$$when\ i = 1,\ 2\ or\ 4$$

$$\tilde{H}_i(z) = H_i(z)$$

This results in using the following formulas:

$$\begin{aligned}
&\textit{for order } 1: \\
&w_1 = a_1 \\
&\textit{for order } 2: \\
&w_1 = a_2 \\
&w_2 = \frac{a_1}{1+a_2} \\
&\textit{for order } 4: \\
&w_1 = a_4 \\
&w_3 = \frac{a_2}{1+a_4} \\
&w_2 = w_4 = 0
\end{aligned}$$

References

[1] M. Lutovac, D. Tosic, B. Evans, *Filter Design for Signal Processing using MATLAB and Mathematica*. Prentice Hall, 2001.

See Also

See Also

dsp.AllpassFilter | dsp.CoupledAllpassFilter | `tf2ca` | `tf2latc` | `wdf2allpass`

Introduced in R2014a

allpassbpc2bpc

Allpass filter for complex bandpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpassbpc2bpc(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpassbpc2bpc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a complex bandpass to complex bandpass frequency transformation. This transformation effectively places two features of an original filter, located at frequencies W_{o1} and W_{o2}, at the required target frequency locations W_{t1} and W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. In most of the cases the features selected for the transformation are the band edges of the filter passbands. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

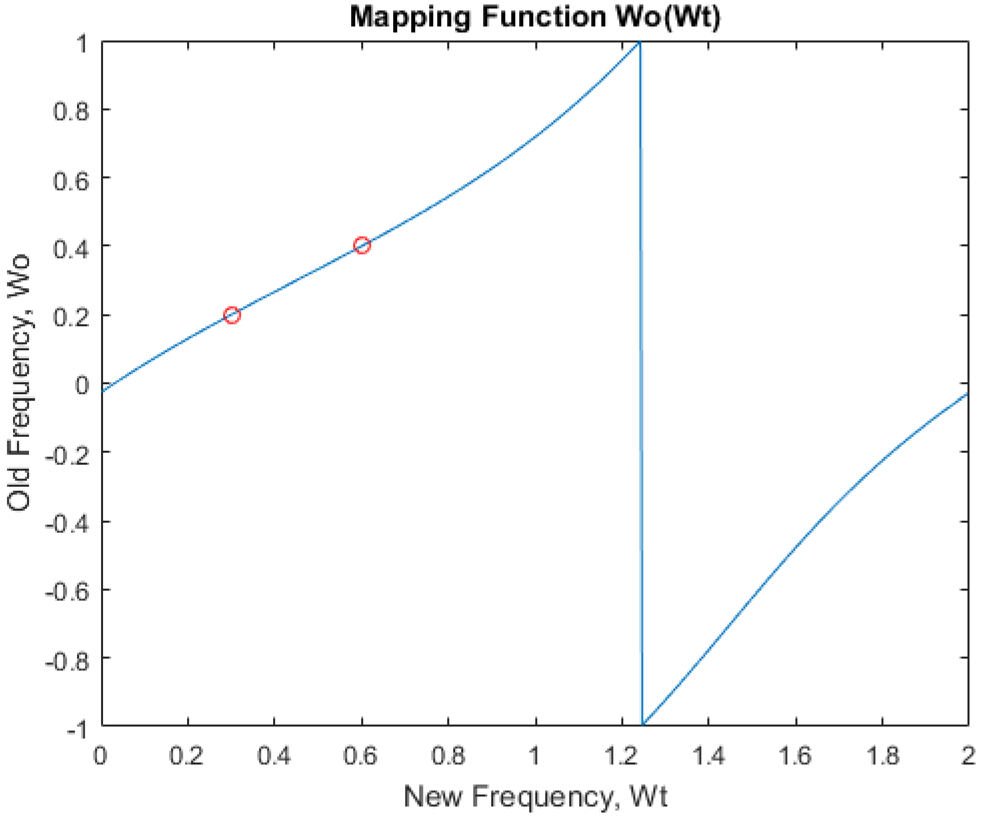
This transformation can also be used for transforming other types of filters; e.g., complex notch filters or resonators can be repositioned at two distinct desired frequencies at any place around the unit circle. This is very attractive for adaptive systems.

Examples

Design of the allpass mapping filter

This example shows how to design allpass mapping filter, changing the complex bandpass filter with the band edges at $W_{o1} = 0.2$ and $W_{o2} = 0.4$ to the new band edges of $W_{t1} = 0.3$ and $W_{t2} = 0.6$. Find the frequency response of the allpass mapping filter:

```
Wo = [0.2, 0.4]; Wt = [0.3, 0.6];
[AllpassNum, AllpassDen] = allpassbpc2bpc(Wo, Wt);
[ha, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi,-angle(ha)/pi, Wt, Wo, 'ro'); title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```



Arguments

Variable	Description
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter

Variable	Description
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`iirbpc2bpc` | `zpkbpc2bpc`

Introduced in R2011a

allpasslp2bp

Allpass filter for lowpass to bandpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bp(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2bp(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the second-order allpass mapping filter for performing a real lowpass to real bandpass frequency transformation. This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "DC mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

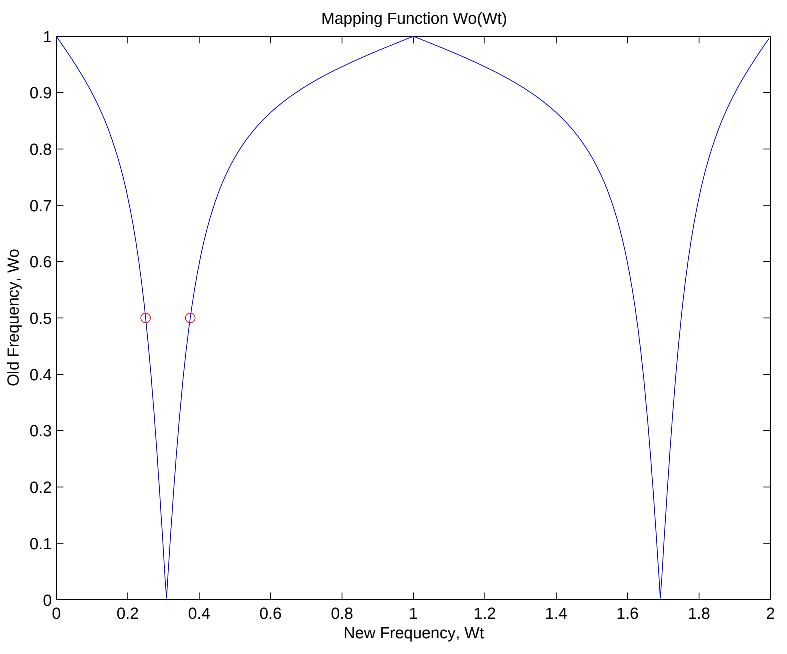
Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and repositioned at two distinct desired frequencies.

Examples

Design the allpass mapping filter changing the lowpass filter with cutoff frequency at W_o=`0.5` to the real–valued bandpass filter with cutoff frequencies at W_{t1}=`0.25` and W_{t2}=`0.375`.

Compute the frequency response and plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
Wo = 0.5; Wt = [0.25, 0.375];
[AllpassNum, AllpassDen] = allpasslp2bp(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```



Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

Constantinides, A.G., "Spectral transformations for digital filters," *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

See Also

`iirlp2bp` | `zpklp2bp`

Introduced in R2011a

allpasslp2bpc

Allpass filter for lowpass to complex bandpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bpc(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2bpc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to complex bandpass frequency transformation. This transformation effectively places one feature of an original filter, located at frequency -W_o, at the required target frequency location, W_{t1}, and the second feature, originally at +W_o, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandpass filters for radio receivers from the high-quality prototype lowpass filter.

Examples

Design the allpass mapping filter changing the real lowpass filter with the cutoff frequency of W_o=`0.5` into a complex bandpass filter with band edges of W_{t1}=`0.2` and

W_{t2}=0.4 precisely defined. Calculate the frequency response of the mapping filter in the full range:

```
Wo = 0.5; Wt = [0.2,0.4];
[AllpassNum, AllpassDen] = allpasslp2bpc(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

iirlp2bpc | zpklp2bpc

Introduced in R2011a

allpasslp2bs

Allpass filter for lowpass to bandstop transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bs(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2bs(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the second-order allpass mapping filter for performing a real lowpass to real bandstop frequency transformation. This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "Nyquist Mobility," which means that the DC feature stays at DC, but the Nyquist feature moves to a location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

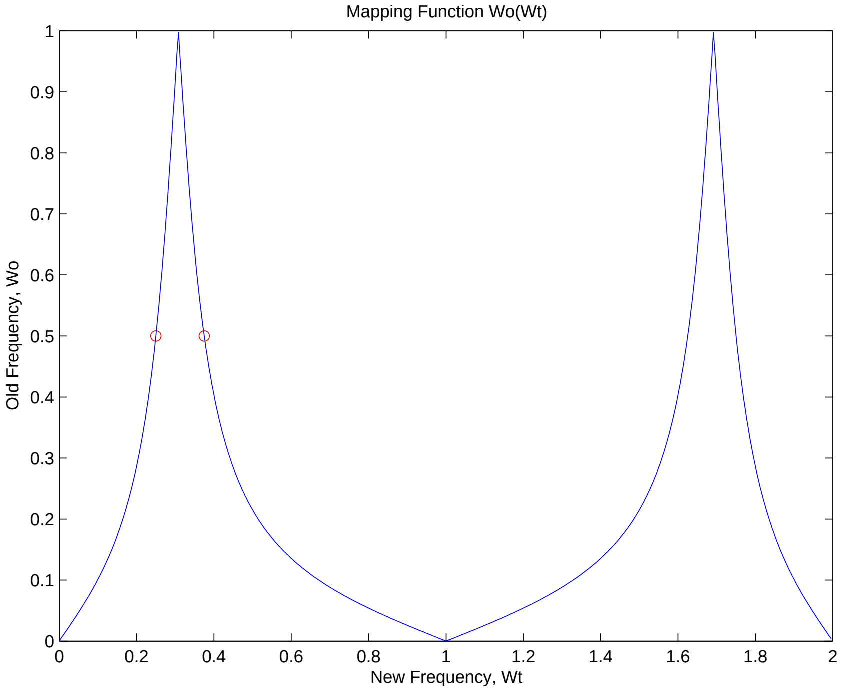
Examples

Design the allpass filter changing the lowpass filter with cutoff frequency at W_o=`0.5` to the real bandstop filter with cutoff frequencies at W_{t1}=`0.25` and W_{t2}=`0.375`:

```
Wo = 0.5; Wt = [0.25, 0.375];
```

```
[AllpassNum, AllpassDen] = allpasslp2bs(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```

In the figure, you find the mapping filter function as determined by the example. Note the response is normalized to π:



Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
AllpassNum	Numerator of the mapping filter

Variable	Description
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

Constantinides, A.G., "Spectral transformations for digital filters," *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

See Also

`iirlp2bs` | `zpklp2bs`

Introduced in R2011a

allpasslp2bsc

Allpass filter for lowpass to complex bandstop transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2bsc(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2bsc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to complex bandstop frequency transformation. This transformation effectively places one feature of an original filter, located at frequency -W_o, at the required target frequency location, W_{t1}, and the second feature, originally at +W_o, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. Additionally the transformation swaps passbands with stopbands in the target filter.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/ resonators. This transformation can be used for designing bandstop filters for band attenuation or frequency equalizers, from the high-quality prototype lowpass filter.

Examples

Design the allpass filter changing the real lowpass filter with the cutoff frequency of W_o=0.5 into a complex bandstop filter with band edges of W_{t1}=0.2 and W_{t2}=0.4 precisely defined:

```
Wo = 0.5; Wt = [0.2,0.4];
[AllpassNum, AllpassDen] = allpasslp2bsc(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

iirlp2bsc | zpklp2bsc

Introduced in R2011a

allpasslp2hp

Allpass filter for lowpass to highpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2hp(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2hp(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to real highpass frequency transformation. This transformation effectively places one feature of an original filter, located originally at frequency, W_o, at the required target frequency location, W_t, at the same time rotating the whole frequency response by half of the sampling frequency. Result is that the DC and Nyquist features swap places.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to highpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband.

Lowpass to highpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way by using the lowpass to highpass transformation.

Examples

Design the allpass filter changing the lowpass filter to the highpass filter with its cutoff frequency moved from $W_o = 0.5$ to $W_t = 0.25$.

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
Wo = 0.5; Wt = 0.25;
[AllpassNum, AllpassDen] = allpasslp2hp(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

Constantinides, A.G., "Spectral transformations for digital filters," *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., “Frequency transformations for digital filters,” *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

See Also

`iirlp2hp` | `zpklp2hp`

Introduced in R2011a

allpasslp2lp

Allpass filter for lowpass to lowpass transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2lp(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2lp(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to real lowpass frequency transformation. This transformation effectively places one feature of an original filter, located originally at frequency W_o, at the required target frequency location, W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

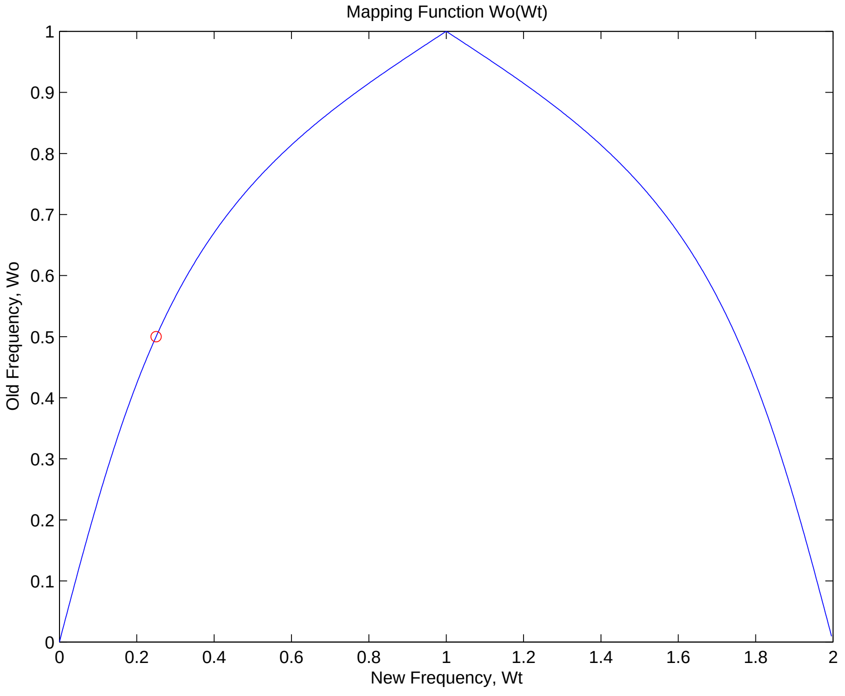
Choice of the feature subject to the lowpass to lowpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband and so on.

Lowpass to lowpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way by applying the lowpass to lowpass transformation.

Examples

Design the allpass filter changing the lowpass filter cutoff frequency originally at W_o=`0.5` to W_t=`0.25`. Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
Wo = 0.5; Wt = 0.25;
[AllpassNum, AllpassDen] = allpasslp2lp(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```



As shown in the figure, `allpasslp2lp` generates a mapping function that converts your prototype lowpass filter to a target lowpass filter with different passband specifications.

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter

Variable	Description
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

Constantinides, A.G., "Spectral transformations for digital filters," *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

See Also

`iirlp2lp` | `zpklp2lp`

Introduced in R2011a

allpasslp2mb

Allpass filter for lowpass to M-band transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt)
[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt,Pass)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `M`th-order allpass mapping filter for performing a real lowpass to real multipassband frequency transformation. Parameter `M` is the number of times an original feature is replicated in the target filter. This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, W_{t1},...,W_{tM}. By default the DC feature is kept at its original location.

`[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility." In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is movable.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

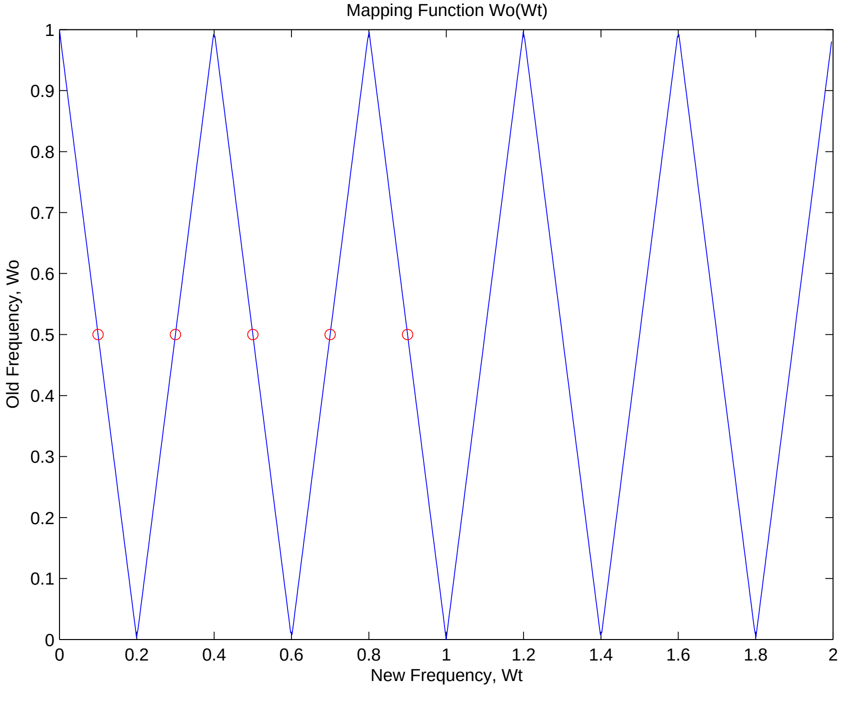
Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations without redesigning them. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design the allpass filter changing the real lowpass filter with the cutoff frequency of W_o=0.5 into a real multiband filter with band edges of W_t=[1:2:9]/10 precisely defined. Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
Wo = 0.5; Wt = [1:2:9]/10;
[AllpassNum, AllpassDen] = allpasslp2mb(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```



Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Pass	Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `'pass'` being the default
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

Franchitti, J.C., “All-pass filter interpolation and frequency transformation problems,“*MSc Thesis*, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.

Feyh, G., J.C. Franchitti and C.T. Mullis, “All-pass filter interpolation and frequency transformation problem,” *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

Mullis, C.T. and R.A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Massachusetts, Addison-Wesley, 1987.

Feyh, G., W.B. Jones and C.T. Mullis, *An extension of the Schur Algorithm for frequency transformations, Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

See Also

`iirlp2mb` | `zpklp2mb`

Introduced in R2011a

allpasslp2mbc

Allpass filter for lowpass to complex M-band transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2mbc(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2mbc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `M`th-order allpass mapping filter for performing a real lowpass to complex multipassband frequency transformation. Parameter `M` is the number of times an original feature is replicated in the target filter. This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, W_{t1},...,W_{tM}.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations without the need to design them again. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design the allpass filter changing the real lowpass filter with the cutoff frequency of W_o=`0.5` into a complex multiband filter with band edges of W_t=`[-3+1:2:9]/10` precisely defined:

```
Wo = 0.5; Wt = [-3+1:2:9]/10;
[AllpassNum, AllpassDen] = allpasslp2mbc(Wo, Wt);
```

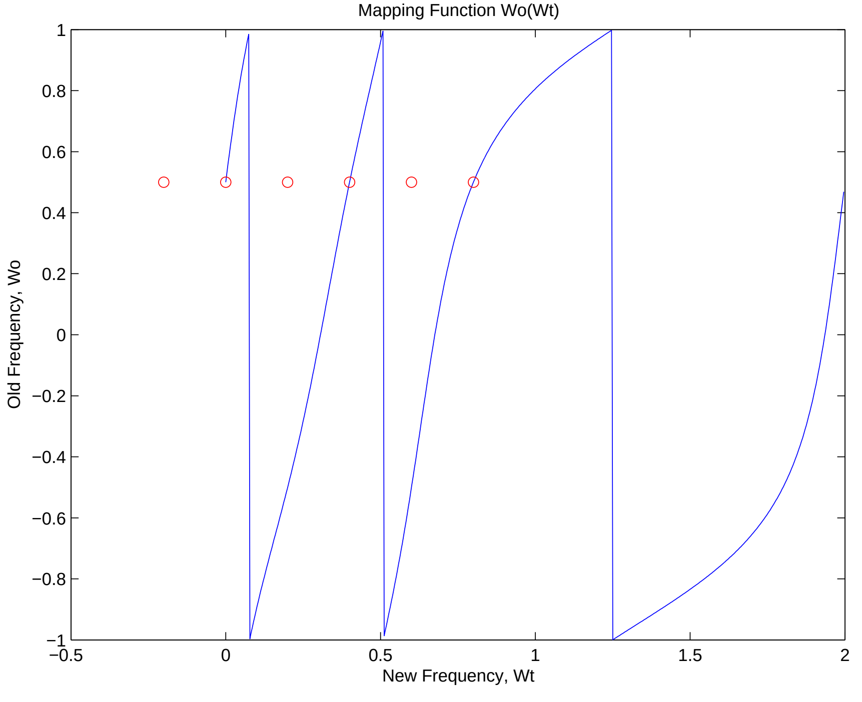
Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, angle(h)/pi, Wt, Wo, 'ro');
 title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```

In this example, the resulting mapping function converts real filters to multiband complex filters.



Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

iirlp2mbc | zpklp2mbc

Introduced in R2011a

allpasslp2xc

Allpass filter for lowpass to complex N-point transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2xc(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2xc(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `N`th-order allpass mapping filter, where `N` is the allpass filter order, for performing a real lowpass to complex multipoint frequency transformation. Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of the, original filter located at frequencies W_{o1},...,W_{oN}, at the required target frequency locations, W_{t1},...,W_{tM}.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design the allpass filter moving four features of an original complex filter given in W_o to the new independent frequency locations W_t. Please note that the transformation creates N replicas of an original filter around the unit circle, where N is the order of the allpass mapping filter:

```
Wo = [-0.2, 0.3, -0.7, 0.4]; Wt = [0.3, 0.5, 0.7, 0.9];
[AllpassNum, AllpassDen] = allpasslp2xc(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Arguments

Variable	Description
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`iirlp2xc` | `zpklp2xc`

Introduced in R2011a

allpasslp2xn

Allpass filter for lowpass to N-point transformation

Syntax

```
[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt)
[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt,Pass)
```

Description

`[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `N`th-order allpass mapping filter, where `N` is the allpass filter order, for performing a real lowpass to real multipoint frequency transformation. Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies W_{o1},...,W_{oN}, at the required target frequency locations, W_{t1},...,W_{tM}. By default the DC feature is kept at its original location.

`[AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility." In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is movable.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations without the need of designing them again. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Arguments

Variable	Description
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Pass	Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `'pass'` being the default
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

Cain, G.D., A. Krukowski and I. Kale, “High Order Transformations for Flexible IIR Filter Design,” *VII European Signal Processing Conference (EUSIPCO'94)*, vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

Krukowski, A., G.D. Cain and I. Kale, “Custom designed high-order frequency transformations for IIR filters,” *38th Midwest Symposium on Circuits and Systems (MWSCAS'95)*, Rio de Janeiro, Brazil, August 1995.

See Also

`iirlp2xn` | `zpklp2xn`

Introduced in R2011a

allpassrateup

Allpass filter for integer upsample transformation

Syntax

```
[AllpassNum,AllpassDen] = allpassrateup(N)
```

Description

`[AllpassNum,AllpassDen] = allpassrateup(N)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the `N`th-order allpass mapping filter for performing the rateup frequency transformation, which creates `N` equal replicas of the prototype filter frequency response.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Examples

Design the allpass filter creating the effect of upsampling the digital filter four times:

Choose any feature from an original filter, say at W_o=`0.2`:

```
N = 4;
Wo = 0.2; Wt = Wo/N + 2*[0:N-1]/N;
[AllpassNum, AllpassDen] = allpassrateup(N);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Arguments

Variable	Description
N	Frequency replication ratio (upsampling ratio)

Variable	Description
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

iirrateup | zpkrateup

Introduced in R2011a

allpassshift

Allpass filter for real shift transformation

Syntax

```
[AllpassNum,AllpassDen] = allpassshift(Wo,Wt)
```

Description

`[AllpassNum,AllpassDen] = allpassshift(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the second-order allpass mapping filter for performing a real frequency shift transformation. This transformation places one selected feature of an original filter, located at frequency W_o, at the required target frequency location, W_t. This transformation implements the "DC mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the real shift transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be moved to a different frequency by applying a shift transformation. In such a way you can avoid designing the filter from the beginning.

Examples

Design the allpass filter precisely shifting one feature of the lowpass filter originally at W_o=`0.5` to the new frequencies of W_t=`0.25`:

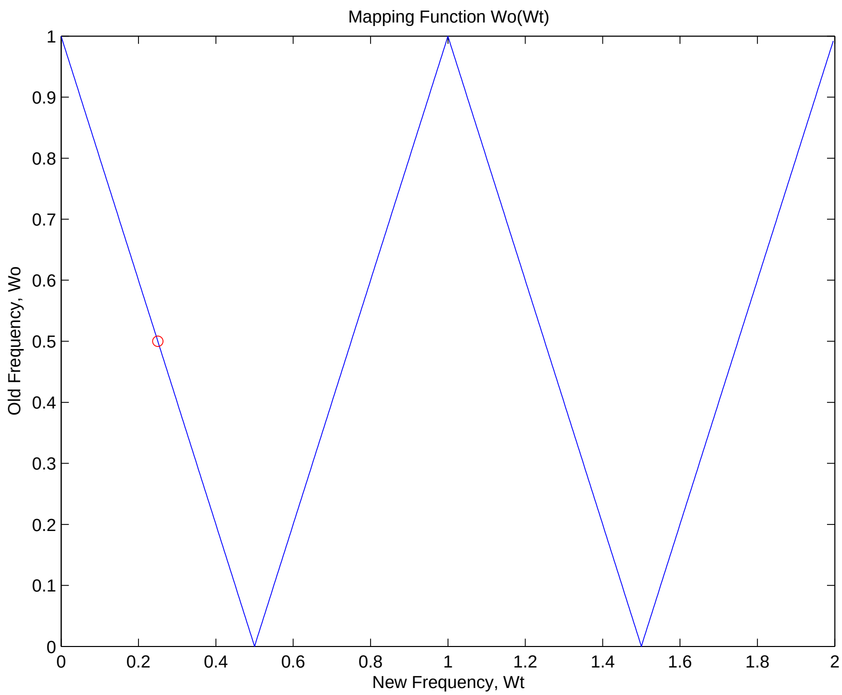
```
Wo = 0.5; Wt = 0.25;
[AllpassNum, AllpassDen] = allpassshift(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function `Wo(Wt)`. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```



Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter

Variable	Description
Wt	Desired frequency location in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirshift` | `zpkshift`

Introduced in R2011a

allpassshiftc

Allpass filter for complex shift transformation

Syntax

```
[AllpassNum,AllpassDen] = allpassshiftc(Wo,Wt)
[AllpassNum,AllpassDen] = allpassshiftc(0,0.5)
[AllpassNum,AllpassDen] = allpassshiftc(0,-0.5)
```

Description

`[AllpassNum,AllpassDen] = allpassshiftc(Wo,Wt)` returns the numerator, `AllpassNum`, and denominator, `AllpassDen`, vectors of the allpass mapping filter for performing a complex frequency shift of the frequency response of the digital filter by an arbitrary amount.

`[AllpassNum,AllpassDen] = allpassshiftc(0,0.5)` calculates the allpass filter for doing the Hilbert transformation, a 90 degree counterclockwise rotation of an original filter in the frequency domain.

`[AllpassNum,AllpassDen] = allpassshiftc(0,-0.5)` calculates the allpass filter for doing an inverse Hilbert transformation, i.e. a 90 degree clockwise rotation of an original filter in the frequency domain.

Examples

Design the allpass filter precisely rotating the whole filter by the amount defined by the location of the selected feature from an original filter, W_o=`0.5`, and its required position in the target filter, W_t=`0.25`:

```
Wo = 0.5; Wt = 0.25;
[AllpassNum, AllpassDen] = allpassshiftc(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

References

Oppenheim, A.V., R.W. Schafer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

Dutta-Roy, S.C. and B. Kumar, "On Digital Differentiators, Hilbert Transformers, and Half-band Low-pass Filters," *IEEE Transactions on Education*, vol. 32, pp. 314-318, August 1989.

See Also

`iirshiftc` | `zpkshiftc`

Introduced in R2011a

autoscale

Automatic dynamic range scaling

Syntax

```
autoscale(hd,x)
hnew = autoscale(hd,x)
```

Description

`autoscale(hd,x)` provides dynamic range scaling for each node of the filter `hd`. This method runs signal `x` through `hd` in floating-point to simulate filtering. `autoscale` uses the maximum and minimum data obtained from that simulation at each filter node to set fraction lengths to cover the simulation full range and maximize the precision. Word lengths are not changed during autoscaling.

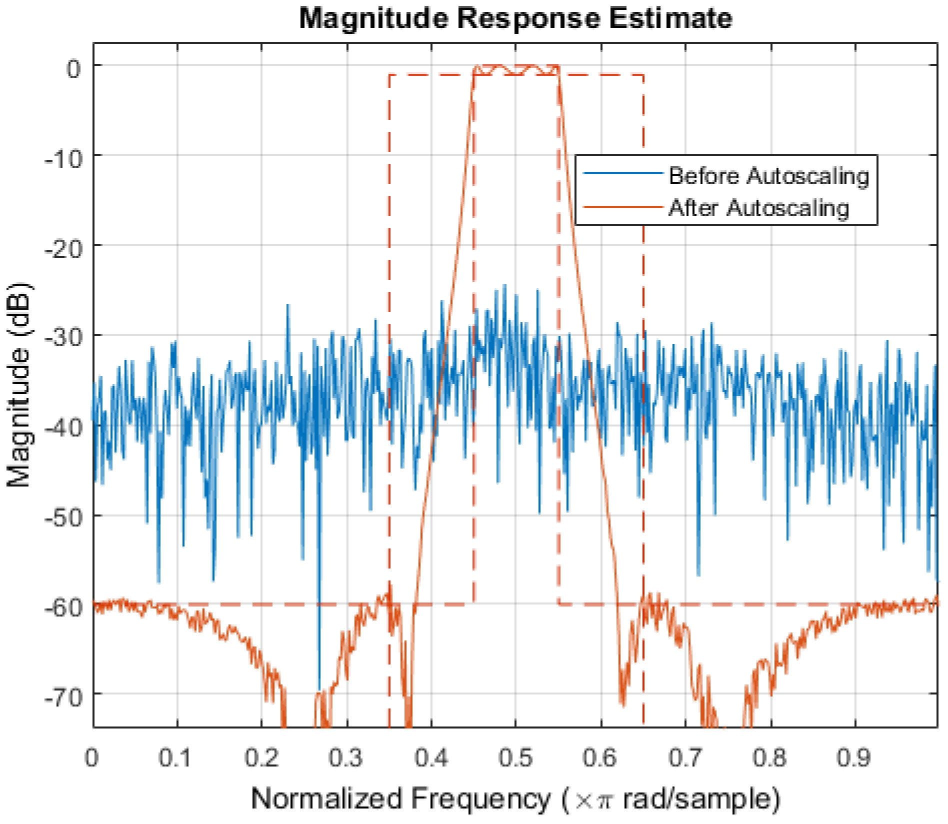
`hnew = autoscale(hd,x)` If you request an output, `autoscale` returns a new filter with the scaled fraction lengths. The original filter is not changed.

For an introductory example of the automatic scale process, see “Floating-Point to Fixed-Point Conversion of IIR Filters”.

Examples

Dynamic Range Scaling in a Lattice ARMA Filter

```
hd = design(fdesign.bandpass,'ellip');
hd = convert(hd,'latticearma');
hd.arithmetic = 'fixed';
rng(4); x = rand(100,10); % Training input data.
hd(2) = autoscale(hd,x);
hfvt = fvtool(hd,'Analysis','magestimate','ShowReference','off');
legend(hfvt, 'Before Autoscaling', 'After Autoscaling')
```



See Also

`qreport`

Introduced in R2011a

block

Generate block from a digital filter

Syntax

```
block(hd)
block(hd,'propertyname1',propertyvalue1,'propertyname2',propertyvalue2,...)
```

Description

`block(hd)` generates a DSP System Toolbox block equivalent to the digital filter, `hd`.

`block(hd,'propertyname1',propertyvalue1,'propertyname2',propertyvalue2,...)` generates a DSP System Toolbox block using the options specified in the property name/ property value pairs. The valid properties and their values are

Property Name	Property Values	Description and Values
`Destination`	`'current'` (default), `'new'`, or *`Subsystemname`*.	Determine which Simulink model gets the block. Enter `'current'`, `'new'`, or specify the name of an existing subsystem with *`subsystemname`*. `'current'` adds the block to your current Simulink model. Specifying `'new'` opens a new model and adds the block. Specifying `'new'` opens a new model and adds the block. If you provide the name of a subsystem in *`subsystemname`*, `block` adds the new block to your specified subsystem.
`Blockname`	`'filter'` (default)	Specify the name of the generated block. The name appears below the block in the model. When you do not specify a block name, the default is `filter`.
`OverwriteBl`	`'off'` (default), or `'on'`.	Tell `block` whether to overwrite an existing block of the same name, or create a new

Property Name	Property Values	Description and Values
		block. `'off'` is the default setting—`block` does not overwrite existing blocks with matching names. Switching from `'off'` to `'on'` directs `block` to overwrite existing blocks.
`MapStates`	`'off'` (default), or `'on'`.	Specify whether to apply the current filter states to the new block. This lets you save states from a filter object you may have used or configured in a specific way. The default setting of `'off'` means the states are not transferred to the block. Choosing `'on'` preserves the current filter states in the block.
`Link2Obj`	`'off'` (default), or `'on'`.	Specify how to set the **Coefficient source** in the block mask. The default setting is `'off'` and the **Coefficient source** is set to **Dialog parameters**. Setting `Link2Obj` to `'on'` sets the **Coefficient source** to **Discrete-time filter object (DFILT)**. The `Link2Obj` and `MapCoeffstoPorts` cannot be simultaneously `'on'`.
`MapCoeffstc`	`'off'` (default) or `'on'`	Specify whether to map the coefficients of the filter to the ports of the block. The `Link2Obj` and `MapCoeffstoPorts` cannot be simultaneously `'on'`.
`CoeffNames`	`{'Num'}` (default FIR) `{'Num','Den'}` (default direct form IIR) `{'Num','Den','g'}` (default IIR SOS), `{'K'}` (default form lattice)	Specify the coefficient variable names as a cell array of character vectors. `MapCoeffsToPorts` must be set to `'on'` for this property to apply.

Property Name	Property Values	Description and Values
InputProces	'columns as channels' (default), 'elements as channels'	Specify sample-based (elements as channels) or frame-based (columns as channels) processing.
RateOption	'enforce single rate' (default) or 'allow multirate'	Specify how the block adjusts the rate at the output to accommodate the reduced number of samples. This parameter applies only when InputProcessing is 'columns as channels'.

Using block to Realize a Fixed-Point Digital Filter

When the source filter hd is fixed-point, the input word and fraction lengths for the block are derived from the block input signal. The realization process issues a warning and ignores the input word and input fraction lengths that are part of the source filter object, choosing to inherit the settings from the input data. Other fixed-point properties map directly to settings for word and fraction length in the realized block.

Examples

Create a Block from a Lowpass Filter

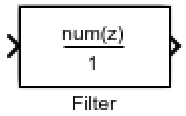
Create a lowpass filter specification object. Specify the passband frequency to be 0.15? rad/sample and the stopband frequency to be 0.25? rad/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB.

In the first example, use block with the default syntax, letting the function determine the block name and configuration.

```
d = fdesign.lowpass('Fp,Fst,Ap,Ast',0.15,0.25,1,60);
hd = design(d);
```

Now use the default syntax to create a block.

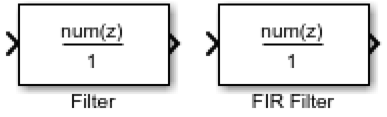
```
block(hd);
```



In this second example, define the block name to meet your needs by using the property name/property value pair input arguments.

```
block(hd, 'blockname', 'FIR Filter');
```

The figure shows the blocks in a Simulink model. When you try these examples, you see that the second block writes over the first block location. You can avoid this by moving the first block before you generate the second, always naming your block with the blockname property, or setting the Destination property to new which puts the filter block in a new Simulink model.



See Also

realizemdl

Topics

"Realize Filters as Simulink Subsystem Blocks"

Introduced in R2011a

butter

Butterworth IIR filter design using specification object

Syntax

```
butterFilter = design(d,'butter','SystemObject',true)
butterFilter =
design(d,'butter',designoption,value,'SystemObject',true)
```

Description

`butterFilter = design(d,'butter','SystemObject',true)` designs a Butterworth IIR digital filter using the specifications supplied in the object `d`.

`butterFilter =`
`design(d,'butter',designoption,value,'SystemObject',true)` returns a Butterworth IIR filter where you specify a design option and value.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `butter`, refer to the command line help system. For example, to get specific information about using `butter` with `d`, the specification object, enter the following at the MATLAB prompt.

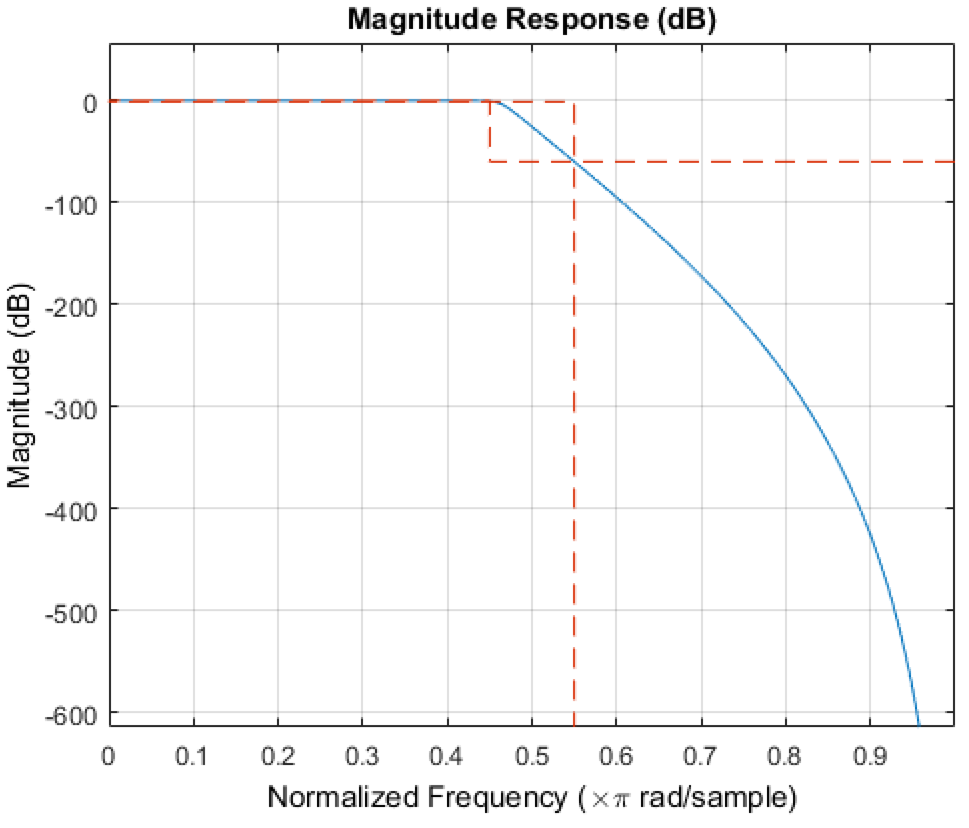
```
help(d,'butter')
```

Examples

Design a Butterworth Filter

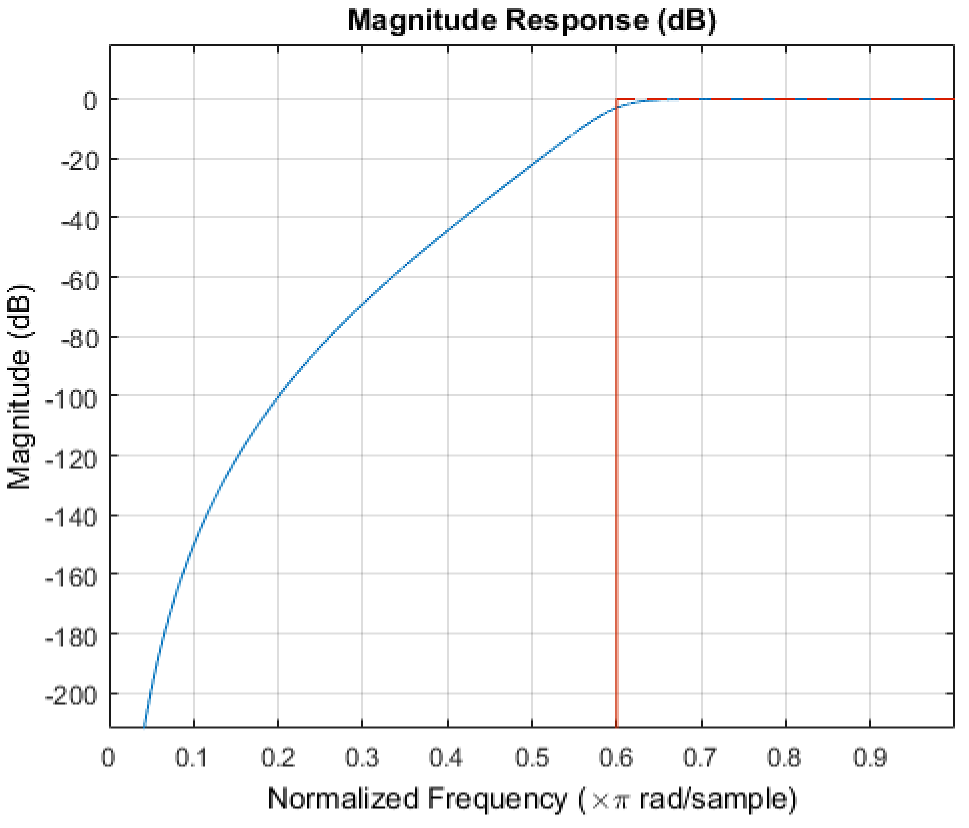
Construct a default lowpass filter specification object and design a Butterworth filter.

```
d = fdesign.lowpass;
Lowpass = design(d,'butter','matchexactly','stopband','SystemObject',true);
fvtool(Lowpass);
```



Construct a highpass filter specification object and design a Butterworth filter. Specify the order to be 8 and 3 dB frequency to be 0.6*pi radians/sample.

```
d = fdesign.highpass('N,F3dB',8,.6);
Highpass = design(d,'butter','SystemObject',true);
fvtool(Highpass)
```



See Also

`cheby1` | `cheby2` | `ellip`

Introduced in R2011a

ca2tf

Convert coupled allpass filter to transfer function form

Syntax

```
[b,a]=ca2tf(d1,d2)
[b,a]=ca2tf(d1,d2,beta)
[b,a,bp]=ca2tf(d1,d2)
[b,a,bp]=ca2tf(d1,d2,beta)
```

Description

`[b,a]=ca2tf(d1,d2)` returns the vector of coefficients `b` and the vector of coefficients `a` corresponding to the numerator and the denominator of the transfer function

$$H(z) = B(z) / A(z) = \frac{1}{2}\left[H1(z) + H2(z)\right]$$

`d1` and `d2` are real vectors corresponding to the denominators of the allpass filters `H1(z)` and `H2(z)`.

`[b,a]=ca2tf(d1,d2,beta)` where `d1`, `d2` and `beta` are complex, returns the vector of coefficients `b` and the vector of coefficients `a` corresponding to the numerator and the denominator of the transfer function

$$H(z) = B(z) / A(z) = \frac{1}{2}\left[-(\bar{\beta}) \cdot H1(z) + \beta \cdot H2(z)\right]$$

`[b,a,bp]=ca2tf(d1,d2)`, where `d1` and `d2` are real, returns the vector `bp` of real coefficients corresponding to the numerator of the power complementary filter G(z)

$$G(z) = Bp(z) / A(z) = \frac{1}{2}\left[H1(z) - H2(z)\right]$$

`[b,a,bp]=ca2tf(d1,d2,beta)`, where `d1`, `d2` and `beta` are complex, returns the vector of coefficients `bp` of real or complex coefficients that correspond to the numerator of the power complementary filter G(z)

$$G(z) = Bp(z) / A(z) = \frac{1}{2_j}\left[-(\bar{\beta}) \cdot H1(z) + \beta \cdot H2(z)\right]$$

Examples

Create a filter, convert the filter to coupled allpass form, and convert the result back to the original structure (create the power complementary filter as well).

```
[b,a]=cheby1(10,.5,.4);
[d1,d2,beta]=tf2ca(b,a);        % tf2ca returns the denominators
                                of the allpasses
[num,den,numpc]=ca2tf(d1,       % Reconstruct the original filter
d2,beta);                       plus the power complementary one
[h,w,s]=freqz(num,den);
hpc = freqz(numpc,den);
s.plot = 'mag';
s.yunits = 'sq';
freqzplot([h hpc],w,s);         % Plot the mag response of the
                                original filter and the power
                                complementary one
```

Examples

Convert Coupled Allpass to Transfer Function Form

Create a filter, convert the filter to coupled allpass form, and convert the result back to the original structure (create the power complementary filter as well).

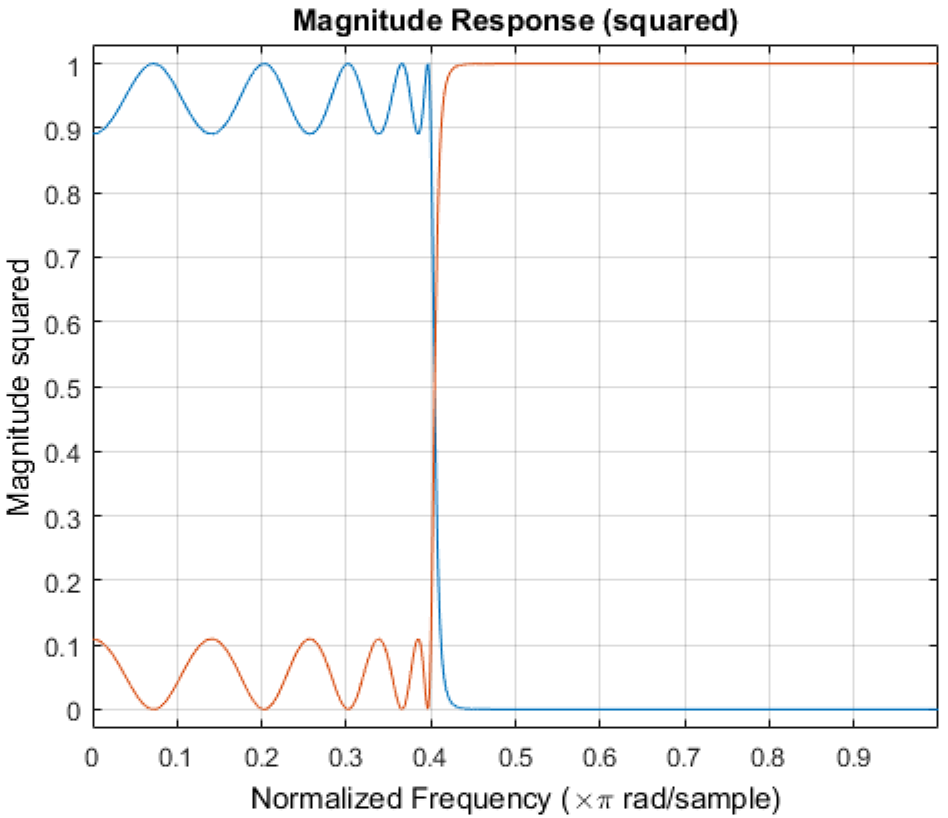
```
[b,a]=cheby1(10,.5,.4);
 [d1,d2,beta]=tf2ca(b,a);
```

tf2ca returns the denominators of the allpasses

```
[num,den,numpc]=ca2tf(d1, d2,beta);
```

Plot the mag response of the original filter and the power complementary one .

```
fvtool(num,den,numpc,den,'Analysis','magnitude','MagnitudeDisplay',...
    'Magnitude Squared')
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

`cl2tf` | `iirpowcomp` | `tf2ca` | `tf2cl`

Introduced in R2011a

cascade

Cascade of filter system objects

Syntax

```
FC = cascade(obj1,obj2,...objn)
```

Description

`FC = cascade(obj1,obj2,...objn)` returns an object, `FC`, of type dsp.FilterCascade. `FC` is a cascaded version of the input System objects `obj1`, `obj2`,`....objn`. You can input multiple System objects to the function. The input System objects must be supported by the cascade method. For the list of supported System objects, see "Input Arguments" on page 5-200.

Examples

Design Two-Stage Decimator

Design a two-stage decimator by cascading `dsp.CICDecimator` and `dsp.CICCompensationDecimator` System objects.

Construct the objects

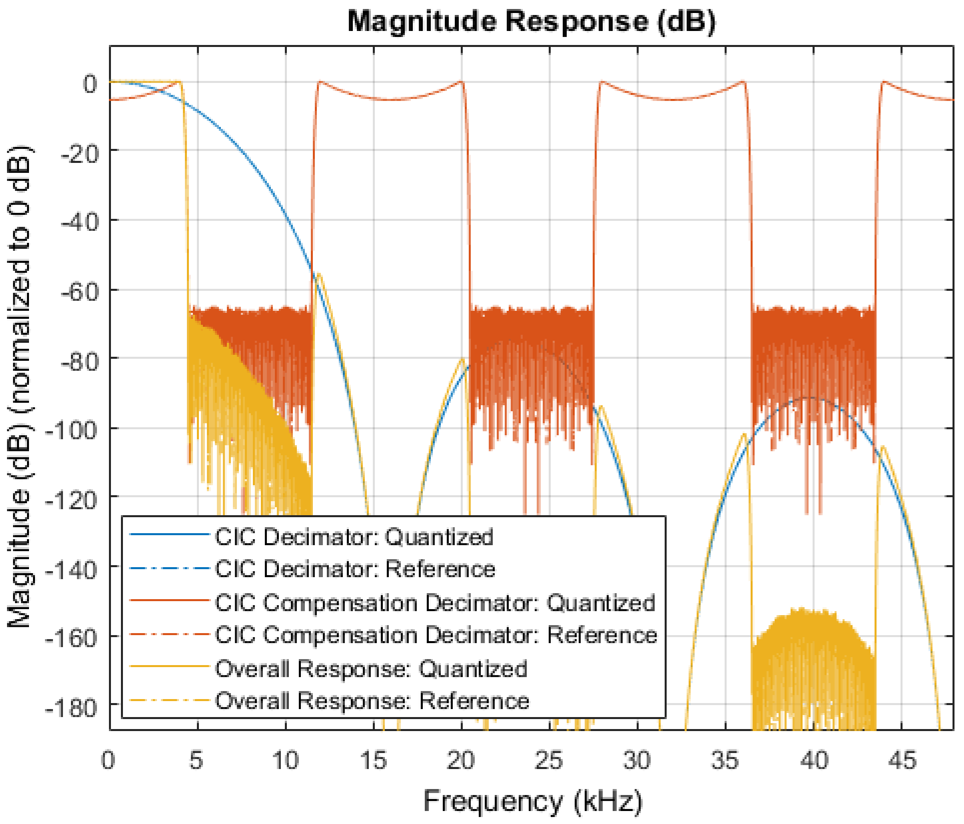
```
CICDecim = dsp.CICDecimator('DecimationFactor', 6, ...
                            'NumSections', 6);
fs = 16e3;     % Sampling frequency of input of compensation decimator
fPass = 4e3;   % Passband frequency
fStop = 4.5e3; % Stopband frequency
CICCompDecim = dsp.CICCompensationDecimator(CICDecim, ...
                              'DecimationFactor', 2, ...
                              'PassbandFrequency', fPass, ...
                              'StopbandFrequency', fStop, ...
                              'SampleRate', fs);
```

Create a cascade of the two objects using the `cascade` method

```
 FC = cascade(CICDecim, CICCompDecim);
```

Visualize the frequency response of the cascade

```
f = fvtool(CICDecim, CICCompDecim, FC, 'Fs', [fs*6, fs, fs*6],...
       'Arithmetic', 'fixed');
set(f, 'NormalizeMagnitudeto1', 'on');
legend(f,'CIC Decimator','CIC Compensation Decimator', ...
            'Overall Response');
```



Input Arguments

`obj1` — Filter to be cascaded
filter System object

`obj1`, `obj2`,`....objn` are filters to be cascaded. To see the list of System objects you can pass to the cascade method, type

`dsp.FilterCascade.helpSupportedSystemObjects`
in the MATLAB command prompt.

Output Arguments

`FC` — cascaded filter

`dsp.FilterCascade` System object

Cascaded filter, returned as a System object of type `dsp.FilterCascade`. For information on the properties of the filter in each stage, type `info(FC)` in the MATLAB command prompt.

See Also

See Also

dsp.FilterCascade

Introduced in R2016a

cheby1

Chebyshev Type I filter using specification object

Syntax

```
chebOneFilter = design(d,'cheby1','SystemObject',true)
chebOneFilter = design(d,'cheby1',designoption,value,designoption,
value,'SystemObject',true)
```

Description

`chebOneFilter = design(d,'cheby1','SystemObject',true)` designs a type I Chebyshev IIR digital filter using the specifications supplied in the object `d`. Depending on the filter specification object, `cheby1` may or may not be a valid design. Use `designmethods` with the filter specification object to determine if a Chebyshev type I filter design is possible.

`chebOneFilter = design(d,'cheby1',designoption,value,designoption, value,'SystemObject',true)` returns a type I Chebyshev IIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `cheby1`, refer to the command line help system. For example, to get specific information about using `cheby1` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'cheby1')
```

Examples

Design a Chebyshev Type I Filter

Construct a default lowpass filter specification object and design a Chebyshev Type I filter.

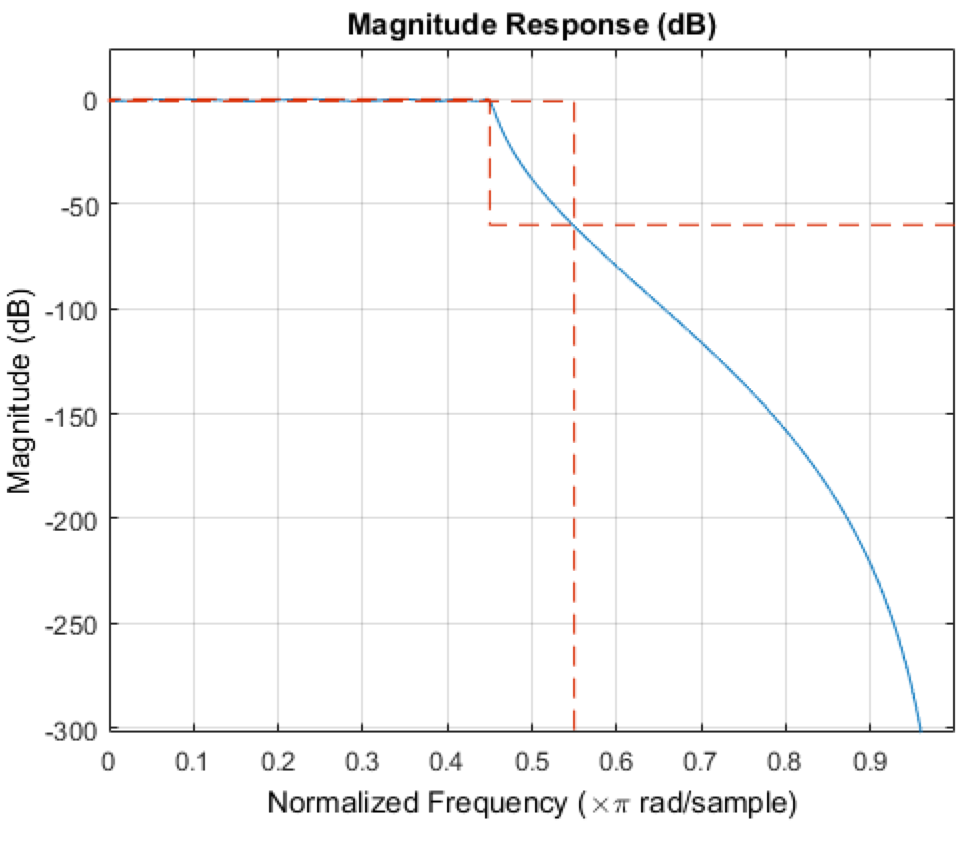
```
d = fdesign.lowpass; designopts(d,'cheby1');
```

Use `matchexactly` option to ensure the performance of the filter in the passband.

```
LowpassCheb1 = design(d,'cheby1','matchexactly','passband',...
    'SystemObject',true);
```

Use `fvtool` to view the resulting filter

```
fvtool(LowpassCheb1)
```

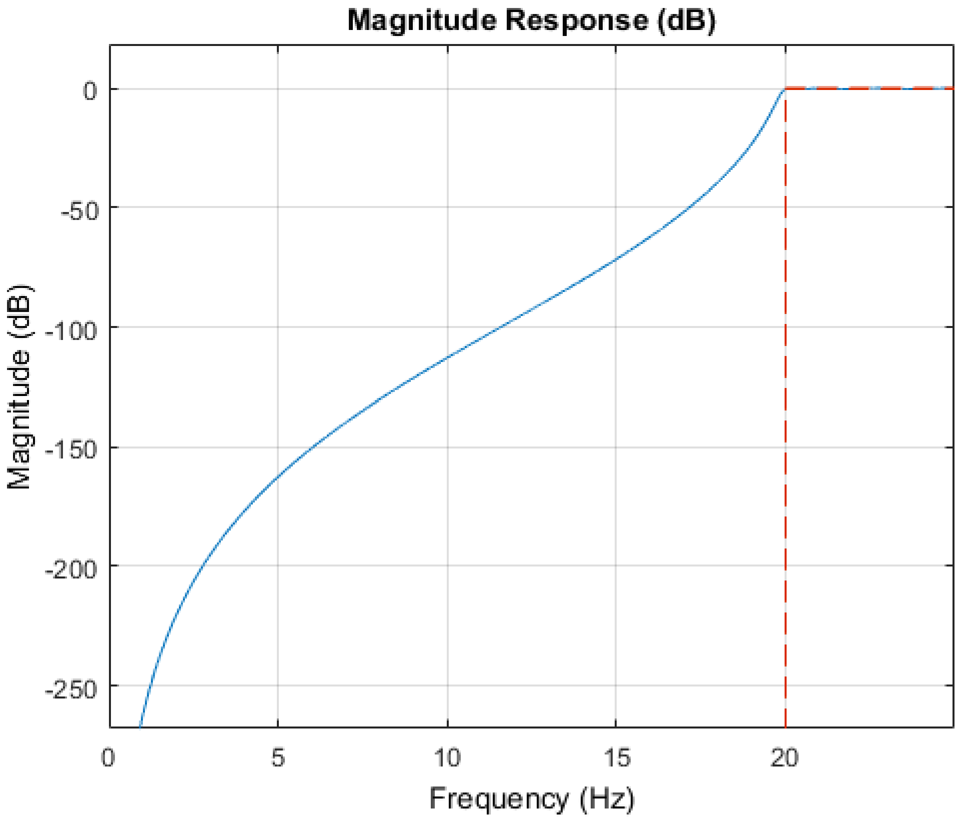


Construct a highpass filter specification object and design a Chebyshev Type I filter. Specify the filter order, passband edge frequency, and the passband ripple to get the filter exactly as required.

```
d = fdesign.highpass('n,fp,ap',7,20,.4,50);
HighpassCheb1 = design(d,'cheby1','SystemObject',true);
```

Use `fvtool` to view the resulting filter

```
fvtool(HighpassCheb1)
```



By design, `cheby1` returns filters that use second-order sections (SOS). For many applications, and for most fixed-point applications, SOS filters are particularly well-suited.

See Also

`design` | `designmethods` | `fdesign`

Introduced in R2011a

cheby2

Chebyshev Type II filter using specification object

Syntax

```
chebTwoFilter= design(d,'cheby2','SystemObject',true)
chebTwoFilter = design(d,'cheby2',designoption,value,designoption,
value,'SystemObject',true)
```

Description

`chebTwoFilter= design(d,'cheby2','SystemObject',true)` designs a Chebyshev II IIR digital filter using the specifications supplied in the object `d`.

`chebTwoFilter = design(d,'cheby2',designoption,value,designoption, value,'SystemObject',true)` returns a Chebyshev II IIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `cheby2`, refer to the command line help system. For example, to get specific information about using `cheby2` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'cheby2')
```

Examples

Design a Chebyshev Type II Filter

Construct a default lowpass filter specification object and design a Chebyshev Type II filter.

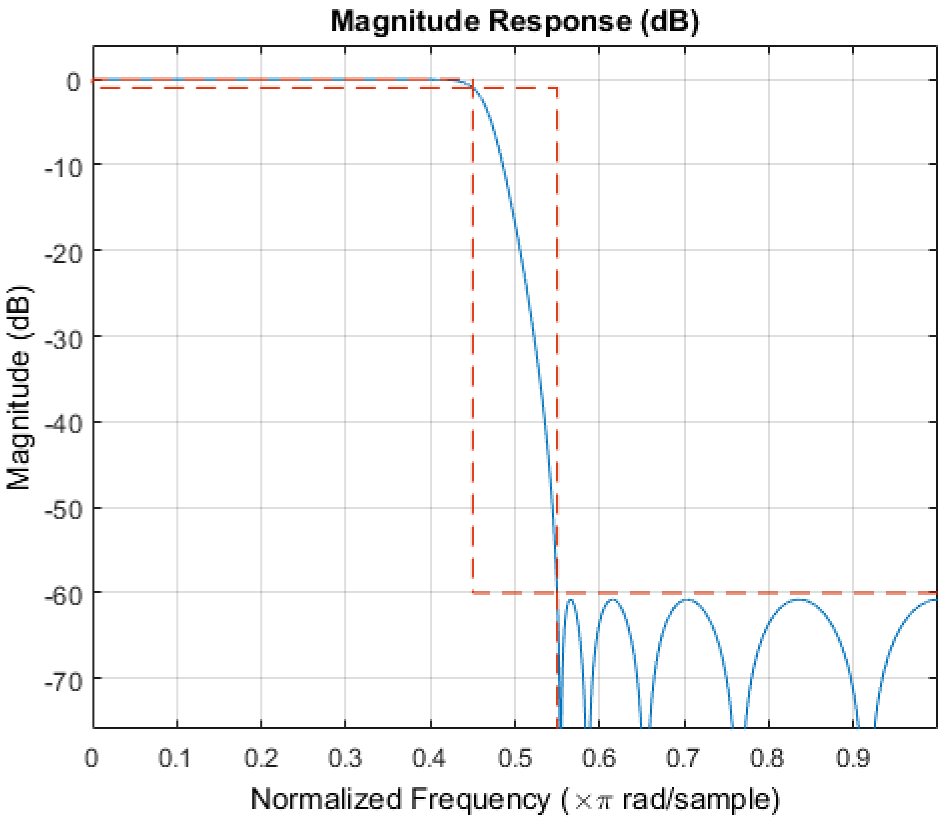
```
d = fdesign.lowpass;
```

Use `matchexactly` option to ensure the performance of the filter in the passband.

```
LowpassCheb2 = design(d,'cheby2','matchexactly','passband',...
    'SystemObject',true);
```

Use `fvtool` to view the resulting filter

```
fvtool(LowpassCheb2)
```



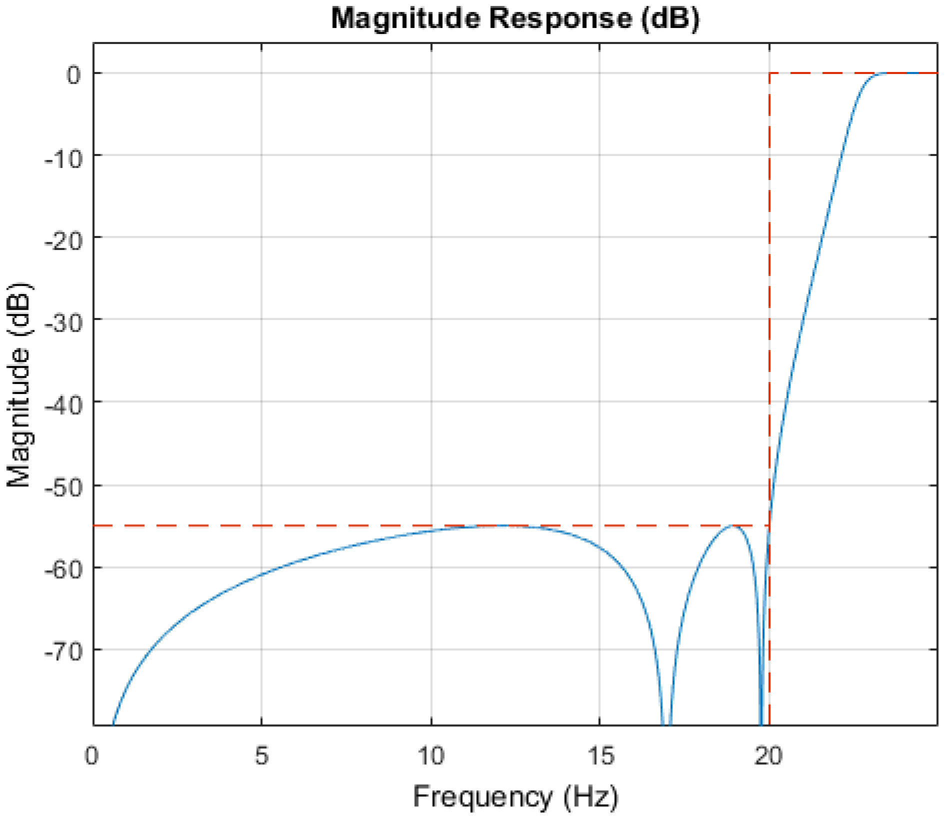
Construct a highpass filter specification object and design a Chebyshev Type II filter. Specify the filter order, stopband edge frequency, and the stopband attenuation to get the filter exactly as required.

```
d = fdesign.highpass('n,fst,ast',5,20,55,50);
```

```
HighpassCheb2 = design(d,'cheby2','SystemObject',true);
```

Use `fvtool` to view the resulting filter

```
fvtool(HighpassCheb2)
```



By design, `cheby2` returns filters that use second-order sections (SOS). For many applications, and for most fixed-point applications, SOS filters are particularly well-suited.

See Also

`butter` | `cheby1` | `ellip`

Introduced in R2011a

cl2tf

Convert coupled allpass lattice to transfer function form

Syntax

```
[b,a] = cl2tf(k1,k2)
[b,a] = cl2tf(k1,k2,beta)
[b,a,bp] = cl2tf(k1,k2)
[b,a,bp] = cl2tf(k1,k2,beta)
```

Description

`[b,a] = cl2tf(k1,k2)` returns the numerator and denominator vectors of coefficients `b` and `a` corresponding to the transfer function

$$H(z) = B(z) / A(z) = \frac{1}{2}\left[H1(z) + H2(z)\right]$$

where *H1(z)* and *H2(z)* are the transfer functions of the allpass filters determined by `k1` and `k2`, and `k1` and `k2` are real vectors of reflection coefficients corresponding to allpass lattice structures.

`[b,a] = cl2tf(k1,k2,beta)` where `k1`, `k2` and `beta` are complex, returns the numerator and denominator vectors of coefficients b and a corresponding to the transfer function

$$H(z) = B(z) / A(z) = \frac{1}{2}\left[-(\bar{\beta})\cdot H1(z) + \beta \cdot H2(z)\right]$$

`[b,a,bp] = cl2tf(k1,k2)` where `k1` and `k2` are real, returns the vector `bp` of real coefficients corresponding to the numerator of the power complementary filter *G(z)*

$$G(z) = Bp(z) / A(z) = \frac{1}{2}\left[H1(z) - H2(z)\right]$$

`[b,a,bp] = cl2tf(k1,k2,beta)` where `k1`, `k2` and `beta` are complex, returns the vector of coefficients `bp` of possibly complex coefficients corresponding to the numerator of the power complementary filter *G(z)*

$$G(z) = Bp(z)\,/\,A(z) = \frac{1}{2_j}\left[-(\bar{\beta}) \cdot H1(z) + \beta \cdot H2(z)\right]$$

Examples

Convert Coupled Allpass Lattice to Transfer Function

`tf2cl` returns the reflection coeffs

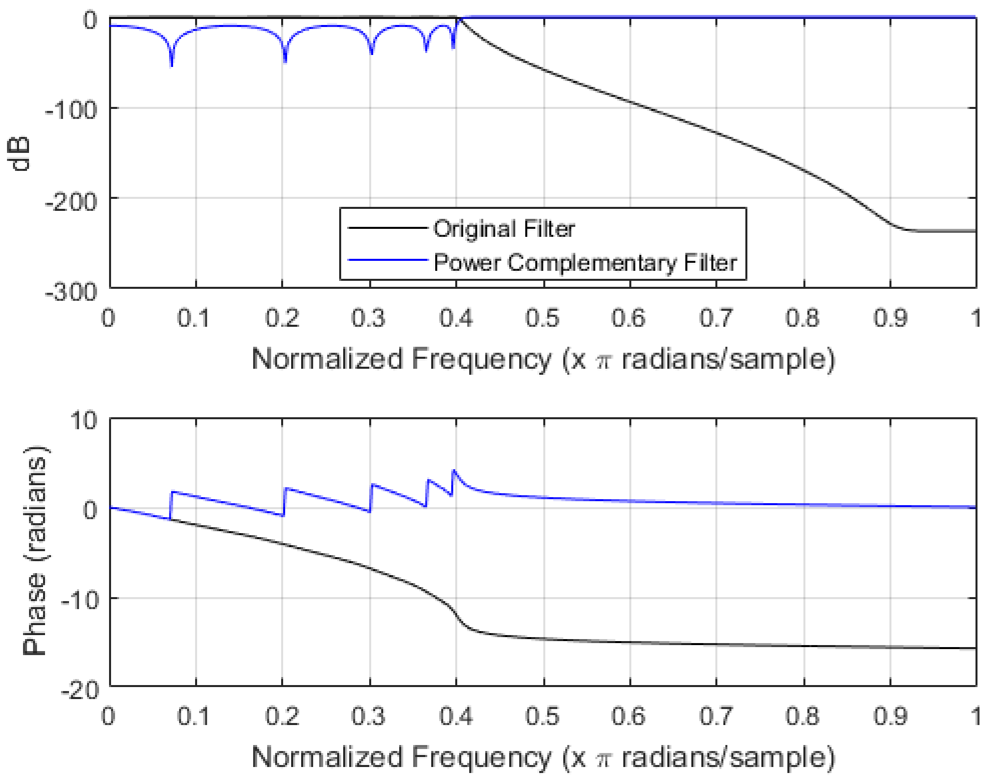
```
[b,a]=cheby1(10,.5,.4);
[k1,k2,beta]=tf2cl(b,a);
```

Construct the original and the power complementary filter.

```
[num,den,numpc]=cl2tf(k1,k2,beta);
[h,w]=freqz(num,den); hpc = freqz(numpc,den);
```

Show the frequency response

```
subplot(211);
plot(w./pi,20*log10(abs(h)),'k'); hold on; grid on;
plot(w./pi,20*log10(abs(hpc)),'b');
xlabel('Normalized Frequency (x \pi radians/sample)');
ylabel('dB');
legend('Original Filter','Power Complementary Filter',...
    'Location','best');
subplot(212);
plot(w./pi,unwrap(angle(h)),'k'); hold on; grid on;
xlabel('Normalized Frequency (x \pi radians/sample)');
ylabel('Phase (radians)');
plot(w./pi,unwrap(angle(hpc)),'b');
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

`tf2cl` | `tf2ca` | `ca2tf` | `tf2latc` | `latc2tf` | `iirpowcomp`

Introduced in R2011a

coeffs

Filter coefficients

Syntax

```
s = coeffs(sysobj)
s = coeffs(sysobj,Name,Value)
```

Description

`s = coeffs(sysobj)` returns the coefficients of filter System object, `sysobj`, in the structure `s`.

`s = coeffs(sysobj,Name,Value)` returns filter coefficients for the filter System object `sysobj` with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`sysobj`

Filter System object.

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpassFilter
dsp.AllpoleFilter
dsp.BiquadFilter
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.CICDecimator
dsp.CICInterpolator
dsp.CoupledAllpassFilter

Filter System objects
dsp.FarrowRateConverter
dsp.FilterCascade
dsp.FIRDecimator
dsp.FIRFilter
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.FIRInterpolator
dsp.FIRRateConverter
dsp.HighpassFilter
dsp.IIRFilter
dsp.IIRHalfbandDecimator
dsp.IIRHalfbandInterpolator
dsp.LowpassFilter
dsp.NotchPeakFilter
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Arithmetic'` — The types of values:
`‘double’ | 'single' | 'fixed'`

Specify the arithmetic used during analysis. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The `coeffs` method returns the quantized filter coefficients when the arithmetic is set to `'single'` or `'fixed'`. When you specify `'fixed'`, the arithmetic changes depend on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type.

Output Arguments

`s`

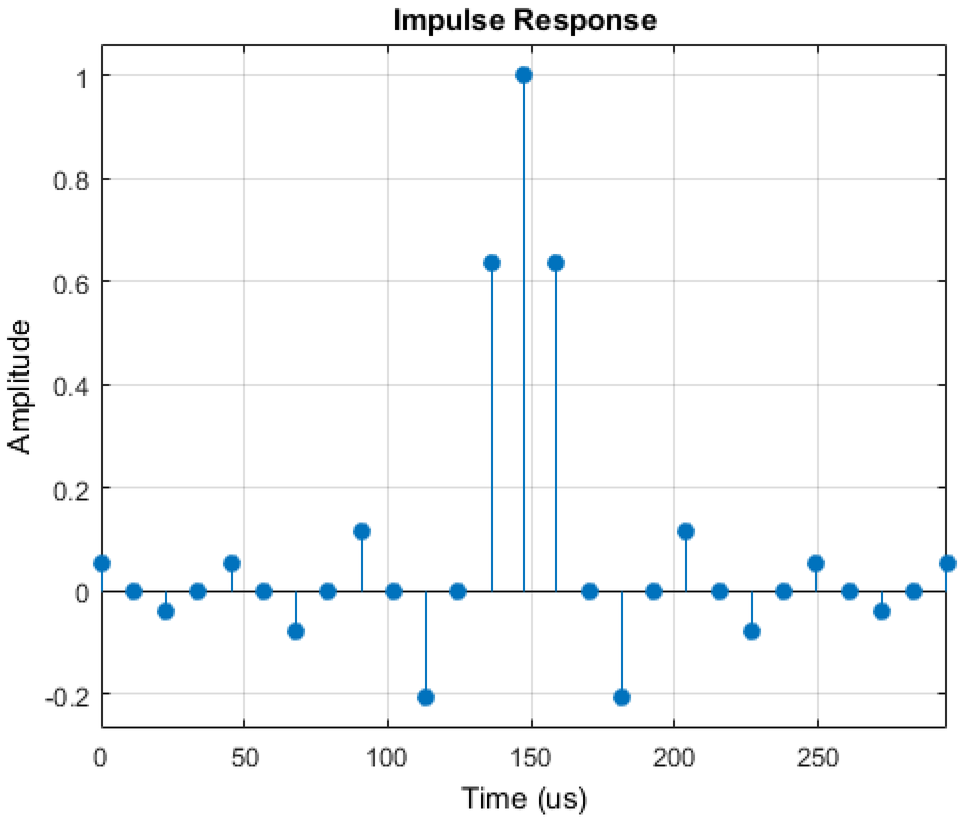
Structure with a single field, `Numerator`, containing filter coefficients.

Examples

Coefficients of an FIR Halfband Interpolator

```
FIRHalfbandInterp = dsp.FIRHalfbandInterpolator('Specification',...
    'Filter order and transition width','FilterOrder',26);
C = coeffs(FIRHalfbandInterp);
C.Numerator

% Impulse response of the filter
fvtool(FIRHalfbandInterp,'impulse')
```

```
ans =

  Columns 1 through 7

    0.0525         0   -0.0379         0    0.0537         0   -0.0771

  Columns 8 through 14

         0    0.1172         0   -0.2060         0    0.6345    1.0000

  Columns 15 through 21

    0.6345         0   -0.2060         0    0.1172         0   -0.0771

  Columns 22 through 27

         0    0.0537         0   -0.0379         0    0.0525
```



See Also

See Also

Functions
`freqz` | `grpdelay` | `impz` | `info` | `phasez` | `stepz` | `zerophase` | `zplane`

Introduced in R2011a

coeread

Read Xilinx COE file

Syntax

```
hd = coeread(filename)
```

Description

`hd = coeread(filename)` extracts the Distributed Arithmetic FIR filter coefficients defined in the XILINX CORE Generator .COE file specified by `filename`. It returns a `dfilt` object, the fixed-point filter `hd`. If you do not provide the file type extension `.coe` with the `filename`, the function assumes the `.coe` extension.

See Also

`coewrite` | `dfilt` | `dfilt.dffir`

Introduced in R2011a

coewrite

Write Xilinx COE file

Syntax

```
coewrite(hd)
coewrite(hd,radix)
coewrite(...,filename)
```

Description

`coewrite(hd)` writes a XILINX Distributed Arithmetic FIR filter coefficient .COE file which can be loaded into the XILINX CORE Generator. The coefficients are extracted from the fixed-point `dfilt` object `hd`. Your fixed-point filter must be a direct form FIR structure `dfilt` object with one section and whose `Arithmetic` property is set to `fixed`. You cannot export single-precision, double-precision, or floating-point filters as `.coe` files, nor multiple-section filters. To enable you to provide a name for the file, `coewrite` displays a dialog box where you fill in the file name. If you do not specify the name of the output file, the default file name is `untitled.coe`.

`coewrite(hd,radix)` indicates the radix (number base) used to specify the FIR filter coefficients. Valid `radix` values are `2` for binary, `10` for decimal, and `16` for hexadecimal (default).

`coewrite(...,filename)` writes a XILINX.COE file to `filename`. If you omit the file extension, `coewrite` adds the `.coe` extension to the name of the file.

The `coewrite` function always generates the XILINX.COE file in your current folder. To use this function, you must have write permission in your current folder.

Examples

`coewrite` generates an ASCII text file that contains the filter coefficients in a format the XILINX CORE Generator can read and load. In this example, you create a 30th-order fixed-point filter and generate the `.coe` file that includes the filter coefficients as well as associated information about the filter.

```
b = firceqrip(30,0.4,[0.05 0.03]); hq = dfilt.dffir(b);
set(hq,'arithmetic','fixed'); coewrite(hq,10,'mycoefile');
```

The `coewrite` function generates the output file, `mycoefile.coe`, in your current folder. The `.coe` file contains the radix, coefficient width, and filter coefficients. The file reports the filter coefficients in column-major order. The radix, coefficient width, and filter coefficients are the minimum set of data needed in a `.coe` file.

See Also

`coeread` | `dfilt` | `dfilt.dffir`

Introduced in R2011a

constraincoeffwl

Constrain coefficient wordlength

Syntax

```
Hq = constraincoeffwl(Hd,wordlength)
Hq = constraincoeffwl(Hd,wordlength,'Ntrials',N)
Hq = constraincoeffwl(Hd,wordlength,...,'NoiseShaping',NSFlag)
Hq = constraincoeffwl(Hd,wordlength,...,'Apasstol',Apasstol)
Hq = constraincoeffwl(Hd,wordlength,...,'Astoptol',Astoptol)
```

Description

`Hq = constraincoeffwl(Hd,wordlength)` returns a fixed-point filter `Hq` meeting the design specifications of the single-stage or multistage FIR filter object `Hd` with a wordlength of at most `wordlength` bits. For multistage filters, `wordlength` can either be a scalar or vector. If `wordlength` is a scalar, the same wordlength is used for all stages. If `wordlength` is a vector, each stage uses the corresponding element in the vector. The vector length must equal the number of stages. `Hd` must be generated using `fdesign` and `design`. `constraincoeffwl` uses a stochastic noise-shaping procedure by default to minimize the wordlength. To obtain repeatable results on successive function calls, initialize the uniform random number generator `rand`

`Hq = constraincoeffwl(Hd,wordlength,'Ntrials',N)` specifies the number of Monte Carlo trials to use. `Hq` is first filter among the trials to meet the specifications in `Hd` with a wordlength of at most `wordlength`.

`Hq = constraincoeffwl(Hd,wordlength,...,'NoiseShaping',NSFlag)` enables or disables the stochastic noise-shaping procedure in the constraint of the wordlength. By default `NSFlag` is `true`. Setting `NSFlag` to `false` constrains the wordlength without using noise-shaping.

`Hq = constraincoeffwl(Hd,wordlength,...,'Apasstol',Apasstol)` specifies the passband ripple tolerance in dB. `'Apasstol'` defaults to `1e-4`.

`Hq = constraincoeffwl(Hd,wordlength,...,'Astoptol',Astoptol)` specifies the stopband tolerance in dB. `'Astoptol'` defaults to `1e-2`

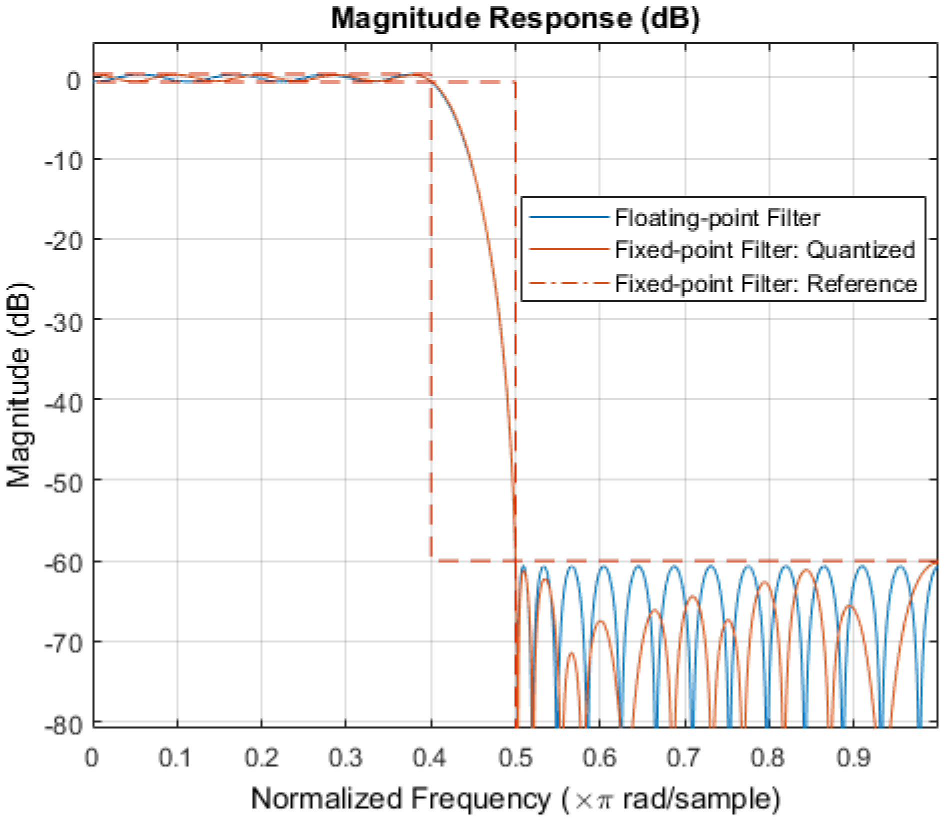
You must have the Fixed-Point Designer software installed to use this function.

Examples

Design Fixed-Point Filter

Design fixed-point filter with a wordlength of at most 11 bits using `constraincoeffwl`

```
Hf = fdesign.lowpass('Fp,Fst,Ap,Ast',.4,.5,1,60);
Hd = design(Hf,'equiripple'); % 43 coefficients
Hq = constraincoeffwl(Hd,11); % 45 11-bit coefficients
hfvt = fvtool(Hd,Hq);
legend(hfvt,'Floating-point Filter','Fixed-point Filter');
```



Tutorials

- "Fixed-Point Data Types"

See Also

`design` | `fdesign` | `maximizestopband` | `minimizecoeffwl` | `measure` | `rand`

Topics

"Fixed-Point Data Types"

Introduced in R2011a

convert

Convert filter structure of discrete-time filter

Syntax

```
hq = convert(hq,newstruct)
```

Description

Discrete-Time Filters

`hq = convert(hq,newstruct)` returns a quantized filter whose structure has been transformed to the filter structure specified by `newstruct`. You can enter any one of the following quantized filter structures:

- `'antisymmetricfir'`: Antisymmetric finite impulse response (FIR)
- `'df1'`: Direct form I
- `'df1t'`: Direct form I transposed
- `'df1sos'`: Direct-Form I, Second-Order Sections
- `'df1tsos'`: Direct-Form I Transposed, Second-Order Sections
- `'df2'`: Direct form II
- `'df2t'`: Direct form II transposed. Default filter structure
- `'df2sos'`: Direct-Form II, Second-Order Sections
- `'df2tsos'`: Direct-Form II Transposed, Second-Order Sections
- `'dffir'`: FIR
- `'dffirt'`: Direct form FIR transposed
- `'latcallpass'`: Lattice allpass
- `'latticeca'`: Lattice coupled-allpass
- `'latticecapc'`: Lattice coupled-allpass power-complementary
- `'latticear'`: Lattice autoregressive (AR)
- `'latticemamax'`: Lattice moving average (MA) maximum phase

- `'latticemamin'`: Lattice moving average (MA) minimum phase
- `'latticearma'`: Lattice ARMA
- `'statespace'`: Single-input/single-output state-space
- `'symmetricfir'`: Symmetric FIR. Even and odd forms

All filters can be converted to the following structures:

- `'df1'`: Direct form I
- `'df1t'`: Direct form I transposed
- `'df1sos'`: Direct-Form I, Second-Order Sections
- `'df1tsos'`: Direct-Form I Transposed, Second-Order Sections
- `'df2'`: Direct form II
- `'df2t'`: Direct form II transposed. Default filter structure
- `'df2sos'`: Direct-Form II, Second-Order Sections
- `'df2tsos'`: Direct-Form II Transposed, Second-Order Sections
- `'statespace'`: Single-input/single-output state-space
- `'symmetricfir'`: Symmetric FIR. Even and odd forms

For the following filter classes, you can specify other conversions as well:

- Minimum phase FIR filters can be converted to `latticemamin`
- Maximum phase FIR filters can be converted to `latticemamax`
- Allpass filters can be converted to `latcallpass`

`convert` generates an error when you specify a conversion that is not possible.

Examples

Convert Direct-Form II Transposed Structure to Direct-Form I

```
[b,a]=ellip(5,3,40,.7); hq = dfilt.df2t(b,a)
hq2 = convert(hq,'df1')
```

```
hq =
```

```
     FilterStructure: 'Direct-Form II Transposed'
          Arithmetic: 'double'
           Numerator: [1x6 double]
         Denominator: [1x6 double]
    PersistentMemory: false


hq2 =

     FilterStructure: 'Direct-Form I'
          Arithmetic: 'double'
           Numerator: [1x6 double]
         Denominator: [1x6 double]
    PersistentMemory: false
```

See Also

mfilt | dfilt

Introduced in R2011a

cost

Estimate cost for implementing filter System objects

Syntax

```
c = cost(sysobj)
c = cost(sysobj,Name,Value)
```

Description

`c = cost(sysobj)` returns a cost estimate `c` for the filter System object `sysobj`.

`c = cost(sysobj,Name,Value)` returns a cost estimate `c` for the filter System object `sysobj` with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`sysobj`

Filter System object.

All filter system objects support analysis methods. For more information on analysis methods, see “Analysis Methods for Filter System Objects” on page 3-2. For a list of analysis methods supported by a particular filter System object, for example `dsp.FIRHalfbandDecimator`, enter `dsp.FIRHalfbandDecimator.helpFilterAnalysis` at the MATLAB Command prompt.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Arithmetic'` — Value types:
`‘double’ | 'single' | 'fixed'`

Specify the arithmetic used during analysis. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

c

Cost estimate, `c` contains the following fields:

Estimated Value	Description
`NumCoefficients`	Number of filter coefficients (excluding coefficients with values 0, 1 or -1)
`NumStates`	Number of filter states
`MultiplicationsPerInputSample`	Number of multiplication operations performed for each input sample
`AdditionsPerInputSample`	Number of addition operations performed for each input sample

Examples

Cost of FIR Filter

This example shows how to compute the cost of implementing an FIR Filter created using `dsp.FIRFilter` object.

```
Fs = 8000; Fcutoff = 2000;
firFilt = dsp.FIRFilter('Numerator', fir1(130,Fcutoff/(Fs/2)));
cost(firFilt)
```

```
ans =

  struct with fields:

                  NumCoefficients: 131
                        NumStates: 130
    MultiplicationsPerInputSample: 131
          AdditionsPerInputSample: 130
```

See Also

See Also

Functions
`coeffs`

Introduced in R2011a

cumsec

Cumulative second-order section of `BiquadFilter` System object

Syntax

```
sect = cumsec(biquad)
sections = cumsec(biquad,indices)
sections = cumsec(biquad,indices,secondary)
cumsec(biquad,...)
sections = cumsec(biquad,Name,Value)
```

Description

`sect = cumsec(biquad)` returns a cell array, `sect`, which contains cumulative sections of the `dsp.BiquadFilter` filter System object, `biquad`. Each element in `sect` is a filter with the structure of the original filter. The first element is the first filter section of `biquad`. The second element of `sect` is a filter that represents the combination of the first and second sections of `biquad`. The third element of `sections` is a filter which combines sections 1, 2, and 3 of `biquad`. This pattern continues until the final element of `sections` contains all the sections of `biquad` and should be identical to `biquad`.

`sections = cumsec(biquad,indices)` returns the cumulative sections of the `dsp.BiquadFilter` filter System object `biquad` whose indices in the original filter are in the vector `indices`.

`sections = cumsec(biquad,indices,secondary)` uses the secondary scaling points `secondary` in the sections to determine where the sections should be split when `secondary` is `true`. `secondary` is `false` by default. This option only applies for `dsp.BiquadFilter` objects with `'Direct form II'` and `'Direct form I transposed'` structures. For these structures, the secondary scaling points refer to the location between the recursive and the nonrecursive part, that is the 'middle' of the section.

`cumsec(biquad,...)` plots the magnitude response of the cumulative sections using `fvtool`.

`sections = cumsec(biquad,Name,Value)` returns the cumulative sections of the filter System object `biquad` with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`biquad`

`dsp.BiquadFilter` filter System object with one of the following filter structures:

Structure	Description
`df1sos`	Direct-form I filter object with second-order sections.
`df1tsos`	Direct-Form I transposed filter with second-order sections.
`df2sos`	Direct-form II filter object with second-order sections.
`df2tsos`	Direct-Form II transposed filter with second-order sections.

`indices`

Filter indices. Use `indices` to specify the filter sections `cumsec` uses to compute the cumulative responses.

`secondary`

This option applies only when `biquad` has the `df2sos` and `df1tsos` structures. For these second-order section structures, the secondary scaling points refer to the scaling locations between the recursive and the nonrecursive parts of the section (the "middle" of the section). Argument `secondary` accepts either `true` or `false`. By default, `secondary` is `false`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

'Arithmetic' — Value types:
'double' | 'single' | 'fixed'

Specify the arithmetic used during analysis. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed' , the arithmetic changes depending on the setting of the CoefficientDataType property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

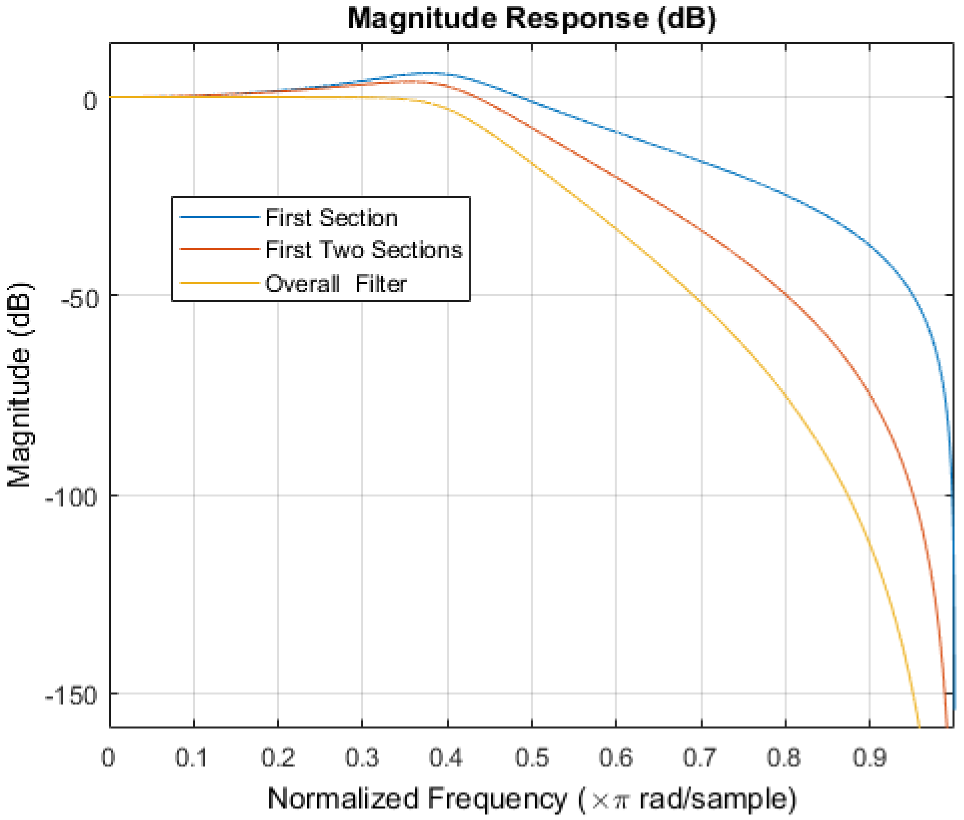
When you do not specify the arithmetic, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type.

Examples

Frequency Response of SOS Filter

This example plots the relative responses of the sections of a sixth-order filter with three sections. Each curve adds one more section to form the filter response.

```
Lowpass = fdesign.lowpass('n,fc',6,.4); ButterLowpass = butter(Lowpass,'SystemObject',t
CumSections = cumsec(ButterLowpass); hfvt = fvtool(CumSections{1},CumSections{2},CumSec
legend(hfvt,'First Section','First Two Sections','Overall  Filter');
```



See Also

`scale` | `scalecheck`

Introduced in R2011a

denormalize

Undo filter coefficient and gain changes caused by normalize

Syntax

```
denormalize(hq)
```

Description

`denormalize(hq)` reverses the coefficient changes you make when you use `normalize` with `hq`. The filter coefficients do not change if you call `denormalize(hq)` before you use `normalize(hq)`. Calling `denormalize` more than once on a filter does not change the coefficients after the first `denormalize` call.

Examples

Denormalize Filter Coefficients

Construct a quantized filter `hd`.

```
d=fdesign.highpass('n,F3dB',14,0.45);
hd =design(d,'butter');
hd.arithmetic='fixed';
```

Normalize the filter coefficients

```
normalize(hd)
NormSOSMatrix = hd.sosMatrix;
```

After normalizing the filter coefficients, restore them to their original values by reversing the effects of the `normalize` function.

```
denormalize(hd)
eqSOSMatrices = isequal(NormSOSMatrix,hd.sosMatrix)
```

```
eqSOSMatrices =
```

```
logical

 0
```

See Also

normalize

Introduced in R2011a

design

Apply design method to filter specification object

Syntax

```
filt = design(D,'Systemobject',true)
filt = design(D,METHOD,'Systemobject',true)
filt =
design(D,METHOD,PARAM1,VALUE1,PARAM2,VALUE2,...,'Systemobject',true)
filt = design(D,METHOD,OPTS,'Systemobject',true)
```

Description

`filt = design(D,'Systemobject',true)` uses the filter specifications object `D` to generate a filter System object, `filt`. When you do not provide a design method as an input argument, `design` uses the default design method. Use `designmethods(D,'default')` to see the default design method for your filter specifications object.

`filt = design(D,METHOD,'Systemobject',true)` uses the design method specified by `METHOD`. `METHOD` must be one of the options returned by `designmethods`. Use `designmethods(D,'default')` to determine which algorithm is used by default.

The design method you provide as the `designmethod` input argument must be one of the methods returned by

```
designmethods(D,'Systemobject',true)
```

To help you design filters more quickly, the input argument `METHOD` accepts a variety of special keywords that force `design` to behave in different ways. The following table presents the keywords you can use for `METHOD` and how `design` responds to the keyword.

Design Method Keyword	Description of the Design Response
`'FIR'`	Forces `design` to produce an FIR filter. When no FIR design method exists for object `D`, `design` returns an error.

Design Method Keyword	Description of the Design Response
`'IIR'`	Forces `design` to produce an IIR filter. When no IIR design method exists for object `D`, `design` returns an error.
`'ALLFIR'`	Produces filters from every applicable FIR design method for the specifications in `D`, one filter for each design method. As a result, `design` returns multiple filters in the output object.
`'ALLIIR'`	Produces filters from every applicable IIR design method for the specifications in `D`, one filter for each design method. As a result, `design` returns multiple filters in the output object.
`'ALL'`	Designs filters using all applicable design methods for the specifications object `D`. As a result, `design` returns multiple filters, one for each design method. `design` uses the design methods in the order that `designmethods(D,'Systemobject',true)` returns them.

Keywords are not case sensitive.

When `design` returns multiple filters in the output object, use indexing to see the individual filters. For example, to see the third filter in `filt`, enter:

```
filt(3)
```

`filt = design(D,METHOD,PARAM1,VALUE1,PARAM2,VALUE2,...,'Systemobject',true)` specifies design-method options. Use `help(D,METHOD)` for complete information on which design-method-specific options are available. You can also use `designopts(D,METHOD)` for a less-detailed listing of the design-method-specific options.

`filt = design(D,METHOD,OPTS,'Systemobject',true)` specifies design-method options using the structure `OPTS`. `OPTS` is usually obtained from `designopts` and then specified as an input to `design`. Use `help(D,METHOD)` for more information on optional inputs.

If you are specifying design-method-specific options using `OPTS`, you can also set `OPTS.SystemObject` to `true` instead of calling `design` with the `'SystemObject', true` syntax.

Examples

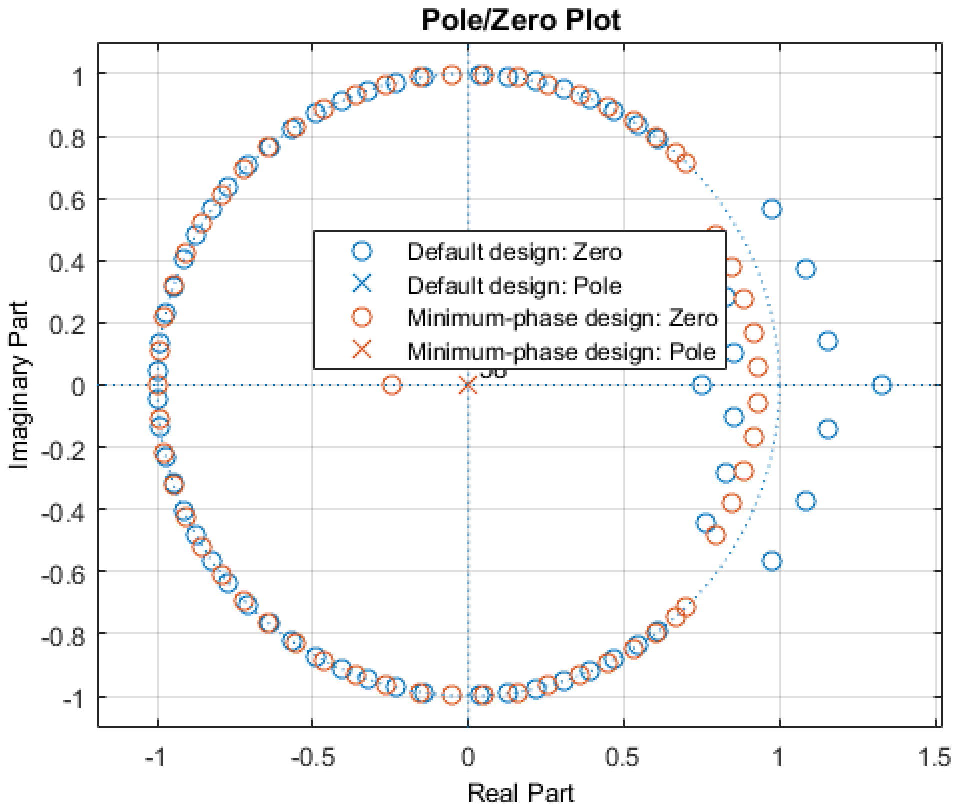
Design of Lowpass Filters

Design an FIR equiripple lowpass filter. Specify a passband edge frequency of 0.2π rad/sample and a stopband edge frequency of 0.25π rad/sample. Set the passband ripple to 0.5 dB and the stopband attenuation to 40 dB. Use the default equiripple method.

```
D = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.25,0.5,40);
filt = design(D,'SystemObject',true);
```

Design a minimum-phase FIR equiripple filter. Display pole-zero plots of the default and minimum-phase designs.

```
filtMin = design(D,'equiripple','MinPhase',true,'SystemObject',true);

fvt = fvtool(filt,filtMin,'Analysis','polezero');
legend(fvt,'Default design','Minimum-phase design')
```



Redesign the filter using the elliptic method. Match the passband exactly.

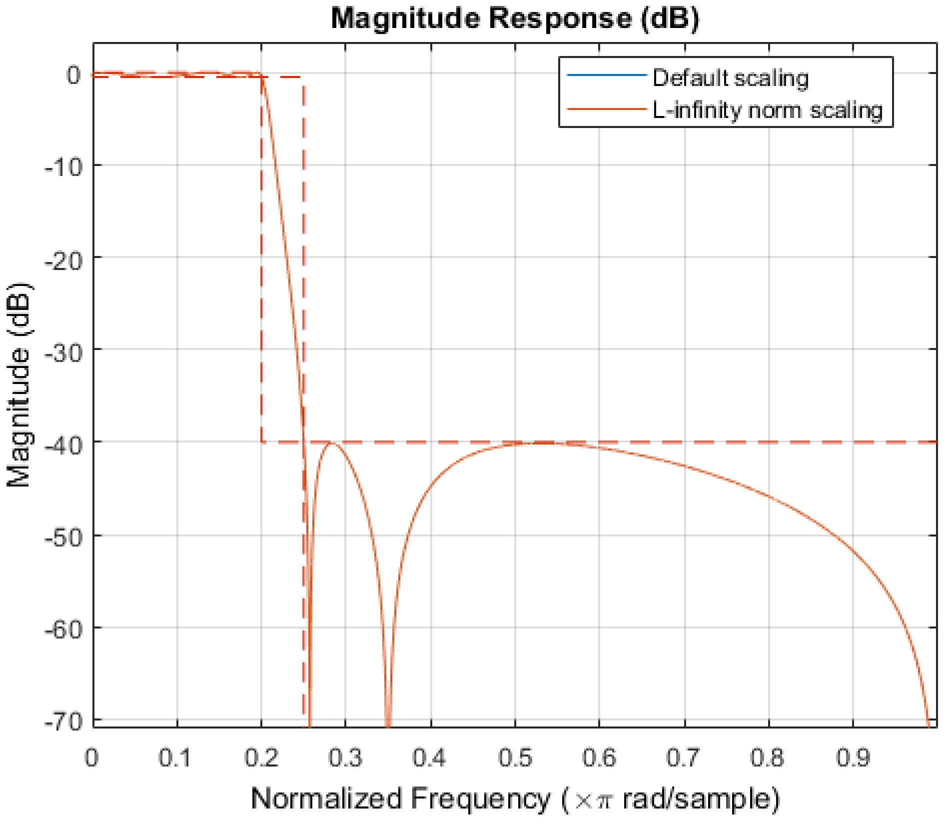
```
filt = design(D,'ellip','MatchExactly','passband','SystemObject',true);
```

You can specify the P-th norm scaling on the second-order sections. Use L-infinity norm scaling in the time domain.

```
filtL = design(D,'ellip','MatchExactly','passband','SOSScaleNorm','linf', ...
    'SystemObject',true);
```

Display the frequency responses of the Butterworth filters.

```
fvt = fvtool(filt,filtL);
legend(fvt,'Default scaling','L-infinity norm scaling')
```



See Also

See Also

`designmethods` | `designopts`

Introduced in R2009a

designMultirateFIR

Multirate FIR filter design

Syntax

`B = designMultirateFIR(L,M)`

`B = designMultirateFIR(L,M,P)`

`B = designMultirateFIR(L,M,P,Astop)`

Description

`B = designMultirateFIR(L,M)` designs a multirate FIR filter with interpolation factor `L` and decimation factor `M`. The output `B` is the vector of designed FIR coefficients. To design a pure interpolator, set *M* to 1. To design a pure decimator, set *L* to 1.

`B = designMultirateFIR(L,M,P)` designs a multirate FIR filter with half-polyphase length `P`. By default, the half-polyphase length is `12`.

`B = designMultirateFIR(L,M,P,Astop)` designs a multirate FIR filter with stopband attenuation `Astop`. By default, the stopband attenuation is 80 dB.

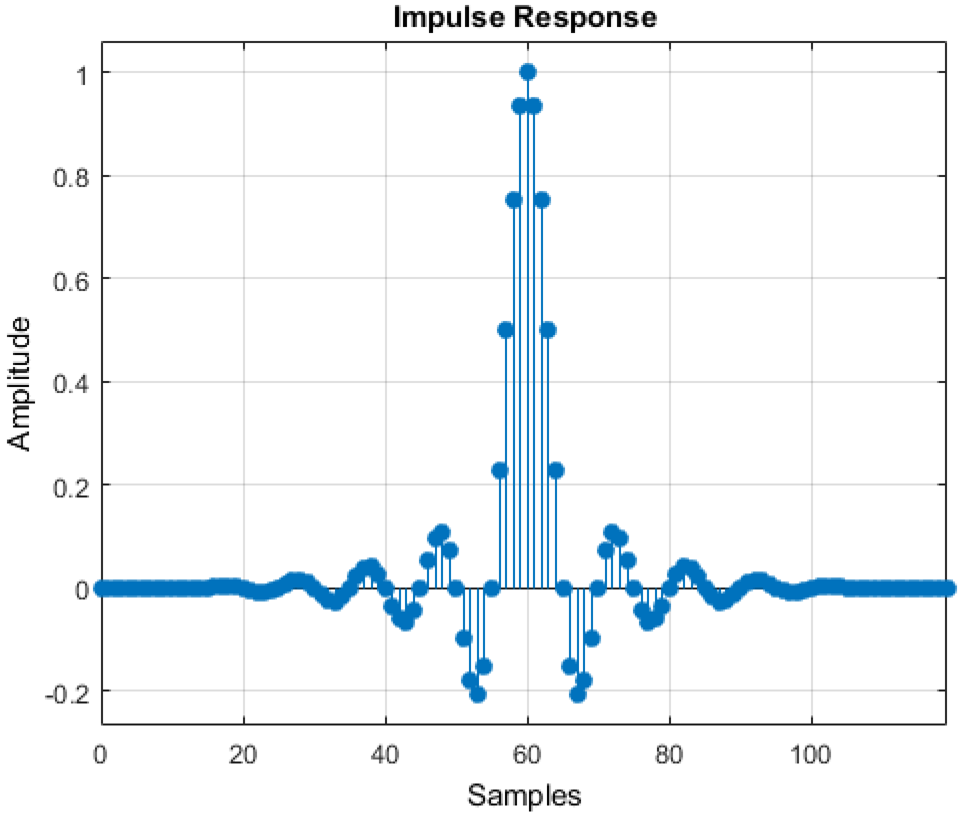
Examples

Design an FIR Interpolator

To design an FIR Interpolator using the `designMultirateFIR` function, you must specify the interpolation factor of interest (usually greater than 1) and a decimation factor equal to 1. You can use the default half-polyphase length of 12 and the default stopband attenuation of 80 dB. Alternately, you can also specify the half-polyphase length and stopband attenuation values.

Design an FIR interpolator with interpolation factor set to 5. Use the default half-polyphase length of 12 and the default stopband attenuation of 80 dB.

```
b = designMultirateFIR(5,1);
fvtool(b,'impulse')
```

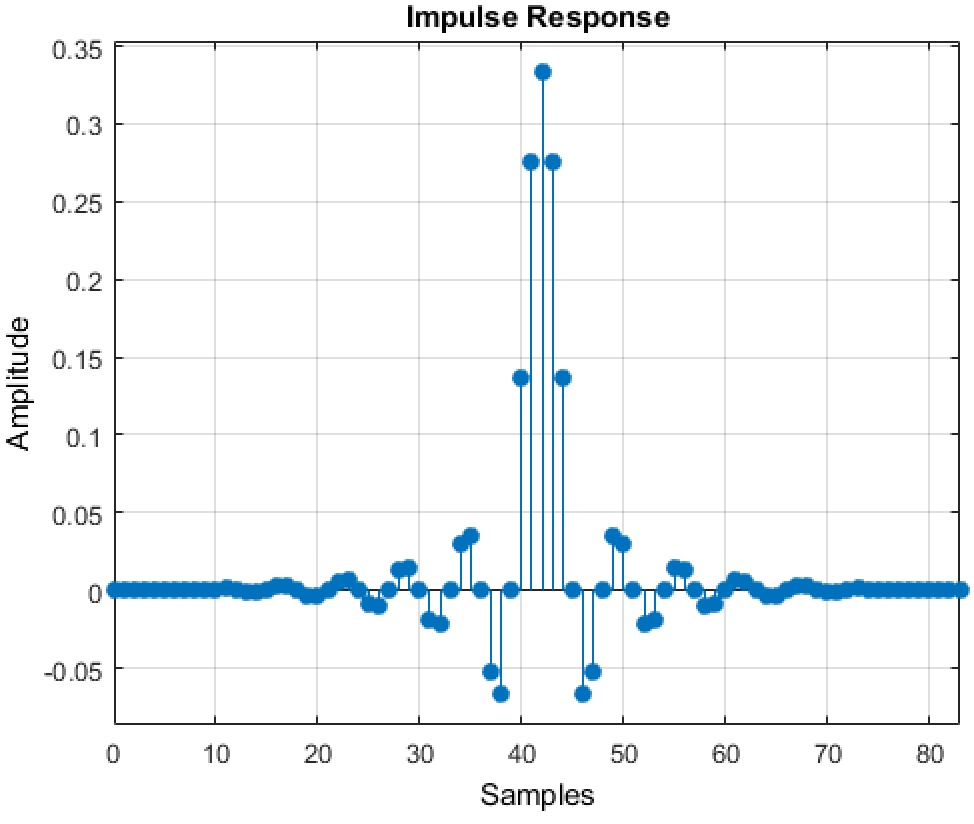


Design an FIR Decimator

To design an FIR Decimator using the `designMultirateFIR` function, you must specify the decimation factor of interest (usually greater than 1) and an interpolation factor equal to 1. You can use the default half-polyphase length of 12 and the default stopband attenuation of 80 dB. Alternately, you can also specify the half-polyphase length and stopband attenuation values. Design an FIR decimator with decimation factor set to 3, and half-polyphase length set to 14. Use the default stopband attenuation of 80 dB.

```
b = designMultirateFIR(1,3,14);
```

```
fvtool(b,'impulse');
```



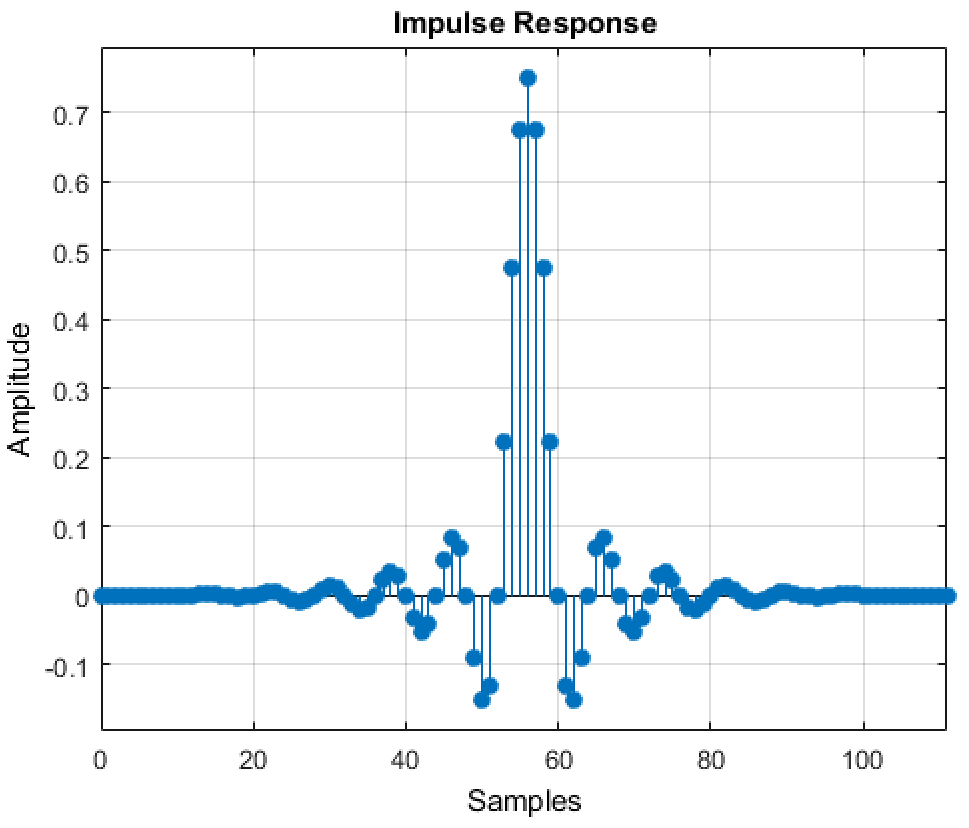
Design an FIR Rate Converter

To design an FIR Rate Comverter using the `designMultirateFIR` function, you must specify an interpolation factor and decimation factor of interest (usually greater than 1). You can use the default half-polyphase length of 12 and the default stopband attenuation of 80 dB. Alternately, you can also specify the half-polyphase length and stopband attenuation values.

Design an FIR rate converter with interpolation factor set to 3, decimation factor set to 4, half-polyphase length set to 14, and stopband attenuation set to 90 dB.

```
b = designMultirateFIR(3,4,14,90);
```

```
fvtool(b,'impulse');
```



Input Arguments

L — Interpolation factor

positive scalar integer

Interpolation factor, specified as a positive scalar integer. To design a decimator only, set L to 1.

Example: 2

Data Types: `single` | `double`

`M` — Decimation factor
positive scalar integer

Decimation factor, specified as a positive scalar integer. To design an interpolator only, set M to `1`.

Example: `2`

Data Types: `single` | `double`

`P` — Half-polyphase length
12 (default) | positive scalar integer

Half-polyphase length, specified as a positive scalar integer.

Example: `12`

Example: `20`

Data Types: `single` | `double`

`Astop` — Stopband attenuation
90 (default) | nonnegative scalar

Stopband attenuation in dB, specified as a nonnegative real scalar greater than or equal to `0`.

Example: `0.0`

Example: `80.5`

Data Types: `single` | `double`

Output Arguments

`B` — Coefficients
real-valued vector

Multirate FIR filter coefficients, returned as a real-valued $N + 1$ length vector, where $N = 2*P*R$, where P is the half-polyphase length, R is defined as max(L,M). For more details, see the “More About” on page 5-249 section.

More About

`designMultirateFIR` designs an Nth order, Rth band Nyquist FIR filter using the N + 1 length Kaiser window vector to window the truncated impulse response of the FIR filter.

Filter order N is defined as $N = 2*P*R$ and R is defined as max(L,M).

The truncated impulse response $d(n)$ is delayed by $N/2$ samples to make it causal. The truncated and delayed impulse response is given by:

$$d(n-N/2)=\frac{\sin(w_c(n-N/2))}{\pi(n-N/2)},\quad n=0,\cdots,\frac{N}{2},\cdots,N$$

where $w_c = \pi/R$.

For every Rth band, the impulse response of the Nyquist filters is exactly zero. Because of this property, when Nyquist filters are used for pure interpolation, the input samples remain unaltered after interpolating.

A Kaiser window is used because of its near-optimum performance while providing a robust way of designing a Nyquist filter. The window depends on two parameters: length $N + 1$ and shape parameter β.

The Kaiser window is defined by:

$$w(n)=\frac{I_0\left(\beta\sqrt{1-\left(\frac{n-N/2}{N/2}\right)^2}\right)}{I_0(\beta)},\qquad 0\le n\le N,$$

where I_0 is the zeroth-order modified Bessel function of the first kind.

The shape parameter β is calculated from:

$$\beta=\begin{cases}0.1102(A_{stop}-8.7) & \text{if } A_{stop}\ge 50\\ 0.5842(A_{stop}-\ 21)^{0.4}+0.07886(A_{stop}-\ 21) & \text{if } 21<A_{stop}<50\\ 0 & \text{if } A_{stop}\le 21,\end{cases}$$

where A_{stop} is the stopband attenuation in dB.

The windowed impulse response is given by

$$h(n) = w(n)d(n - N/2) = w(n)\frac{\sin(w_c(n - N/2))}{\pi(n - N/2)}, \quad n = 0, \cdots, \frac{N}{2}, \cdots, N$$

h(n) for *n = 0,..,N/2,...N* are the coefficients of the multirate filter. These coefficients are defined by the interpolation factor, *L*, and decimation factor, *M*.

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The inputs to the function must be constants.

See Also

See Also

`fdesign.decimator` | `fdesign.halfband` | `fdesign.interpolator` | `firhalfband` | `firnyquist` | `rcosdesign`

Introduced in R2016a

designmethods

Methods available for designing filter from specification object

Syntax

```
M = designmethods(D,'SystemObject',true)
M = designmethods(D,'default')
M = designmethods(D,TYPE,'SystemObject',true)
M = designmethods(D,'full','SystemObject',true)
```

Description

`M = designmethods(D,'SystemObject',true)` returns the available design methods for designing filter System objects for the filter specification object, `D`.

`M = designmethods(D,'default')` returns the default design method for the filter specification object `D`.

`M = designmethods(D,TYPE,'SystemObject',true)` returns the `TYPE` design methods for the filter specification object, `D`. `TYPE` can be either `'FIR'` or `'IIR'`.

`M = designmethods(D,'full','SystemObject',true)` returns the full name for each of the available design methods. For example, `designmethods` with the `'full'` argument returns `Butterworth` for the `butter` method.

Examples

Valid Design Methods for Lowpass Filter

Construct a lowpass filter specification object and determine the valid design methods.

```
D = fdesign.lowpass('Fp,Fst,Ap,Ast',500,600,0.5,60,1e4);
M = designmethods(D,'SystemObject',true)
```

```
M =

  8×1 cell array
```

```
'butter'
'cheby1'
'cheby2'
'ellip'
'equiripple'
'ifir'
'kaiserwin'
'multistage'
```

Obtain detailed command line help on the Chebyshev type I design method.

```
help(D,M{2})
```

```
 DESIGN Design a Chebyshev type I iir filter.
    HD = DESIGN(D, 'cheby1') designs a Chebyshev type I filter specified by the
    FDESIGN object D, and returns the DFILT/MFILT object HD.

    HD = DESIGN(D, ..., 'SystemObject', true) implements the filter, HD,
    using a System object instead of a DFILT/MFILT object.

    HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a filter with the
    structure STRUCTURE.  STRUCTURE is 'df2sos' by default and can be any of
    the following:

    'df1sos'
    'df2sos'
    'df1tsos'
    'df2tsos'
    'cascadeallpass'
    'cascadewdfallpass'

    Some of the listed structures may not be supported by System object
    filters. Type validstructures(D, 'cheby1', 'SystemObject', true) to
    get a list of structures supported by System objects.

    HD = DESIGN(..., 'MatchExactly', MATCH) designs a Chebyshev type I
    filter and matches the frequency and magnitude specification for the
    band MATCH exactly.  The other band will exceed the specification.  MATCH
    can be 'stopband' or 'passband' and is 'passband' by default.

    HD = DESIGN(..., 'SOSScaleNorm', NORM) designs an SOS filter and scales
    the coefficients using the P-Norm NORM. NORM can be either a
    discrete-time-domain norm or a frequency-domain norm. Valid time-domain
```

```
norms are 'l1','l2', and 'linf'. Valid frequency-domain norms are
'L1','L2', and 'Linf'. Note that L2-norm is equal to l2-norm
(Parseval's theorem) but the same is not true for other norms.

The different norms can be ordered in terms of how stringent they are
as follows: 'l1' >= 'Linf' >= 'L2' = 'l2' >= 'L1' >= 'linf'.
Using the most stringent scaling, 'l1', the filter is the least prone
to overflow, but also has the worst signal-to-noise ratio. Linf-scaling is
the most commonly used scaling in practice.

Scaling is turned off by default, which is equivalent to setting
SOSScaleNorm = ''.

HD = DESIGN(..., 'SOSScaleOpts', OPTS) designs an SOS filter and scales
the coefficients using an FDOPTS.SOSSCALING object OPTS.  Scaling options
are:
Property                Default     Description/Valid values
---------               -------     ------------------------
'sosReorder'            'auto'      Reorder section prior to scaling.
                                    {'auto','none','up','down','lowpass',
                                     'highpass','bandpass','bandstop'}
'MaxNumerator'          2           Maximum value for numerator coefficients
'NumeratorConstraint'   'none'      {'none', 'unit', 'normalize','po2'}
'OverflowMode'          'wrap'      {'wrap','saturate'}
'ScaleValueConstraint'  'unit'      {'unit','none','po2'}
'MaxScaleValue'         'Not used'  Maximum value for scale values

When sosReorder is set to 'auto', the sections will be automatically
reordered depending on the response type of the design (lowpass,
highpass, etc.).

Note that 'MaxScaleValue' will only be used when 'ScaleValueConstraint'
is set to something other than 'unit'. If 'MaxScaleValue' is set to a
number, the 'ScaleValueConstraint' will be changed to 'none'. Further,
if SOSScaleNorm is off (as it is by default), then all the SOSScaleOpts
will be ignored.

For more information about P-Norm and scaling options see help for DFILT\SCALE.

% Example #1 - Compare passband and stopband MatchExactly.
   h     = fdesign.lowpass('Fp,Fst,Ap,Ast', .1, .3, 1, 60);
   Hd    = design(h, 'cheby1', 'MatchExactly', 'passband');
   Hd(2) = design(h, 'cheby1', 'MatchExactly', 'stopband');
```

```
% Compare the passband edges in FVTool.
fvtool(Hd);
axis([.09 .11 -2 0]);
```

See Also

See Also

design | designopts | fdesign

Introduced in R2009a

designopts

Valid input arguments and values for specification object and method

Syntax

```
OPTS = designopts(D,METHOD)
```

Description

`OPTS = designopts(D,METHOD)` returns a structure array with the default design parameters used by the design method `METHOD`. `METHOD` must be one of the options returned by `designmethods`.

Use `help(D,METHOD)` to get a description of the design parameters.

If you have DSP System Toolbox software installed, `OPTS` has the `SystemObject` property if at least one of the structures available for that design method is supported by System objects. However, not all structures for that design method are supported by System objects.

Examples

Butterworth Filter Design Options

Create a lowpass filter with a numerator and denominator order of 10 and a 3-dB frequency of 0.2π rad/sample. Obtain the default design parameters for a Butterworth design. Test whether the filter structure is a direct-form II biquad.

```
D = fdesign.lowpass('Nb,Na,F3dB',10,10,0.2);
OPTS = designopts(D,'butter')
if (OPTS.FilterStructure == 'df2sos')
    fprintf('The default filter structure is Direct-Form II\n');
    fprintf('with second-order sections.\n');
end
```

```
OPTS =

  struct with fields:

    FilterStructure: 'df2sos'
       SOSScaleNorm: ''
       SOSScaleOpts: [1×1 fdopts.sosscaling]
       SystemObject: 0

The default filter structure is Direct-Form II
with second-order sections.
```

See Also

See Also

design | designmethods | fdesign | validstructures

Introduced in R2009a

dfilt

Discrete-time filter

Syntax

```
hd = dfilt.structure(input1,...)
hd = [dfilt.structure(input1,...), dfilt.structure(input1,...),...]
hd = design(d,'designmethod')
```

Description

`hd = dfilt.`*`structure`*`(input1,...)` returns a discrete-time filter, `hd`, of type *`structure`*. Each structure takes one or more inputs. When you specify a `dfilt.`*`structure`* with no inputs, a default filter is created.

Note You must use a *`structure`* with `dfilt`.

`hd = [dfilt.`*`structure`*`(input1,...), dfilt.`*`structure`*`(input1,...),...]` returns a vector containing `dfilt` filters.

Structures

Structures for `dfilt.structure` specify the type of filter structure. Available types of structures for `dfilt` are shown below.

dfilt.structure	Description	Coefficient Mapping Support in realizemdl
`dfilt.allpass`	Allpass filter	Supported
`dfilt.cascadeallpas`	Cascade of allpass filter sections	Supported
`dfilt.cascadewdfall`	Cascade of allpass wave digital filters	Supported
`dfilt.delay`	Delay	Not supported
`dfilt.df1`	Direct-form I	Supported

dfilt.structure	Description	Coefficient Mapping Support in realizemdl
`dfilt.df1sos`	Direct-form I, second-order sections	Supported
`dfilt.df1t`	Direct-form I transposed	Supported
`dfilt.df1tsos`	Direct-form I transposed, second-order sections	Supported
`dfilt.df2`	Direct-form II	Supported
`dfilt.df2sos`	Direct-form II, second-order sections	Supported
`dfilt.df2t`	Direct-form II transposed	Supported
`dfilt.df2tsos`	Direct-form II transposed, second-order sections	Supported
`dfilt.dffir`	Direct-form FIR	Supported
`dfilt.dffirt`	Direct-form FIR transposed	Supported
`dfilt.dfsymfir`	Direct-form symmetric FIR	Supported
`dfilt.dfasymfir`	Direct-form antisymmetric FIR	Supported
`dfilt.farrowfd`	Generic fractional delay Farrow filter	Supported
`dfilt.farrowlinearf`	Linear fractional delay Farrow filter	Not supported
`dfilt.fftfir`	Overlap-add FIR	Not supported
`dfilt.latticeallpas`	Lattice allpass	Supported
`dfilt.latticear`	Lattice autoregressive (AR)	Supported
`dfilt.latticearma`	Lattice autoregressive moving-average (ARMA)	Supported
`dfilt.latticemamax`	Lattice moving-average (MA) for maximum phase	Supported
`dfilt.latticemamin`	Lattice moving-average (MA) for minimum phase	Supported
`dfilt.calattice`	Coupled, allpass lattice	Supported

dfilt.structure	Description	Coefficient Mapping Support in realizemdl
`dfilt.calatticepc`	Coupled, allpass lattice with power complementary output	Supported
`dfilt.statespace`	State-space	Supported
`dfilt.scalar`	Scalar gain object	Supported
`dfilt.wdfallpass`	Allpass wave digital filter object	Supported
`dfilt.cascade`	Filters arranged in series	Supported
`dfilt.parallel`	Filters arranged in parallel	Supported

For more information on each structure, refer to its reference page.

`hd = design(d,'designmethod')` returns the `dfilt` object `hd` resulting from the filter specification object `d` and the design method you specify in *`designmethod`*. When you omit the `designmethod` argument, `design` uses the default design method to construct a filter from the object `d`.

With this syntax, you design filters by:

1. Specifying the filter specifications, such as the response shape (perhaps highpass) and details (passband edges and attenuation).
2. Selecting a method (such as `equiripple`) to design the filter.
3. Applying the method to the specifications object with `design(d,'designmethod')`.

Using the specification-based technique can be more effective than the coefficient-based filter design techniques.

Design Methods for Design Syntax

When you use the `hd = design(d,'`*`designmethod`*`')` syntax, you have a range of design methods available depending on `d`, the filter specification object. The next table lists all of the design methods in the toolbox.

Design Method String	Filter Design Result
`butter`	Butterworth IIR

Design Method String	Filter Design Result
cheby1	Chebyshev Type I IIR
cheby2	Chebyshev Type II IIR
ellip	Elliptic IIR
equiripple	Equiripple with the same ripple in the pass and stopbands
firls	Least-squares FIR
freqsamp	Frequency-Sampled FIR
ifir	Interpolated FIR
iirlpnorm	Least Pth norm IIR
iirls	Least-Squares IIR
kaiserwin	Kaiser-windowed FIR
lagrange	Fractional delay filter
multistage	Multistage FIR
window	Windowed FIR

As the specifications object `d` changes, the available methods for designing filters from `d` also change. For instance, if `d` is a lowpass filter with the default specification `'Fp,Fst,Ap,Ast'`, the applicable methods are:

```
% Create an object to design a lowpass filter.
d=fdesign.lowpass;
designmethods(d) % What design methods apply to object d?
```

If you change the specification string to `'N,F3dB'`, the available design methods change:

```
d=fdesign.lowpass('N,F3dB');
designmethods(d)
```

Analysis Methods

Methods provide ways of performing functions directly on your `dfilt` object without having to specify the filter parameters again. You can apply these methods directly on the variable you assigned to your `dfilt` object.

For example, if you create a `dfilt` object, `hd`, you can check whether it has linear phase with `islinphase(hd)`, view its frequency response plot with `fvtool(hd)`, or obtain its frequency response values with `h = freqz(hd)`. You can use all of the methods described here in this way.

Note If your variable `hd` is a 1-D array of `dfilt` filters, the method is applied to each object in the array. Only `freqz`, `grpdelay`, `impz`, `is*`, `order`, and `stepz` methods can be applied to arrays. The `zplane` method can be applied to an array only if `zplane` is used without outputs.

Some of the methods listed here have the same name as functions in Signal Processing Toolbox software. They behave similarly.

Method	Description
`addstage`	Adds a stage to a `cascade` or `parallel` object, where a stage is a separate, modular filter. Refer to `dfilt.cascade` and `dfilt.parallel`.
`block`	`block(hd)` creates a block of the `dfilt` object. The `block` method can specify these properties and values: `'Destination'` indicates where to place the block. `'Current'` places the block in the current Simulink model. `'New'` creates a new model. Default value is `'Current'`. `'Blockname'` assigns the entered string to the block name. Default name is `'Filter'`. `'OverwriteBlock'`indicates whether to overwrite the block generated by the block method (`'on'`) and defined by `Blockame`. Default is `'off'`. `'MapStates'` specifies initial conditions in the block (`'on'`). Default is `'off'`. Refer to "Using Filter States" in Signal Processing Toolbox documentation.
`cascade`	Returns the series combination of two `dfilt` objects. Refer to `dfilt.cascade`.
`coeffs`	Returns the filter coefficients in a structure containing fields that use the same property names as those in the original `dfilt`.

Method	Description
convert	Converts a dfilt object from one filter structure, to another filter structure.
fcfwrite	Writes a filter coefficient ASCII file. The file can contain a single filter or a vector of objects. The default file name is untitled.fcf. fcfwrite(hd,filename) writes to a disk file named filename in the current working folder. The .fcf extension is added automatically. fcfwrite(...,fmt) writes the coefficients in the format fmt, where valid fmt strings are: 'hex' for hexadecimal 'dec' for decimal 'bin' for binary representation
fftcoeffs	Returns the frequency-domain coefficients used when filtering with a dfilt.fftfir
filter	Performs filtering using the dfilt object.
firtype	Returns the type (1-4) of a linear phase FIR filter.
freqz	Plots the frequency response in fvtool. Unlike the freqz function, this dfilt freqz method has a default length of 8192.
grpdelay	Plots the group delay in fvtool.
impz	Plots the impulse response in fvtool.
impzlength	Returns the length of the impulse response.
info	Displays dfilt information, such as filter structure, length, stability, linear phase, and, when appropriate, lattice and ladder length.
isallpass	Returns a logical 1 (i.e., true) if the dfilt object in an allpass filter or a logical 0 (i.e., false) if it is not.

Method	Description
iscascade	Returns a logical 1 if the dfilt object is cascaded or a logical 0 if it is not.
isfir	Returns a logical 1 if the dfilt object has finite impulse response (FIR) or a logical 0 if it does not.
islinphase	Returns a logical 1 if the dfilt object is linear phase or a logical 0 if it is not.
ismaxphase	Returns a logical 1 if the dfilt object is maximum-phase or a logical 0 if it is not.
isminphase	Returns a logical 1 if the dfilt object is minimum-phase or a logical 0 if it is not.
isparallel	Returns a logical 1 if the dfilt object has parallel stages or a logical 0 if it does not.
isreal	Returns a logical 1 if the dfilt object has real-valued coefficients or a logical 0 if it does not.
isscalar	Returns a logical 1 if the dfilt object is a scalar or a logical 0 if it is not scalar.
issos	Returns a logical 1 if the dfilt object has second-order sections or a logical 0 if it does not.
isstable	Returns a logical 1 if the dfilt object is stable or a logical 0 if it are not.
nsections	Returns the number of sections in a second-order sections filter. If a multistage filter contains stages with multiple sections, using nsections returns the total number of sections in all the stages (a stage with a single section returns 1).
nstages	Returns the number of stages of the filter, where a stage is a separate, modular filter.
nstates	Returns the number of states for an object.
order	Returns the filter order. If hd is a single-stage filter, the order is given by the number of delays needed for a minimum realization of the filter. If hd has multiple stages, the order is given by the number of delays needed for a minimum realization of the overall filter.

Method	Description
`parallel`	Returns the parallel combination of two `dfilt` filters. Refer to `dfilt.parallel`.
`phasez`	Plots the phase response in `fvtool`.

Method	Description
`realizemdl`	(Available only with Simulink.) `realizemdl(hd)` creates a Simulink model containing a subsystem block realization of your `dfilt`. `realizemdl(hd,p1,v1,p2,v2,...)` creates the block using the properties `p1`, `p2`,... and values `v1`, `v2`,... specified. The following properties are available: `'Blockname'` specifies the name of the block. The default value is `'Filter'`. `'Destination'` specifies whether to add the block to a current Simulink model or create a new model. Valid values are `'Current'` and `'New'`. `'OverwriteBlock'` specifies whether to overwrite an existing block that was created by `realizemdl` or create a new block. Valid values are `'on'` and `'off'`. Only blocks created by `realizemdl` are overwritten. The following properties optimize the block structure. Specifying `'on'` turns the optimization on and `'off'` creates the block without optimization. The default for each block is `'off'`. `'OptimizeZeros'` removes zero-gain blocks. `'OptimizeOnes'` replaces unity-gain blocks with a direct connection. `'OptimizeNegOnes'` replaces negative unity-gain blocks with a sign change at the nearest summation block. `'OptimizeDelayChains'` replaces delay chains made up of *n* unit delays with a single delay by *n*.

Method	Description
removestage	Removes a stage from a cascade or parallel dfilt. Refer to dfilt.cascade and dfilt.parallel.
setstage	Overwrites a stage of a cascade or parallel dfilt. Refer to dfilt.cascade and dfilt.parallel.
sos	Converts the dfilt to a second-order sections dfilt. If hd has a single section, the returned filter has the same class. sos(hd,flag) specifies the ordering of the second-order sections. If flag='UP', the first row contains the poles closest to the origin, and the last row contains the poles closest to the unit circle. If flag='down', the sections are ordered in the opposite direction. The zeros are always paired with the poles closest to them. sos(hd,flag,scale) specifies the scaling of the gain and the numerator coefficients of all second-order sections. scale can be 'none', 'inf' (infinity-norm), or 'two' (2-norm). Using infinity-norm scaling with up ordering minimizes the probability of overflow in the realization. Using 2-norm scaling with down ordering minimizes the peak roundoff noise.
ss	Converts the dfilt to state-space. To see the separate A,B,C,D matrices for the state-space model, use [A,B,C,D]=ss(hd).
stepz	Plots the step response in fvtool stepz(hd,n) computes the first n samples of the step response. stepz(hd,n,Fs) separates the time samples by T = 1/Fs, where Fs is assumed to be in hertz.
sysobj	Converts the dfilt to a filter System object. See the reference page for a list of supported objects.
tf	Converts the dfilt to a transfer function.
zerophase	Plots the zero-phase response in fvtool.

Method	Description
`zpk`	Converts the `dfilt` to zeros-pole-gain form.
`zplane`	Plots a pole-zero plot in `fvtool`.

Viewing Properties

As with any object, use `get` to view a `dfilt` properties. To see a specific property, use

```
get(hd,'property')
```

To see all properties for an object, use

```
get(hd)
```

Note `dfilt` objects include an `arithmetic` property. You can change the internal arithmetic of the filter from double- precision to single-precision using: `hd.arithmetic = 'single'`.

If you have Fixed-Point Designer software, you can change the `arithmetic` property to fixed-point using: `hd.arithmetic = 'fixed'`

Changing Properties

To set specific properties, use

```
set(hd,'property1',value,'property2',value,...)
```

You must use single quotation marks around the property name. Use single quotation marks around the `value` argument when the value is a string, such as `specifyall` or `fixed`.

Copying an Object

To create a copy of an object, use the `copy` method.

```
h2 = copy(hd)
```

Note Using the syntax `H2 = hd` copies only the object handle and does not create a new, independent object.

Converting Between Filter Structures

To change the filter structure of a dfilt object hd, use

```
hd2 = convert(hd,'structure_string');
```

where structure_string is any valid structure name in single quotation marks. If hd is a cascade or parallel structure, each stage is converted to the new structure.

Using Filter States

Two properties control the filter states:

- states — Stores the current states of the filter. Before the filter is applied, the states correspond to the initial conditions and after the filter is applied, the states correspond to the final conditions. For df1, df1t, df1sos and df1tsos structures, states returns a filtstates object.
- PersistentMemory — Controls whether filter states are saved. The default value is 'false', which causes the initial conditions to be reset to zero before filtering and turns off the display of states information. Setting PersistentMemory to 'true' allows the filter to use your initial conditions or to reuse the final conditions from a previous filtering operation as the initial conditions of the next filtering operation. The true setting also displays information about the filter states.

Note If you set the states and want to use them for filtering, you must set PersistentMemory to 'true' before you use the filter.

Examples

Create a direct-form I filter, and use a method to see if it is stable.

```
[b,a] = butter(8,0.25);
hd = dfilt.df1(b,a);
isstable(hd)
```

If a dfilt's numerator values do not fit on a single line, a description of the vector is displayed. To see the specific numerator values for this example, use

```
B = get(hd,'numerator');
```

```
% or
B1 = hd.numerator;
```

Create an array containing two `dfilt` objects, apply a method and verify that the method acts on both objects. Use a method to test whether the objects are FIR objects.

```
b = fir1(5,.5);
hd = dfilt.dffir(b);         % Create an FIR filter object
[b,a] = butter(5,.5);        % Create IIR filter
hd(2) = dfilt.df2t(b,a);      % Create DF2T object and place
                              % in the second column of hd.
[h,w] = freqz(hd);
test_fir = isfir(hd)
% hd(1) is FIR and hd(2) is not.
```

Refer to the reference pages for each structure for more examples.

See Also

dfilt | design | fdesign | realizemdl | sos | stepz | dfilt.cascade | dfilt.df1 | dfilt.df1t | dfilt.df2 | dfilt.df2t | dfilt.dfasymfir | dfilt.dffir | dfilt.dffirt | dfilt.dfsymfir | dfilt.latticeallpass | dfilt.latticear | dfilt.latticearma | dfilt.latticemamax | dfilt.latticemamin | dfilt.parallel | dfilt.statespace | filter | freqz | grpdelay | impz | zplane

Introduced in R2011a

dfilt.allpass

Allpass filter

Syntax

```
hd = dfilt.allpass(c)
```

Description

`hd = dfilt.allpass(c)` constructs an allpass filter with the minimum number of multipliers from the elements in vector `c`. To be valid, `c` must contain one, two, three, or four real elements. The number of elements in `c` determines the order of the filter. For example, `c` with two elements creates a second-order filter and `c` with four elements creates a fourth-order filter.

The transfer function for the allpass filter is defined by

$$H(z) = \frac{c(n) + c(n-1)z^{-1} + \ldots + z^{-n}}{1 + c(1)z^{-1} + \ldots + c(n)z^{-n}}$$

given the coefficients in `c`.

To construct a cascade of allpass filter objects, use `dfilt.cascadeallpass`. For more information about creating cascades of allpass filters, refer to `dfilt.cascadeallpass`.

Properties

The following table provides a list of all the properties associated with an allpass `dfilt` object.

Property Name	Brief Description
`AllpassCoefficients`	Contains the coefficients for the allpass filter object

Property Name	Brief Description
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.
`States`	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to `filtstates` in Signal Processing Toolbox documentation or in the Help system.

Examples

This example constructs and displays the information about a second-order allpass filter that uses the minimum number of multipliers.

```
c = [1.5, 0.7];
% Create a second-order dfilt object.
hd = dfilt.allpass(c);
```

See Also

`dfilt` | `dfilt.cascadeallpass` | dsp.CICInterpolator | `dfilt.cascadewdfallpass` | `dfilt.latticeallpass` | dsp.IIRHalfbandDecimator

Introduced in R2011a

dfilt.calattice

Coupled-allpass, lattice filter

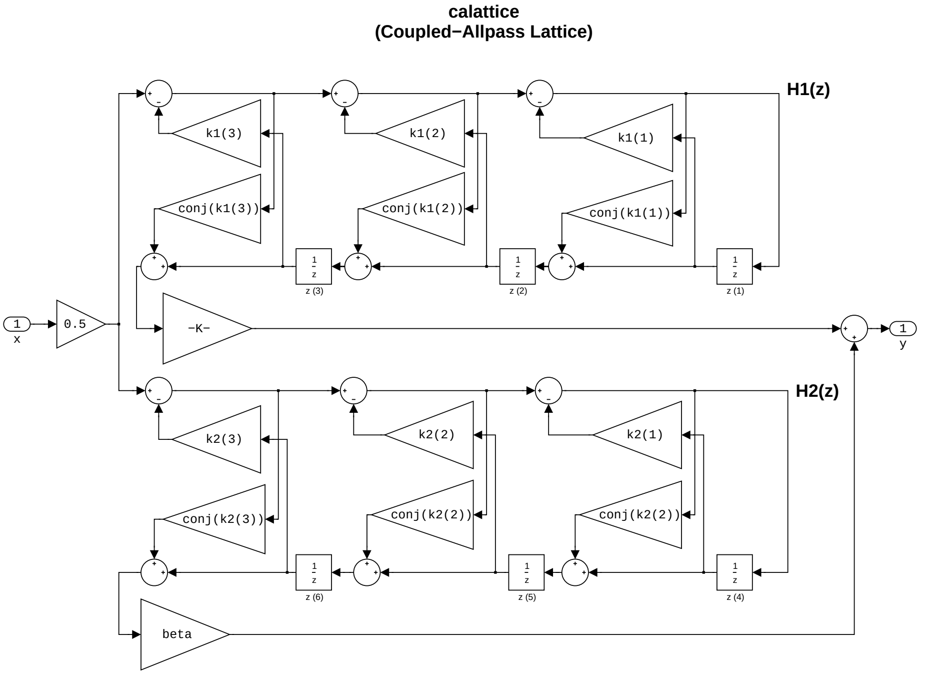
Syntax

```
hd = dfilt.calattice(k1,k2,beta)
hd = dfilt.calattice
```

Description

`hd = dfilt.calattice(k1,k2,beta)` returns a discrete-time, coupled-allpass, lattice filter object `hd`, which is two allpass, lattice filter structures coupled together. The lattice coefficients for each structure are vectors `k1` and `k2`. Input argument `beta` is shown in the diagram below.

`hd = dfilt.calattice` returns a default, discrete-time coupled-allpass, lattice filter object, `hd`. The default values are `k1` = `k2` = [], which is the default value for `dfilt.latticeallpass`, and `beta` = 1. This filter passes the input through to the output unchanged.



Examples

Specify a third-order lattice coupled-allpass filter structure for a `dfilt` filter, `hd` with the following code.

```
k1 = [0.9511 + 1j*0.3088; 0.7511 + 1j*0.1158];
k2 = 0.7502 - 1j*0.1218;
beta = 0.1385 + 1j*0.9904;
hd = dfilt.calattice(k1,k2,beta);
```

The `Allpass1` and `Allpass2` properties store vectors of coefficients.

See Also

`dfilt.calatticepc` | `dfilt` | `dfilt.latticeallpass` | `dfilt.latticear` | `dfilt.latticearma` | `dfilt.latticemamax` | `dfilt.latticemamin`

Introduced in R2011a

dfilt.calatticepc

Coupled-allpass, power-complementary lattice filter

Syntax

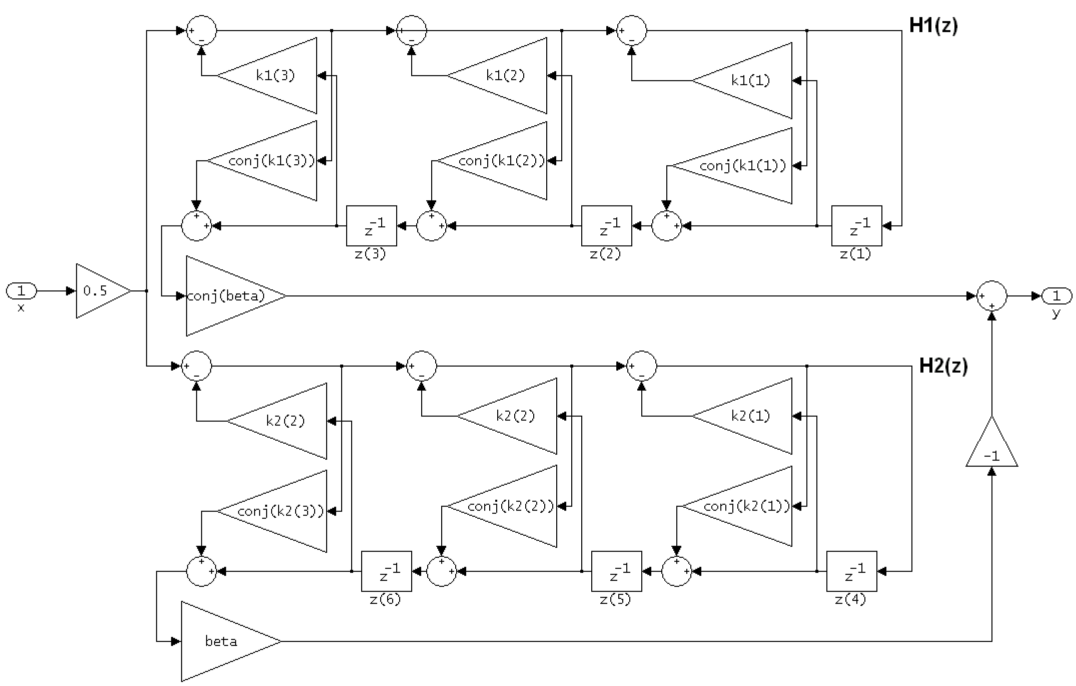
```
hd = dfilt.calatticepc(k1,k2)
hd = dfilt.calatticepc
```

Description

`hd = dfilt.calatticepc(k1,k2)` returns a discrete-time, coupled-allpass, lattice filter object `hd`, with power-complementary output. This object is two allpass lattice filter structures coupled together to produce complementary output. The lattice coefficients for each structure are vectors, `k1` and `k2`, respectively. `beta` is shown in the following diagram.

`hd = dfilt.calatticepc` returns a default, discrete-time, coupled-allpass, lattice filter object `hd`, with power-complementary output. The default values are `k1` = `k2` = [], which is the default value for the `dfilt.latticeallpass`. The default for `beta` = 1. This filter passes the input through to the output unchanged.

calatticepc
(Coupled Allpass Lattice, Power Complementary output)



Examples

Specify a third-order lattice coupled-allpass power complementary filter structure for a filter `hd` with the following code. You see from the returned properties that `Allpass1` and `Allpass2` contain vectors of coefficients for the constituent filters.

```
k1 = [0.9511 + 0.3088i; 0.7511 + 0.1158i];
k2 = 0.7502 - 0.1218i;
beta = 0.1385 + 0.9904i;
hd = dfilt.calatticepc(k1,k2,beta);
```

To see the coefficients for `Allpass1`, check the property values.

```
get(hd,'Allpass1')
```

See Also

`dfilt.calattice` | `dfilt` | `dfilt.latticeallpass` | `dfilt.latticear` | `dfilt.latticearma` | `dfilt.latticemamax` | `dfilt.latticemamin`

Introduced in R2011a

dfilt.cascade

Cascade of discrete-time filters

Syntax

Refer to `dfilt.cascade` in Signal Processing Toolbox documentation for more information.

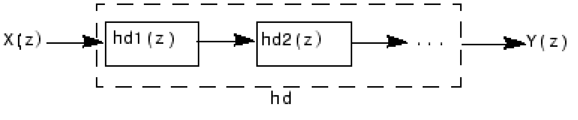
Description

`hd = dfilt.cascade(filterobject1,filterobject2,...)` returns a discrete-time filter object `hd` of type `cascade`, which is a serial interconnection of two or more filter objects `filterobject1`, `filterobject2`, and so on. `dfilt.cascade` accepts any combination of `dfilt` objects (discrete time filters) to cascade, as well as Farrow filter objects.

You can use the standard notation to cascade one or more filters:

```
cascade(hd1,hd2,...)
```

where `hd1`, `hd2`, and so on can be mixed types, such as `dfilt` objects and other filtering objects.



`hd1`, `hd2`, and so on can be fixed-point filters. All filters in the cascade must be the same arithmetic format — `double`, `single`, or `fixed`. `hd`, the filter object returned, inherits the format of the cascaded filters.

Examples

Cascade a lowpass filter and a highpass filter to produce a bandpass filter.

```
[b1,a1]=butter(8,0.6);    % Lowpass
```

```
[b2,a2]=butter(8,0.4,'high');  % Highpass
h1=dfilt.df2t(b1,a1);
h2=dfilt.df2t(b2,a2);
hcas=dfilt.cascade(h1,h2);   % Bandpass with passband 0.4-0.6
% View stage 1 with hcas.Stage(1)
```

See Also

`dfilt` | `dfilt.parallel` | `dfilt.scalar`

Introduced in R2011a

dfilt.cascadeallpass

Cascade of allpass discrete-time filters

Syntax

```
hd = dfilt.cascadeallpass(c1,c2,...)
```

Description

`hd = dfilt.cascadeallpass(c1,c2,...)` constructs a cascade of allpass filters, each of which uses the minimum number of multipliers, given the filter coefficients provided in `c1`, `c2`, and so on.

Each vector `c` represents one section in the cascade filter. `c` vectors must contain one, two, three, or four elements as the filter coefficients for each section. As a result of the design algorithm, each section is a `dfilt.allpass` structure whose coefficients are given in the matching `c` vector, such as the `c1` vector contains the coefficients for the first stage.

States for each section are shared between sections.

Vectors `c` do not have to be the same length. You can combine various length vectors in the input arguments. For example, you can cascade fourth-order sections with second-order sections, or first-order sections.

For more information about the vectors `ci` and about the transfer function of each section, refer to `dfilt.allpass`.

Generally, you do not construct these allpass cascade filters directly. Instead, they result from the design process for an IIR filter. Refer to the first example in Examples for more about using `dfilt.cascadeallpass` to design an IIR filter.

Properties

In the next table, the row entries are the filter properties and a brief description of each property.

Property Name	Brief Description
AllpassCoefficients	Contains the coefficients for the allpass filter object
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.

Examples

Two examples show how dfilt.cascadeallpass works in very different applications — designing a halfband IIR filter and constructing an allpass cascade of dfilt objects.

First, design the IIR halfband filter using cascaded allpass filters. Each branch of the parallel cascade construction is a cascadeallpass filter object.

```
tw = 100;  % Transition width of filter to be designed, 100 Hz.
ast = 80;  % Stopband attenuation of filter to be designed, 80dB.
fs = 2000; % Sampling frequency of signal to be filtered.
% Store halfband design specs in the specifications object d.
d = fdesign.halfband('tw,ast',tw,ast,fs);
```

Now perform the actual filter design. hd contains two dfilt.cascadeallpass objects.

```
hd = design(d,'ellip','filterstructure','cascadeallpass');
% Get summary information about one dfilt.cascadeallpass stage.
StageInfo = hd.Stage(1).Stage(1);
```

This second example constructs a `dfilt.cascadeallpass` filter object directly given allpass coefficients for the input vectors.

```
section1 = 0.8;
section2 = [1.2,0.7];
section3 = [1.3,0.9];
hd = dfilt.cascadeallpass(section1,section2,section3);
% Get information about the filter
% return informatio in character array
S = info(hd);
```

See Also

`dfilt` | `dfilt.allpass` | `dfilt.cascadewdfallpass` | dsp.IIRHalfbandDecimator | dsp.IIRHalfbandDecimator

Introduced in R2011a

dfilt.cascadewdfallpass

Cascade allpass WDF filters to construct allpass WDF

Syntax

```
hd = dfilt.cascadewdfallpass(c1,c2,...)
```

Description

`hd = dfilt.cascadewdfallpass(c1,c2,...)` constructs a cascade of allpass wave digital filters given the allpass coefficients in the vectors `c1`, `c2`, and so on.

Each `c` vector contains the coefficients for one section of the cascaded filter. `C` vectors must have one, two, or four elements (coefficients). Three element vectors are not supported.

When the `c` vector has four elements, the first and third elements of the vector must be 0. Each section of the cascade is an allpass wave digital filter, from `dfilt.wdfallpass`, with the coefficients given by the corresponding c vector. That is, the first section has coefficients from vector `c1`, the second section coefficients come from `c2`, and on until all of the `c` vectors are used.

You can mix the lengths of the `c` vectors. They do not need to be the same length. For example, you can cascade several fourth-order sections (`length(c)` = 4) with first or second-order sections.

Wave digital filters are usually used to create other filters. This toolbox uses them to implement halfband filters, which the first example in Examples demonstrates. They are most often building blocks for filters.

Generally, you do not construct these WDF allpass cascade filters directly. Instead, they result from the design process for an IIR filter. Refer to the first example in Examples for more about using `dfilt.cascadewdfallpass` to design an IIR filter.

For more information about the `c` vectors and the transfer function for the allpass filters, refer to `dfilt.wdfallpass`.

Properties

In the next table, the row entries are the filter properties and a brief description of each property.

Property Name	Brief Description
AllpassCoefficients	Contains the coefficients for the allpass wave digital filter object
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to `filtstates` in Signal Processing Toolbox documentation or in the Help system.

Examples

To demonstrate two approaches to using `dfilt.cascadewdfallpass` to design a filter, these examples show both direct construction and construction as part of another filter.

The first design shown creates an IIR halfband filter that uses lattice wave digital filters. Each branch of the parallel connection in the lattice is an allpass cascade wave digital filter.

```
tw = 100;  % Transition width of filter, 100 Hz.
ast = 80;  % Stopband attenuation of filter, 80 dB.
fs = 2000; % Sampling frequency of signal to filter.
% Store halfband specs.
```

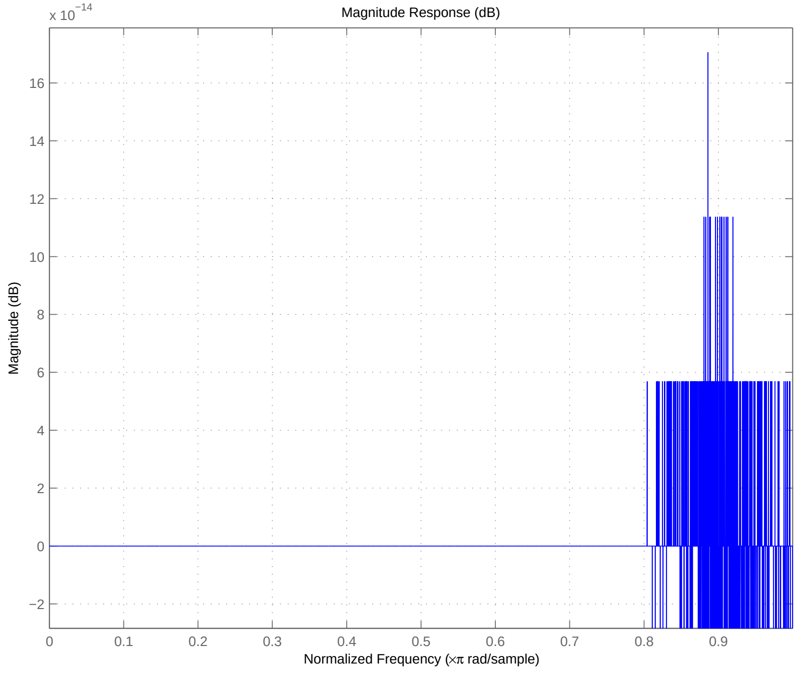
```
d = fdesign.halfband('tw,ast',tw,ast,fs);
```

Now perform the actual halfband design process. `hd` contains two `dfilt.cascadewdfallpass` filters.

```
hd = design(d,'ellip','filterstructure','cascadewdfallpass');
% Summary info on dfilt.cascadewdfallpass.
StageSummary = hd.stage(1).stage(2);
```

This example demonstrates direct construction of a `dfilt.cascadewdfallpass` filter with allpass coefficients.

```
section1 = 0.8;
section2 = [1.5,0.7];
section3 = [1.8,0.9];
hd = dfilt.cascadewdfallpass(section1,section2,section3);
```



See Also

`dfilt` | `dfilt.wdfallpass`

Introduced in R2011a

dfilt.delay

Delay filter

Syntax

```
Hd = dfilt.delay
Hd = dfilt.delay(latency)
```

Description

`Hd = dfilt.delay` returns a discrete-time filter, `Hd`, of type `delay`, which adds a single delay to any signal filtered with `Hd`. The filtered signal has its values shifted by one sample.

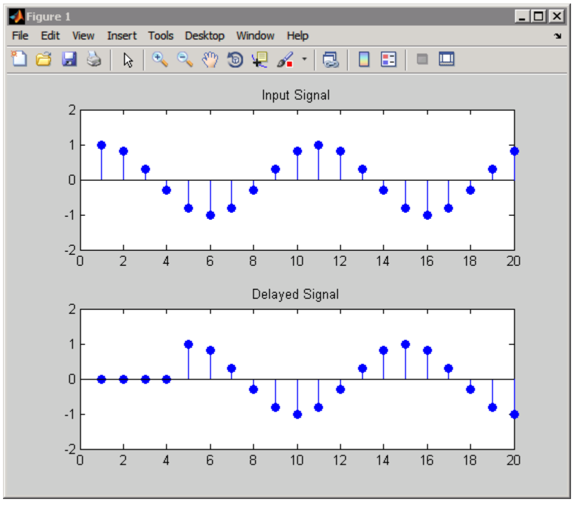
`Hd = dfilt.delay(latency)` returns a discrete-time filter, `Hd`, of type `delay`, which adds the number of delay units specified in `latency` to any signal filtered with `Hd`. The filtered signal has its values shifted by the `latency` number of samples. The values that appear before the shifted signal are the filter states.

Examples

Create a `delay` filter with a `latency` of 4 and filter a simple signal to view the impact of applying a delay.

```
h = dfilt.delay(4);
Fs = 1000;
t = 0:1/Fs:1;
sig = cos(2*pi*100*t);
y = filter(h,sig);
subplot(211);
stem(sig,'markerfacecolor',[0 0 1]);
axis([0 20 -2 2]);
title('Input Signal');
subplot(212);
stem(y,'markerfacecolor',[0 0 1]);
axis([0 20 -2 2]);
```

```
title('Delayed Signal');
```



See Also

dfilt

Introduced in R2011a

dfilt.df1

Discrete-time, direct-form I filter

Syntax

Refer to `dfilt.df1` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.df1` returns a default discrete-time, direct-form I filter object that uses double-precision arithmetic. By default, the numerator and denominator coefficients b and a are set to 1. With these coefficients the filter passes the input to the output without changes.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

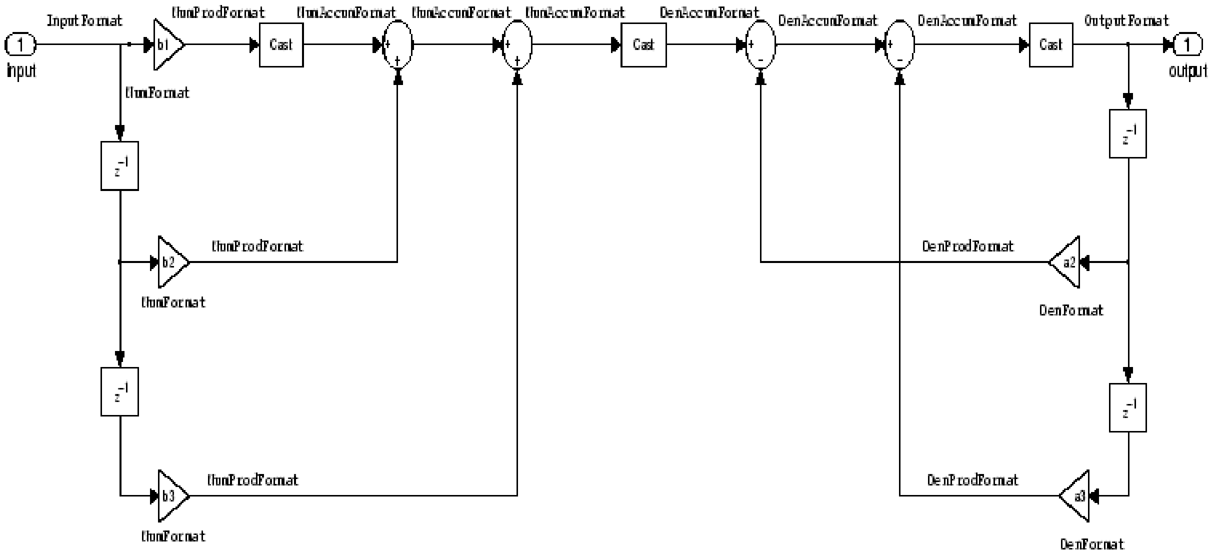
  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to “Arithmetic” on page 6-18.

Note `a(1)`, the leading denominator coefficient, cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form I filter implemented by `dfilt.df1`. To help you see how the filter processes the coefficients, input, output, and states of the filter, as well as numerical operations, the figure includes the locations of the arithmetic and data type format elements within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFormat	`AccumWordLength`	`DenAccumFracLength`	`AccumMode`, `CastBeforeSum`

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenFormat	`CoeffWordLength`	`DenFracLength`	`CoeffAutoScale` , `SignedDenominator`
DenProdFormat	`CoeffWordLength`	`DenProdFracLength`	`ProductMode`, `ProductWordLength`
InputFormat	`InputWordLength`	`InputFracLength`	None
NumAccumFormat	`AccumWordLength`	`NumAccumFracLength`	`AccumMode`, `CastBeforeSum`
NumFormat	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `Signed`, `Numerator`
NumProdFormat	`CoeffWordLength`	`NumProdFracLength`	`ProductWordLength`, `ProductMode`
OutputFormat	`OutputWordLength`	`OutputFracLength`	`OutputMode`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFormat, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFormat refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with df1 implementations of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use `get(hd)` where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 6-2.

Property Name	Brief Description
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` and `DenFracLength` properties to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`DenAccumFracLength`	Specifies the fraction length the filter algorithm uses to interpret the results of product operations involving denominator coefficients. You can change the value for this property when you set `AccumMode` to `SpecifyPrecision`.
`DenFracLength`	Set the fraction length the filter uses to interpret denominator coefficients. `DenFracLength` is always available, but it is read-only until you set `CoeffAutoScale` to `false`.
`Denominator`	Stores the denominator coefficients for the IIR filter.

Property Name	Brief Description
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
Numerator	Holds the numerator coefficient values for the filter.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputWordLength	Determines the word length used for the output data.

Property Name	Brief Description
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
`RoundMode`	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `ceil` - Round toward positive infinity. • `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • `fix` - Round toward zero. • `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to `filtstates` in Signal Processing Toolbox documentation or in the Help system.

Examples

Specify a second-order direct-form I structure for a `dfilt` object, `hd`, with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1(b,a);
% Convert hd to fixed-point filter
set(hd,'arithmetic','fixed')
```

See Also

dfilt | dfilt.df1t | dfilt.df2 | dfilt.df2t

Introduced in R2011a

dfilt.df1sos

Discrete-time, SOS direct-form I filter

Syntax

Refer to `dfilt.df1sos` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.df1sos(s)` returns a discrete-time, second-order section, direct-form I filter object `hd`, with coefficients given in the `s` matrix.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 6-18.

`hd = dfilt.df1sos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form I filter object `hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, and so on.

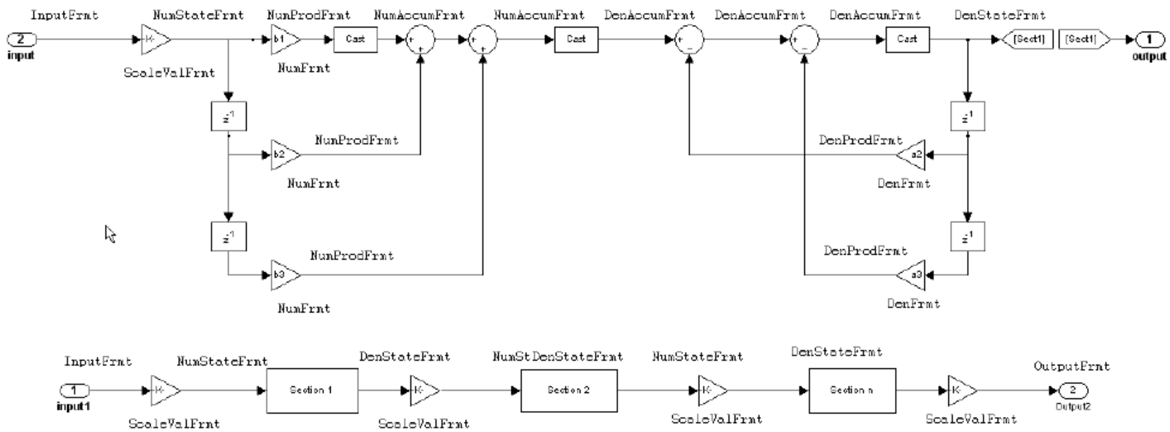
`hd = dfilt.df1sos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. When you do not specify `g`, all gains default to one.

`hd = dfilt.df1sos` returns a default, discrete-time, second-order section, direct-form I filter object, `hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form I filter implemented in second-order sections by `dfilt.df1sos`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt” (format). In this use, “frmt“ means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Similarly consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFrmt	CoeffWordLength	DenFracLength	CoeffAutoScale, Signed, Denominator
DenProdFrmt	ProductWordLength	DenProdFracLength	ProductMode
DenStateFrmt	DenStateWordLength	DenStateFracLength	CastBeforeSum, States
InputFrmt	InputWordLength	InputFracLength	None
NumAccumFrmt	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFrmt	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFrmt	ProductWordLength	NumProdFracLength	ProductMode
NumStateFrmt	NumStateWordLength	NumStateFracLength	States
OutputFrmt	OutputWordLength	OutputFracLength	OutputMode
ScaleValueFrmt	CoeffWordLength	ScaleValueFracLength	CoeffAutoScale, ScaleValues

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties `ProductWordLength`, `DenProdFracLength` and `ProductMode` that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with SOS implementation of direct-form I `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 6-2.

Property Name	Brief Description
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` and `DenFracLength` properties to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`DenAccumFracLength`	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations.

Property Name	Brief Description
	You can change the value for this property when you set `AccumMode` to `SpecifyPrecision`.
`DenFracLength`	Set the fraction length the filter uses to interpret denominator coefficients. `DenFracLength` is always available, but it is read-only until you set `CoeffAutoScale` to `false`.
`DenProdFracLength`	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set `ProductMode` to `SpecifyPrecision`.
`DenStateFracLength`	Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.
`DenStateWordLength`	Specifies the word length used to represent the states associated with denominator coefficients in the filter.
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
`InputFracLength`	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Specifies the word length applied to interpret input data.
`NumAccumFracLength`	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set `AccumMode` to `SpecifyPrecision`.
`NumFracLength`	Sets the fraction length used to interpret the value of numerator coefficients.
`NumStateFracLength`	Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.
`NumWordFracLength`	Specifies the word length used to interpret the states associated with numerator coefficient operations in the filter.

Property Name	Brief Description
OptimizeScaleValues	When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length applied for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
ScaleValueFracLength	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable AutoScaleMode by setting it to false.
ScaleValues	Scaling for the filter objects in SOS filters.

Property Name	Brief Description
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
SosMatrix	Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/section datatype]. A [15x6 double] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type double to represent the coefficients.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Specify a fixed-point, second-order section, direct-form I dfilt object with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1sos(b,a);
% Convert to fixed-point filter
hd.arithmetic = 'fixed';
```

See Also

dfilt | dfilt.df2tsos

Introduced in R2011a

dfilt.df1t

Discrete-time, direct-form I transposed filter

Syntax

Refer to `dfilt.df1t` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.df1t(b,a)` returns a discrete-time, direct-form I transposed filter object `hd`, with numerator coefficients `b` and denominator coefficients `a`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

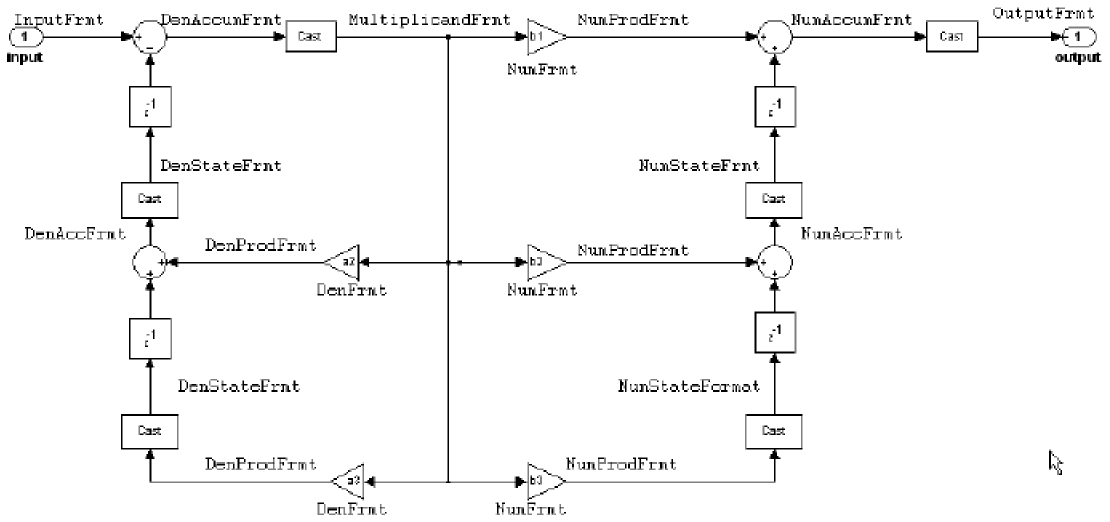
For more information about the property `Arithmetic`, refer to “Arithmetic” on page 6-18.

`hd = dfilt.df1t` returns a default, discrete-time, direct-form I transposed filter object `hd`, with `b`=1 and `a`=1. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The following figure shows the signal flow for the transposed direct-form I filter implemented by `dfilt.df1t`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters "frmt" (format). In this use, "frmt" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	`AccumWordLength`	`DenAccumFracLength`	`AccumMode`, `CastBeforeSum`
DenFrmt	`CoeffWordLength`	`DenFracLength`	`CoeffAutoScale`, , `Signed`, `Denominator`
DenProdFrmt	`CoeffWordLength`	`DenProdFracLength`	`ProductMode`, `ProductWordLength`
DenStateFrmt	`DenStateWordLength`	`DenStateFracLength`	`CastBeforeSum`, `States`
InputFrmt	`InputWordLength`	`InputFracLength`	None
Multiplicandfrmt	`Multiplicand-WordLength`	`Multiplicand-FracLength`	`CastBeforeSum`
NumAccumFrmt	`AccumWordLength`	`NumAccumFracLength`	`AccumMode`, `CastBeforeSum`
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `Signed`, `Numerator`
NumProdFrmt	`CoeffWordLength`	`NumProdFracLength`	`ProductWordLength`, `ProductMode`
NumStateFrmt	`NumStateWordLength`	`NumStateFracLength`	`States`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	`OutputMode`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with df1t implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 6-2.

Property Name	Brief Description
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this

Property Name	Brief Description
	off by setting the value to `false` enables you to change the `NumFracLength` and `DenFracLength` properties to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`DenAccumFracLength`	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set `AccumMode` to `SpecifyPrecision`.
`DenFracLength`	Set the fraction length the filter uses to interpret denominator coefficients. `DenFracLength` is always available, but it is read-only until you set `CoeffAutoScale` to `false`.
`Denominator`	Holds the denominator coefficients for the filter.
`DenProdFracLength`	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set `ProductMode` to `SpecifyPrecision`.
`DenStateFracLength`	Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
`InputFracLength`	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Specifies the word length applied to interpret input data.
`MultiplicandFracLength`	Sets the fraction length for values (multiplicands) used in multiply operations in the filter.
`MultiplicandWordLength`	Sets the word length applied to the values input to a multiply operation (the multiplicands).

Property Name	Brief Description
`NumAccumFracLength`	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set `AccumMode` to `SpecifyPrecision`.
`Numerator`	Holds the numerator coefficient values for the filter.
`NumFracLength`	Sets the fraction length used to interpret the value of numerator coefficients.
`NumProdFracLength`	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set `ProductMode` to `SpecifyPrecision`.
`NumStateFracLength`	For IIR filters, this defines the binary point location applied to the numerator states of the filter. Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.
`OutputFracLength`	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `OutputMode` to `SpecifyPrecision`.
`OutputMode`	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • `AvoidOverflow` — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • `BestPrecision` — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • `SpecifyPrecision` — lets you set the word and fraction lengths used by the output data from filtering.
`OutputWordLength`	Determines the word length used for the output data.

Property Name	Brief Description
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.

Property Name	Brief Description
`RoundMode`	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `ceil` - Round toward positive infinity. • `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • `fix` - Round toward zero. • `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`StateAutoScale`	Setting autoscaling for filter states to `true` reduces the possibility of overflows occurring during fixed-point operations. Set to `false`, `StateAutoScale` lets the filter select the fraction length to limit the overflow potential.
`States`	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
`StateWordLength`	Sets the word length used to represent the filter states.

Examples

Specify a second-order direct-form I transposed filter structure for a `dfilt` object, `hd`, with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1t(b,a);
% Convert filter to single-precision arithmetic
set(hd,'arithmetic','single')
```

See Also

`dfilt` | `dfilt.df1` | `dfilt.df2` | `dfilt.df2t`

Introduced in R2011a

dfilt.df1tsos

Discrete-time, SOS direct-form I transposed filter

Syntax

Refer to `dfilt.df1tsos` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.df1tsos(s)` returns a discrete-time, second-order section, direct-form I, transposed filter object `hd`, with coefficients given in the `s` matrix.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to .

`hd = dfilt.df1tsos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form I, transposed filter object `hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

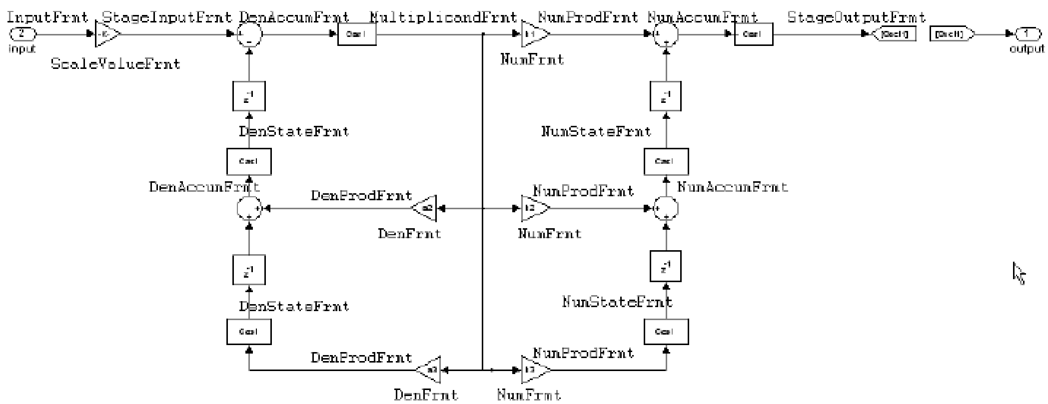
`hd = dfilt.df1tsos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

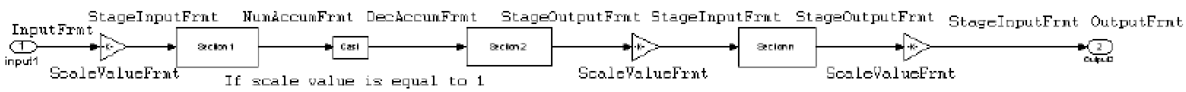
`hd = dfilt.df1tsos` returns a default, discrete-time, second-order section, direct-form I, transposed filter object, `hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form I transposed filter implemented using second-order sections by `dfilt.df1tsos`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.





Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt” (format). In this use, “frmt” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFrmt	CoeffWordLength	DenFracLength	CoeffAutoScale, Signed, Denominator
DenProdFrmt	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
DenStateFrmt	DenStateWordLength	DenStateFracLength	CastBeforeSum, States
InputFrmt	InputWordLength	InputFracLength	None
MultiplicandFrmt	Multiplicand-WordLength	Multiplicand-FracLength	CastBeforeSum
NumAccumFrmt	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFrmt	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFrmt	CoeffWordLength	NumProdFracLength	ProductWordLength, ProductMode
NumStateFrmt	NumStateWordLength	NumStateFracLength	States
OutputFrmt	OutputWordLength	OutputFracLength	OutputMode
ScaleValueFrmt	CoeffWordLength	ScaleValue-FracLength	CoeffAutoScale, ScaleValues

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties ProdWordLength, ProductMode and DenProdFracLength that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with SOS implementation of transposed direct-form I dfilt objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where hd is a filter.

For further information about the properties of this filter or any dfilt object, refer to "Fixed-Point Filter Properties" on page 6-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.

Property Name	Brief Description
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` and `DenFracLength` properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set `AccumMode` to `SpecifyPrecision`.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. `DenFracLength` is always available, but it is read-only until you set `CoeffAutoScale` to `false`.
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set `ProductMode` to `SpecifyPrecision`.
DenStateFracLength	Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.

Property Name	Brief Description
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
MultiplicandFracLength	Sets the fraction length for values (multiplicands) used in multiply operations in the filter.
MultiplicandWordLength	Sets the word length applied to the values input to a multiply operation (the multiplicands)
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
Numerator	Holds the numerator coefficient values for the filter.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
NumStateFracLength	For IIR filters, this defines the binary point location applied to the numerator states of the filter. Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.
NumStateWordLength	For IIR filters, this defines the word length applied to the numerator states of the filter. Specifies the word length used to interpret the states associated with numerator coefficient operations in the filter.
OptimizeScaleValues	When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.

Property Name	Brief Description
`OutputFracLength`	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `OutputMode` to `SpecifyPrecision`.
`OutputMode`	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • `AvoidOverflow` — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • `BestPrecision` — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • `SpecifyPrecision` — lets you set the word and fraction lengths used by the output data from filtering.
`OutputWordLength`	Determines the word length used for the output data.
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.

Property Name	Brief Description
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.

Property Name	Brief Description
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
ScaleValueFracLength	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable AutoScaleMode by setting it to false.
ScaleValues	Scaling for the filter objects in SOS filters.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.

Property Name	Brief Description
SosMatrix	Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/sectiondatatype]. A [15x6 double] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type double to represent the coefficients.
StateAutoScale	Setting autoscaling for filter states to true reduces the possibility of overflows occurring during fixed-point operations. Set to false, StateAutoScale lets the filter select the fraction length to limit the overflow potential.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples

With the following code, this example specifies a second-order section, direct-form I transposed dfilt object for a filter. Then convert the filter to fixed-point operation.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1tsos(b,a);
set(hd,'arithmetic','fixed')
```

See Also

dfilt | dfilt.df1sos | dfilt.df2sos | dfilt.df2tsos

Introduced in R2011a

dfilt.df2

Discrete-time, direct-form II filter

Syntax

Refer to `dfilt.df2` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.df2(b,a)` returns a discrete-time, direct-form II filter object `hd`, with numerator coefficients `b` and denominator coefficients `a`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

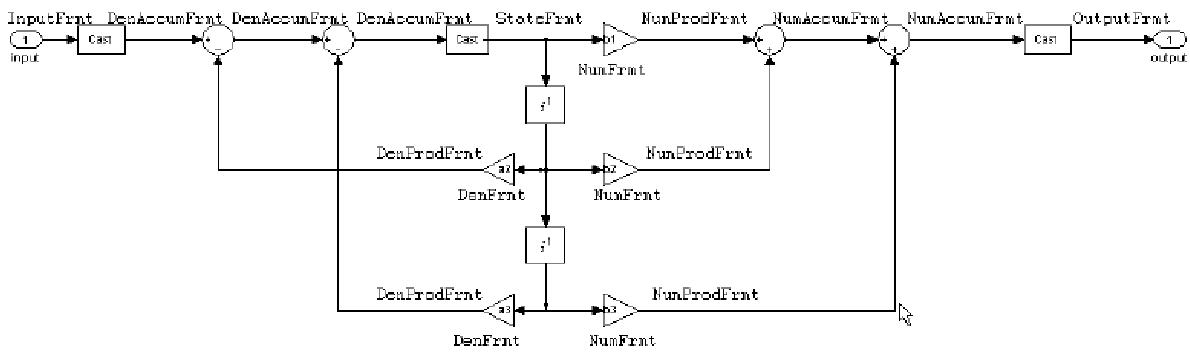
For more information about the property `Arithmetic`, refer to "Arithmetic" on page 6-18.

`hd = dfilt.df2` returns a default, discrete-time, direct-form II filter object `hd`, with `b` = 1 and `a` = 1. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form II filter implemented by `dfilt.df2`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters“frmt” (format). In this use, “frmt” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	`AccumWordLength`	`DenAccumFracLength`	`AccumMode`, `CastBeforeSum`

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenFrmt	`CoeffWordLength`	`DenFracLength`	`CoeffAutoScale`, `Signed`, `Denominator`
DenProdFrmt	`CoeffWordLength`	`DenProdFracLength`	`ProductMode`, `ProductWordLength`
InputFrmt	`InputWordLength`	`InputFracLength`	None
NumAccumFrmt	`AccumWordLength`	`NumAccumFracLength`	`AccumMode`, `CastBeforeSum`
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `Signed`, `Numerator`
NumProdFrmt	`CoeffWordLength`	`NumProdFracLength`	`ProductWordLength`, `ProductMode`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	`OutputMode`
StateFrmt	`StateWordLength`	`StateFracLength`	`States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with the df2 implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 6-2.

Property Name	Brief Description
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` and `DenFracLength` properties to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`DenAccumFracLength`	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set `AccumMode` to `SpecifyPrecision`.

Property Name	Brief Description
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
Denominator	Holds the denominator coefficients for IIR filters.
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
Numerator	Holds the numerator coefficient values for the filter.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.

Property Name	Brief Description
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.

Property Name	Brief Description
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `ceil` - Round toward positive infinity. • `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • `fix` - Round toward zero. • `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateFracLength	When you set `StateAutoScale` to `false`, you enable the `StateFracLength` property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Specify a second-order direct-form II filter structure for a `dfilt` object, `hd`, with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df2(b,a);
% Change filter to fixed-point
set(hd,'arithmetic','fixed')
```

See Also

`dfilt` | `dfilt.df1` | `dfilt.df1t` | `dfilt.df2t`

Introduced in R2011a

dfilt.df2sos

Discrete-time, SOS, direct-form II filter

Syntax

Refer to `dfilt.df2sos` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.df2sos(s)` returns a discrete-time, second-order section, direct-form II filter object `hd`, with coefficients given in the `s` matrix.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to “Arithmetic” on page 6-18.

`hd = dfilt.df2sos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form II object, `hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

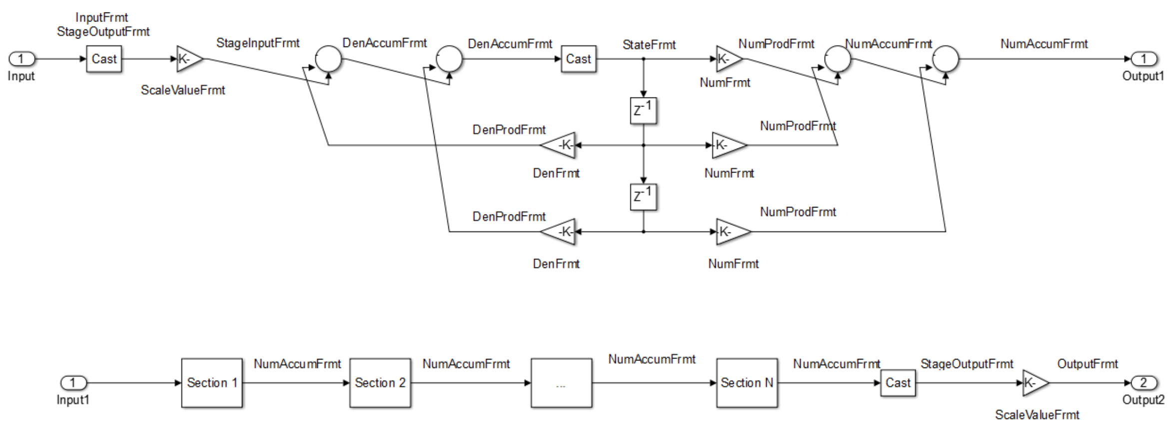
`hd = dfilt.df2sos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`hd = dfilt.df2sos` returns a default, discrete-time, second-order section, direct-form II filter object, `hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The figure below shows the signal flow for the direct-form II filter implemented with second-order sections by `dfilt.df2sos`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters “frmt” (format). In this use, “frmt” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The frmt properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFrmt	CoeffWordLength	DenFracLength	CoeffAutoScale, Signed, sosMatrix
DenProdFrmt	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength, sosMatrix
InputFrmt	InputWordLength	InputFracLength	None
NumAccumFrmt	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFrmt	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, sosMatrix
NumProdFrmt	CoeffWordLength	NumProdFracLength	ProductWordLength, ProductMode
OutputFrmt	OutputWordLength	OutputFracLength	OutputMode
ScaleValueFrmat	CoeffWordLength	ScaleValue-FracLength	CoeffAutoScale, ScaleValues
SectionInputFormt	SectionInput-WordLength	SectionInput-FracLength	SectionInput-AutoScale
SectionOutputFrmt	SectionOutput-WordLength	SectionOutput-FracLength	SectionOutput-AutoScale
StateFrmt	StateWordLength	StateFracLength	CastBeforeSum, States

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with second-order section implementation of direct-form II `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 6-2.

Property Name	Brief Description
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.

Property Name	Brief Description
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.

Property Name	Brief Description
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OptimizeScaleValues	When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.

Property Name	Brief Description
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.

Property Name	Brief Description
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
ScaleValueFracLength	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable AutoScaleMode by setting it to false.
ScaleValues	Scaling for the filter objects in SOS filters.
SectionInputAutoScale	Tells the filter whether to set the stage input data format to minimize the occurrence of overflow conditions.
SectionInputFracLength	Lets you set the fraction length for section inputs in SOS filters, if you set SectionInputAutoScale to false.
SectionInputWordLength	Lets you set the word length for section inputs in SOS filters, if you set SectionInputAutoScale to false.

Property Name	Brief Description
SectionOutputAutoScale	Tells the filter whether to set the section output data format to minimize the occurrence of overflow conditions.
SectionOutputFracLength	Lets you set the fraction length for section outputs in SOS filters, if you set `SectionOutputAutoScale` to `off`.
SectionOutputWordLength	Lets you set the word length for section outputs in SOS filters, if you set `SectionOutputAutoScale` to `false`.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
SosMatrix	Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/section datatype]. A `[15x6 double]` SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type `double` to represent the coefficients.
StateFracLength	When you set `StateAutoScale` to `false`, you enable the `StateFracLength` property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Specify a second-order section, direct-form II `dfilt` object for a Butterworth filter converted to second-order sections, with the following code:

```
[z,p,k] = butter(30,0.5);
[s,g] = zp2sos(z,p,k);
hd = dfilt.df2sos(s,g);
% Convert filter to fixed-point
hd.arithmetic='fixed';
```

See Also

`dfilt` | `dfilt.df1sos` | `dfilt.df1tsos` | `dfilt.df2tsos`

Introduced in R2011a

dfilt.df2t

Discrete-time, direct-form II transposed filter

Syntax

Refer to `dfilt.df2t` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.df2t(b,a)` returns a discrete-time, direct-form II transposed filter object `hd`, with numerator coefficients `b` and denominator coefficients `a`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

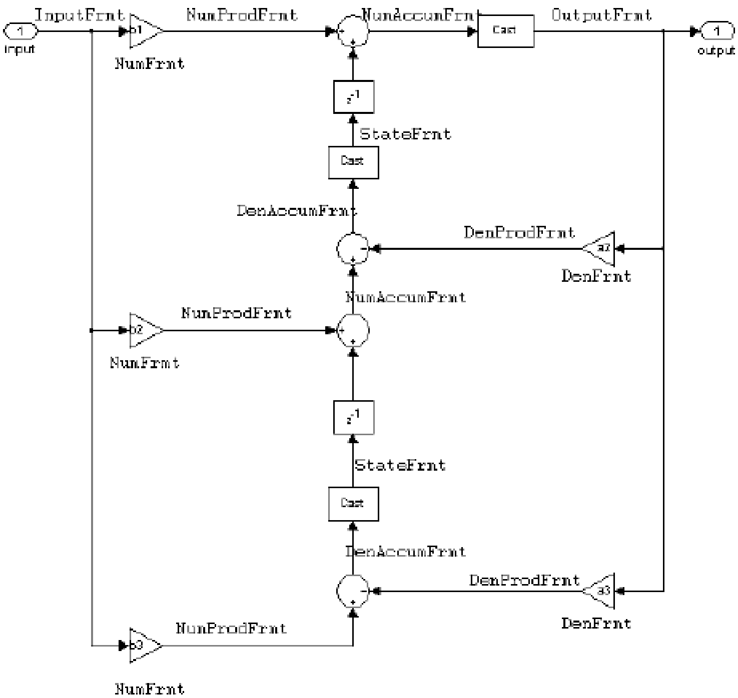
For more information about the property `Arithmetic`, refer to “Arithmetic” on page 6-18.

`hd = dfilt.df2t` returns a default, discrete-time, direct-form II transposed filter object `hd`, with `b` = 1 and `a` = 1. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form II transposed filter implemented by `dfilt.df2t`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters "frmt." In this use, "frmt" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFrmt	CoeffWordLength	DenFracLength	CoeffAutoScale,Signed, Denominator
DenProdFrmt	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
InputFrmt	InputWordLength	InputFracLength	None
NumAccumFrmt	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFrmt	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFrmt	CoeffWordLength	NumProdFracLength	ProductWordLength, ProductMode
OutputFrmt	OutputWordLength	OutputFracLength	None
StateFrmt	StateWordLength	StateFracLength	States

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties ProdWordLength, ProductMode and DenProdFracLength that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with df2t implementation of dfilt objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where hd is a filter.

For further information about the properties of this filter or any dfilt object, refer to "Fixed-Point Filter Properties" on page 6-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value

Property Name	Brief Description
	to `false` enables you to change the `NumFracLength` and `DenFracLength` properties to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`DenAccumFracLength`	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set `AccumMode` to `SpecifyPrecision`.
`DenFracLength`	Set the fraction length the filter uses to interpret denominator coefficients. `DenFracLength` is always available, but it is read-only until you set `CoeffAutoScale` to `false`.
`Denominator`	Holds the denominator coefficients for IIR filters.
`DenProdFracLength`	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set `ProductMode` to `SpecifyPrecision`.
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
`InputFracLength`	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Specifies the word length applied to interpret input data.
`NumAccumFracLength`	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set `AccumMode` to `SpecifyPrecision`.
`Numerator`	Holds the numerator coefficient values for the filter.
`NumFracLength`	Sets the fraction length used to interpret the value of numerator coefficients.
`NumProdFracLength`	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set `ProductMode` to `SpecifyPrecision`.

Property Name	Brief Description
OutputFracLength	Determines how the filter interprets the filter output data.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.

Property Name	Brief Description
`RoundMode`	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `ceil` - Round toward positive infinity. • `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • `fix` - Round toward zero. • `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`StateAutoScale`	Setting autoscaling for filter states to `true` reduces the possibility of overflows occurring during fixed-point operations. Set to `false`, `StateAutoScale` lets the filter select the fraction length to limit the overflow potential.
`StateFracLength`	When you set `StateAutoScale` to `false`, you enable the `StateFracLength` property that lets you set the fraction length applied to interpret the filter states.
`States`	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
`StateWordLength`	Sets the word length used to represent the filter states.

Examples

Create a fixed-point filter by specifying a second-order direct-form II transposed filter structure for a `dfilt` object, and then converting the double-precision arithmetic setting to fixed-point.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df2t(b,a);
% Convert filter to fixed-point
set(hd,'arithmetic','fixed')
```

See Also

`dfilt` | `dfilt.df1` | `dfilt.df1t` | `dfilt.df2`

Introduced in R2011a

dfilt.df2tsos

Discrete-time, SOS direct-form II transposed filter

Syntax

Refer to `dfilt.df2tsos` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.df2tsos(s)` returns a discrete-time, second-order section, direct-form II, transposed filter object `hd`, with coefficients given in the matrix `s`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 6-18.

`hd = dfilt.df2tsos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form II, transposed filter object `hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

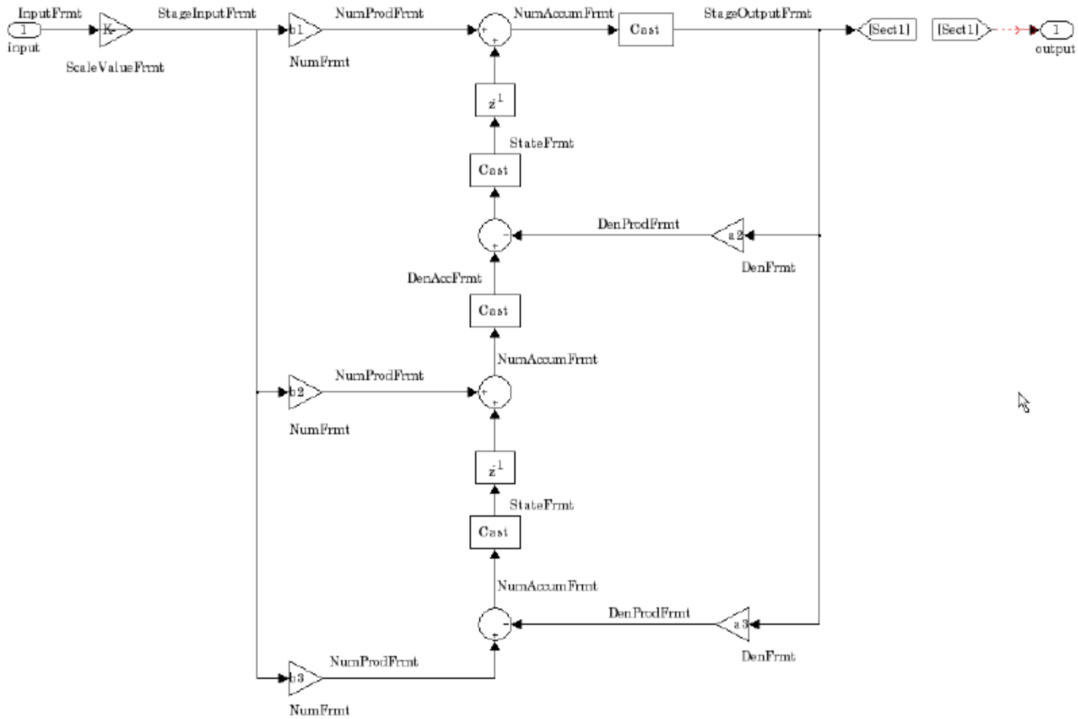
`hd = dfilt.df2tsos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

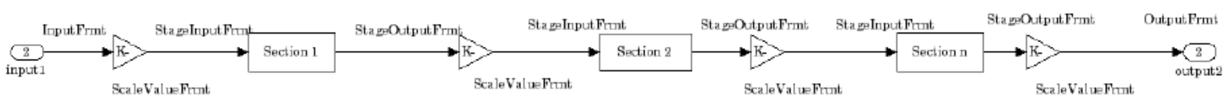
`hd = dfilt.df2tsos` returns a default, discrete-time, second-order section, direct-form II, transposed filter object, `hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The figure below shows the signal flow for the second-order section transposed direct-form II filter implemented by `dfilt.dftsos`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.





Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters "frmt" (format). In this use, "frmt" indicates the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	`AccumWordLength`	`DenAccumFracLength`	`AccumMode`, `CastBeforeSum`
DenFrmt	`CoeffWordLength`	`DenFracLength`	`CoeffAutoScale`, `Signed`, `Denominator`
DenProdFrmt	`CoeffWordLength`	`DenProdFracLength`	`ProductMode`, `ProductWordLength`
InputFrmt	`InputWordLength`	`InputFracLength`	None
NumAccumFrmt	`AccumWordLength`	`NumAccumFracLength`	`AccumMode`, `CastBeforeSum`
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `SignedNumerator`
NumProdFrmt	`CoeffWordLength`	`NumProdFracLength`	`ProductWordLength`, `ProductMode`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	`OutputMode`
ScaleValueFrmt	`CoeffWordLength`	`ScaleValueFracLengt`	`CoeffAutoScale`, `ScaleValues`
SectionInputFormt	`SectionInput-WordLength`	`SectionInput-FracLength`	
SectionOutputFrmt	`SectionOutput-WordLength`	`SectionOutput-FracLength`	
StateFrmt	`StateWordLength`	`StateFracLength`	`States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties

In this table you see the properties associated with second-order section implementation of transposed direct-form II `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to “Fixed-Point Filter Properties” on page 6-2.

Property Name	Brief Description
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.

Property Name	Brief Description
AccumWordLength	Sets the word length used to store data in the accumulator/ buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.

Property Name	Brief Description
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OptimizeScaleValues	When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.

Property Name	Brief Description
`OverflowMode`	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
`ProductMode`	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
ScaleValueFracLength	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable AutoScaleMode by setting it to false.
ScaleValues	Scaling for the filter objects in SOS filters.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
SosMatrix	Holds the filter coefficients as property values — you use set and get to modify them. Displays the matrix in the format [sections x coefficients/section data type]. A [15x6 double] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type double to represent the coefficients.

Property Name	Brief Description
SectionInputFracLength	Lets you set the fraction length for section inputs in SOS filters, if you set SectionInputAutoScale to false.
SectionInputWordLength	Lets you set the word length for section inputs in SOS filters, if you set SectionInputAutoScale to false.
SectionOutputFracLength	Lets you set the fraction length for section outputs in SOS filters, if you set SectionOutputAutoScale to off.
SectionOutputWordLength	Lets you set the word length for section outputs in SOS filters, if you set SectionOutputAutoScale to false.
StateAutoScale	Setting autoscaling for filter states to true reduces the possibility of overflows occurring during fixed-point operations. Set to false, StateAutoScale lets the filter select the fraction length to limit the overflow potential.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Construct a second-order section Butterworth filter for fixed-point filtering. Start by specifying a Butterworth filter, and then convert the filter to second-order sections, with the following code:

```
[z,p,k] = butter(30,0.5);
[s,g] = zp2sos(z,p,k);
hd = dfilt.df2tsos(s,g);
% convert filter to fixed-point
hd.arithmetic='fixed';
```

See Also

dfilt | dfilt.df1sos | dfilt.df1tsos | dfilt.df2sos

Introduced in R2011a

dfilt.dfasymfir

Discrete-time, direct-form antisymmetric FIR filter

Syntax

Refer to `dfilt.dfasymfir` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.dfasymfir(b)` returns a discrete-time, direct-form, antisymmetric FIR filter object `hd`, with numerator coefficients `b`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

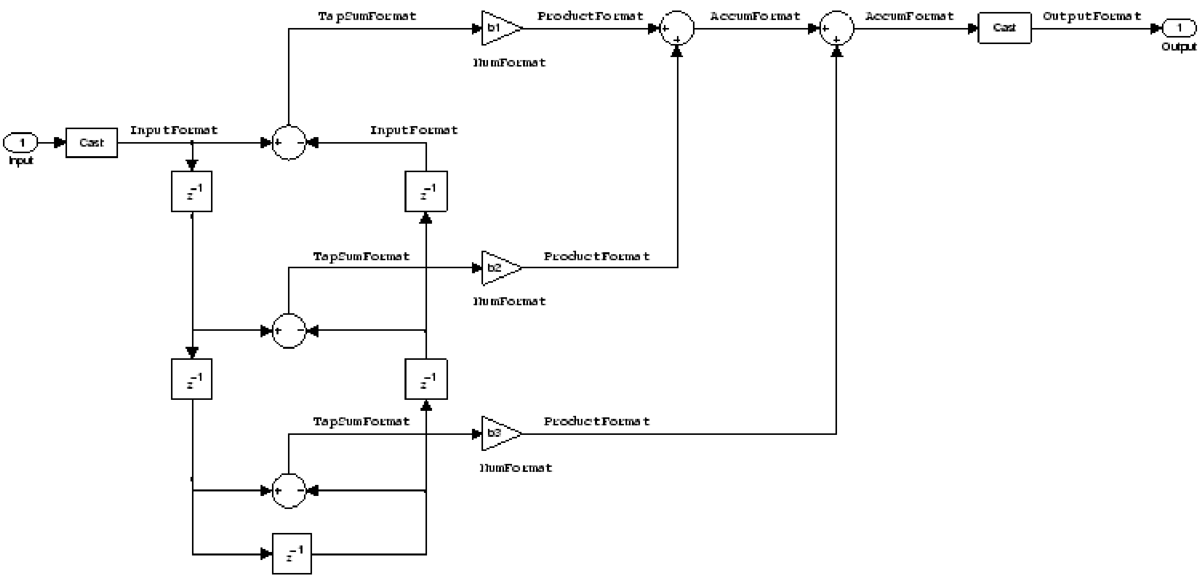
For more information about the property `Arithmetic`, refer to "Arithmetic" on page 6-18.

`hd = dfilt.dfasymfir` returns a default, discrete-time, direct-form, antisymmetric FIR filter object `hd`, with `b`=1. This filter passes the input through to the output unchanged.

Note Only the coefficients in the first half of vector `b` are used because `dfilt.dfasymfir` assumes the coefficients in the second half are antisymmetric to those in the first half. For example, in the figure coefficients, b(4) = -b(3), b(5) = -b(2), and b(6) = -b(1).

Fixed-Point Filter Structure

The following figure shows the signal flow for the odd-order antisymmetric FIR filter implemented by `dfilt.dfasymfir`. The even-order filter uses similar flow. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	AccumWordLength	AccumFracLength	None
InputFormat	InputWordLength	InputFracLength	None
NumFormat	CoeffWordLength	NumFracLength	CoeffAutoScale, , Signed, Numerator
OutputFormat	OutputWordLength	OutputFracLength	None
ProductFormat	ProductWordLength	ProductFracLength	None
TapSumFormat	InputWordLength	InputFracLength	InputFormat

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties ProductFracLength and ProductWordLength that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with an antisymmetric FIR implementation of dfilt objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 6-2.

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits [27]	Specifies the fraction length used to interpret data output by the accumulator.
`AccumWordLength`	Any integer number of bits[33]	Sets the word length used to store data in the accumulator.
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
`CoeffWordLength`	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
`FilterInternals`	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data. Also controls `TapSumFracLength`.

Name	Values	Description
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data. Also determines TapSumWordLength.
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
OutputFracLength	Any positive or negative integer number of bits [29]	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilerInternals to SpecifyPrecision.
OutputWordLength	Any integer number of bits [33]	Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
ProductFracLength	Any positive or negative integer number of bits [27]	Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
ProductWordLength	Any integer number of bits [33]	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.

Name	Values	Description
RoundMode	[convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	fi object to match the filter arithmetic setting	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation.

Examples

Odd Order

Specify a fifth-order direct-form antisymmetric FIR filter structure for a `dfilt` object, `hd`, with the following code:

```
b = [-0.008 0.06 -0.44 0.44 -0.06 0.008];
hd = dfilt.dfasymfir(b);
```

Even Order

Specify a fourth-order direct-form antisymmetric FIR filter structure for `dfilt` object `hd`, with the following code:

```
b = [-0.01 0.1 0.0 -0.1 0.01];
hd = dfilt.dfasymfir(b);
hd.arithmetic='fixed';
FilterCoefs = get(hd,'numerator');
% or equivalently
FilterCoefs = hd.numerator;
```

See Also

`dfilt` | `dfilt.dffir` | `dfilt.dffirt` | `dfilt.dfsymfir`

Introduced in R2011a

dfilt.dffir

Discrete-time, direct-form FIR filter

Syntax

Refer to `dfilt.dffir` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.dffir(b)` returns a discrete-time, direct-form finite impulse response (FIR) filter object `hd`, with numerator coefficients `b`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

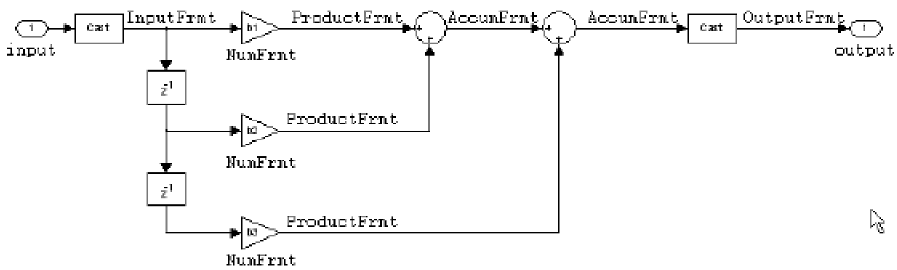
  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to “Arithmetic” on page 6-18.

`hd = dfilt.dffir` returns a default, discrete-time, direct-form FIR filter object `hd`, with `b=1`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form FIR filter implemented by `dfilt.dffir`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters“frmt” (format). In this use, “frmt” indicates the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFrmt	`AccumWordLength`	`AccumFracLength`	None
InputFrmt	`InputWordLength`	`InputFracLength`	None
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `Signed`, `Numerator`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	None

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
ProductFrmt	`ProductWordLength`	`ProductFracLength`	None

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFrmt, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFrmt refers to the properties `ProductFracLength` and `ProductWordLength` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with direct-form FIR implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 6-2.

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits [30]	Specifies the fraction length used to interpret data output by the accumulator.
`AccumWordLength`	Any integer number of bits[34]	Sets the word length used to store data in the accumulator.

Name	Values	Description
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
`CoeffWordLength`	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
`FilterInternals`	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
`NumFracLength`	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
`OutputFracLength`	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `FilerInternals` to `SpecifyPrecision`.
`OutputWordLength`	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting `FilterInternals` to `SpecifyPrecision`.

Name	Values	Description
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
ProductFracLength	Any positive or negative integer number of bits [30]	Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
ProductWordLength	Any integer number of bits [32]	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.

Name	Values	Description
RoundMode	[convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	fi object to match the filter arithmetic setting	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation.

Examples

Specify a second-order direct-form FIR filter structure for a `dfilt` object `hd`, with the following code that constructs the filter in double-precision format and then converts the filter to fixed-point operation:

```
b = [0.05 0.9 0.05];
hd = dfilt.dffir(b);
% Create fixed-point filter
hd.arithmetic='fixed';
% Change FilterInternals property to
% SpecifyPrecision enabling other properites
hd.FilterInternals='SpecifyPrecision';
```

See Also

`dfilt` | `dfilt.dfasymfir` | `dfilt.dffirt` | `dfilt.dfsymfir`

Introduced in R2011a

dfilt.dffirt

Discrete-time, direct-form FIR transposed filter

Syntax

Refer to `dfilt.dffirt` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.dffirt(b)` returns a discrete-time, direct-form FIR transposed filter object `hd`, with numerator coefficients `b`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

- To change to fixed-point filtering, enter

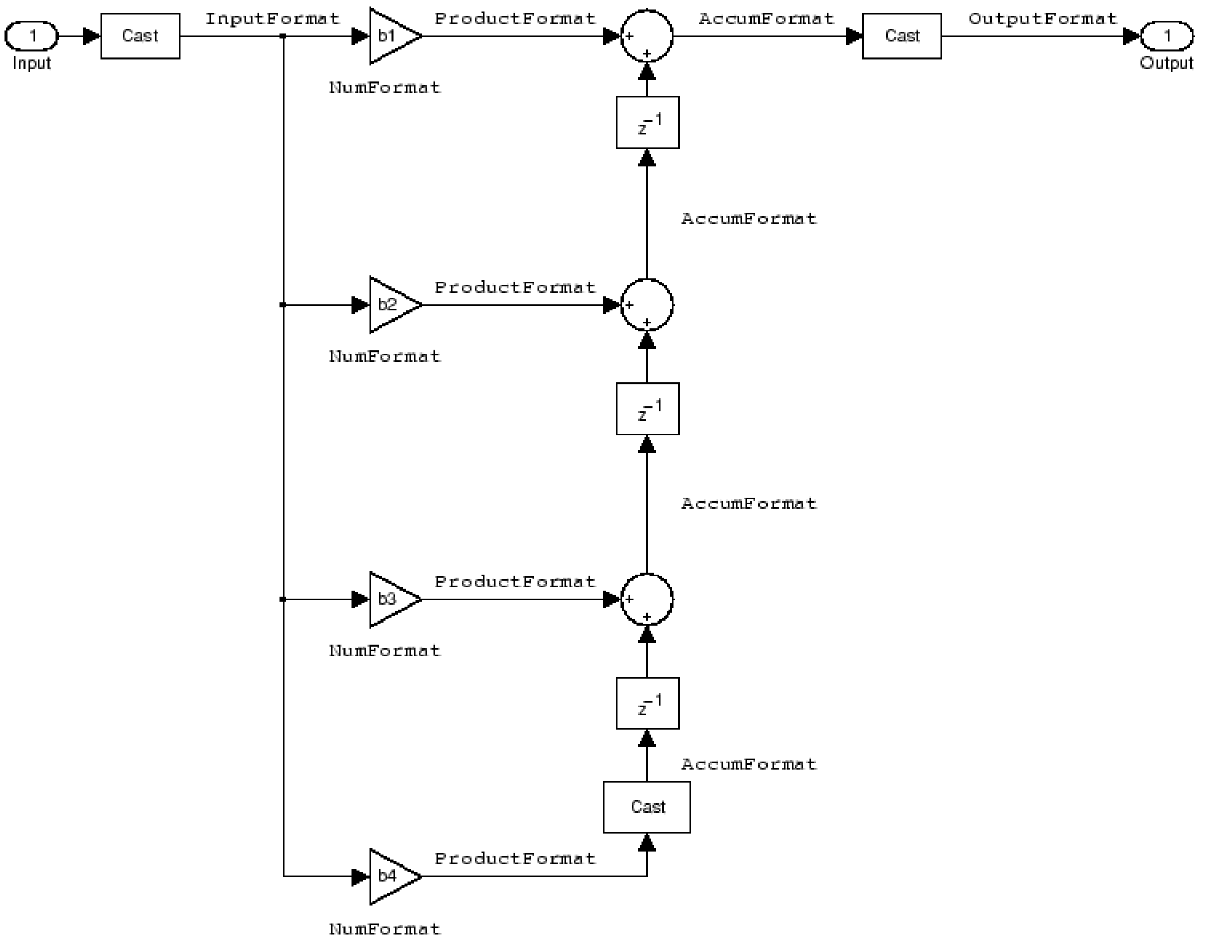
```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 6-18.

`hd = dfilt.dffirt` returns a default, discrete-time, direct-form FIR transposed filter object `hd`, with `b` = 1. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the transposed direct-form FIR filter implemented by `dfilt.dffirt`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word “format.” In this use, “format” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	`AccumWordLength`	`AccumFracLength`	None
InputFormat	`InputWordLength`	`InputFracLength`	None
NumFormat	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `Signed`, `Numerator`
OutputFormat	`OutputWordLength`	`OutputFracLength`	None
ProductFormat	`ProductWordLength`	`ProductFracLength`	None

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties `ProductFracLength` and `ProductWordLength` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with the transposed direct-form FIR implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where hd is a filter.

For further information about the properties of this filter or any dfilt object, refer to "Fixed-Point Filter Properties" on page 6-2.

Name	Values	Description
AccumFracLength	Any positive or negative integer number of bits [30]	Specifies the fraction length used to interpret data output by the accumulator.
AccumWordLength	Any integer number of bits[34]	Sets the word length used to store data in the accumulator.
Arithmetic	fixed for fixed-point filters	Setting this to fixed allows you to modify other filter properties to customize your fixed-point filter.
CoeffAutoScale	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength property value to specify the precision used.
CoeffWordLength	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.

Name	Values	Description
`InputWordLength`	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
`NumFracLength`	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
`OutputFracLength`	Any positive or negative integer number of bits [30]	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `FilerInternals` to `SpecifyPrecision`.
`OutputWordLength`	Any integer number of bits [34]	Determines the word length used for the output data. You make this property editable by setting `FilterInternals` to `SpecifyPrecision`.
`OverflowMode`	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.

<table>
<tr><th>Name</th><th>Values</th><th>Description</th></tr>
<tr><td><code>RoundMode</code></td><td><code>[convergent]</code>, <code>ceil</code>, <code>fix</code>, <code>floor</code>, <code>nearest</code>, <code>round</code></td><td>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).
• <code>ceil</code> - Round toward positive infinity.
• <code>convergent</code> - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.
• <code>fix</code> - Round toward zero.
• <code>floor</code> - Round toward negative infinity.
• <code>nearest</code> - Round toward nearest. Ties round toward positive infinity.
• <code>round</code> - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.
The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.</td></tr>
<tr><td><code>Signed</code></td><td>[true], false</td><td>Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.</td></tr>
<tr><td><code>States</code></td><td><code>fi</code> object to match the filter arithmetic setting</td><td>Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use <code>fi</code> objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation.</td></tr>
</table>

Examples

Specify a second-order direct-form FIR transposed filter structure for a `dfilt` object, `hd`, with the following code:

```
b = [0.05 0.9 0.05];
hd = dfilt.dffirt(b);
set(hd,'arithmetic','fixed')
```

See Also

`dfilt` | `dfilt.dffir` | `dfilt.dfasymfir` | `dfilt.dfsymfir`

Introduced in R2011a

dfilt.dfsymfir

Discrete-time, direct-form symmetric FIR filter

Syntax

Refer to `dfilt.dfsymfir` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.dfsymfir(b)` returns a discrete-time, direct-form symmetric FIR filter object `hd`, with numerator coefficients `b`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

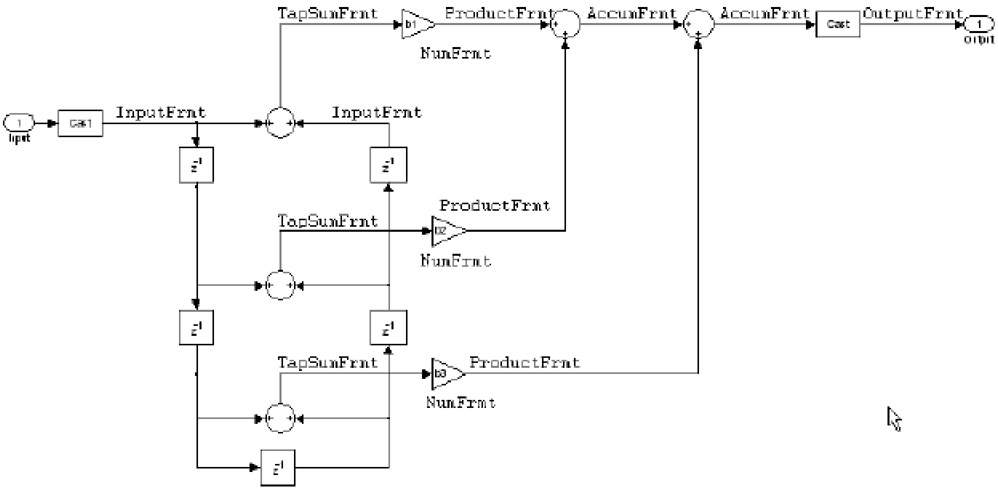
For more information about the property `Arithmetic`, refer to "Arithmetic" on page 6-18.

`hd = dfilt.dfsymfir` returns a default, discrete-time, direct-form symmetric FIR filter object `hd`, with `b=1`. This filter passes the input through to the output unchanged.

Note Only the coefficients in the first half of vector `b` are used because `dfilt.dfsymfir` assumes the coefficients in the second half are symmetric to those in the first half. In the following figure, for example, b(3) = 0, b(4) = b(2) and b(5) = b(1).

Fixed-Point Filter Structure

In the following figure you see the signal flow diagram for the symmetric FIR filter that `dfilt.dfsymfir` implements.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters“frmt” (format). In this use, “frmt” indicates the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFrmt	`AccumWordLength`	`AccumFracLength`	None

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
InputFrmt	`InputWordLength`	`InputFracLength`	None
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `Signed`, `Numerator`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	None
ProductFrmt	`ProductWordLength`	`ProductFracLength`	None
TapSumFrmt	`InputWordLength`	`InputFracLength`	None

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFrmt, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFrmt refers to the properties `ProductFracLength` and `ProductWordLength` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with the symmetric FIR implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to “Fixed-Point Filter Properties” on page 6-2.

Name	Values	Description
AccumFracLength	Any positive or negative integer number of bits [27]	Specifies the fraction length used to interpret data output by the accumulator.
AccumWordLength	Any integer number of bits[33]	Sets the word length used to store data in the accumulator.
Arithmetic	fixed for fixed-point filters	Setting this to fixed allows you to modify other filter properties to customize your fixed-point filter.
CoeffAutoScale	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength property value to specify the precision used.
CoeffWordLength	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.

Name	Values	Description
OutputFracLength	Any positive or negative integer number of bits [29]	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilerInternals to SpecifyPrecision.
OutputWordLength	Any integer number of bits [33]	Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductFracLength	Any positive or negative integer number of bits [29]	Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
ProductWordLength	Any integer number of bits [33]	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.

Name	Values	Description
`RoundMode`	[`convergent`], `ceil`, `fix`, `floor`, `nearest`, `round`	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `ceil` - Round toward positive infinity. • `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • `fix` - Round toward zero. • `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	`fi` object to match the filter arithmetic setting	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation.

Examples

Odd Order

Specify a fifth-order direct-form symmetric FIR filter structure for a `dfilt` object `hd`, with the following code:

```
b = [-0.008 0.06 0.44 0.44 0.06 -0.008];
hd = dfilt.dfsymfir(b);
% Create fixed-point filter
set(hd,'arithmetic','fixed')
% Change FilterInternals property to
% SpecifyPrecision
hd.Filterinternals='SpecifyPrecision';
```

Even Order

Specify a fourth-order, fixed-point, direct-form symmetric FIR filter structure for a `dfilt` object `hd`, with the following code:

```
b = [-0.01 0.1 0.8 0.1 -0.01];
hd = dfilt.dfsymfir(b);
set(hd,'arithmetic','fixed');
```

See Also

`dfilt` | `dfilt.dfasymfir` | `dfilt.dffir` | `dfilt.dffirt`

Introduced in R2011a

dfilt.farrowfd

Fractional Delay Farrow filter

Syntax

```
Hd = dfilt.farrowfd(D, COEFFS)
```

Description

`Hd = dfilt.farrowfd(D, COEFFS)` Constructs a discrete-time fractional delay Farrow filter with COEFFS coefficients and D delay.

Examples

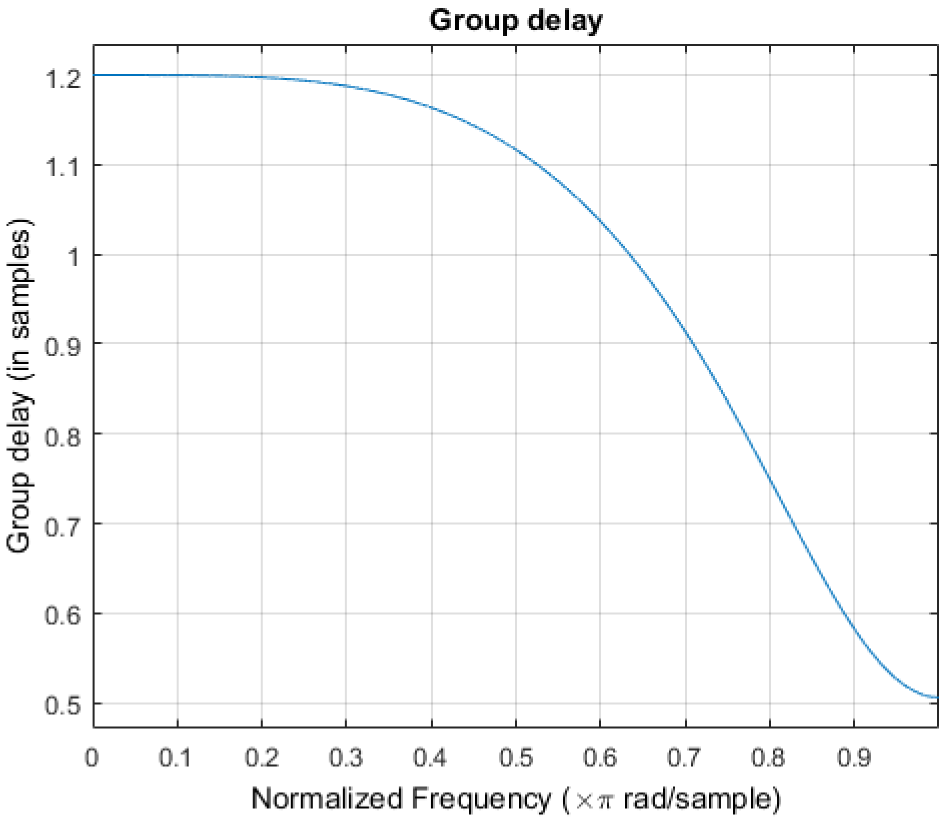
Design a Fractional Delay Farrow Filter

Farrow filters can be designed with the `dfilt.farrowfd` filter designer.

```
coeffs = [-1/6 1/2 -1/3 0;1/2 -1 -1/2 1;
-1/2 1/2 1 0;1/6 0 -1/6 0];
farrow = dfilt.farrowfd(0.5, coeffs);
```

Design a cubic fractional delay filter with the Lagrange method.

```
fdelay = .2; % Fractional delay
d = fdesign.fracdelay(fdelay,'N',3);
cubicfarrow = design(d, 'lagrange', 'FilterStructure', 'farrowfd');
fvtool(cubicfarrow, 'Analysis', 'grpdelay');
```



For more information about fractional delay filter implementations, see the "Fractional Delay Filters Using Farrow Structures" example, `farrowdemo`.

See Also

Topics
`dfilt`

Introduced in R2011a

dfilt.farrowlinearfd

Farrow Linear Fractional Delay filter

Syntax

```
Hd = dfilt.farrowlinearfd(D)
```

Description

`Hd = dfilt.farrowlinearfd(D)` Constructs a discrete-time linear fractional delay Farrow filter with the delay D.

Examples

Farrow linear fractional delay filter with 1/2 sample delay:

```
 Hd = dfilt.farrowlinearfd(0.5);
x = cos(pi/10*(0:159));
y = filter(Hd,x);
stem(x(1:40));
axis([0 40 -2 2]);
hold on;
stem(y(1:40),'color',[1 0 0],'markerfacecolor',[1 0 0 ]);
legend('Original Signal','Filtered Signal','Location','best');
```

For more information about fractional delay filter implementations, see the “Fractional Delay Filters Using Farrow Structures” example, farrowdemo.

See Also

Topics

dfilt

Introduced in R2011a

dfilt.fftfir

Discrete-time, overlap-add, FIR filter

Syntax

```
Hd = dfilt.fftfir(b,len)
Hd = dfilt.fftfir(b)
Hd = dfilt.fftfir
```

Description

This object uses the overlap-add method of block FIR filtering, which is very efficient for streaming data.

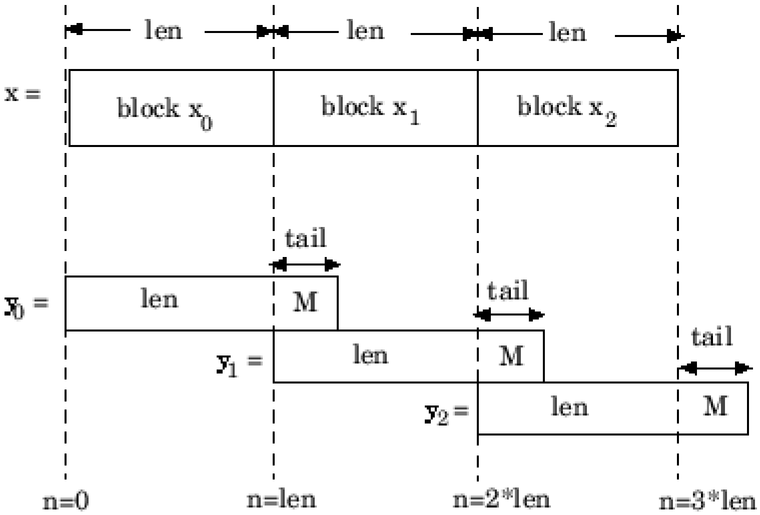
`Hd = dfilt.fftfir(b,len)` returns a discrete-time, FFT, FIR filter, `Hd`, with numerator coefficients, `b` and block length, `len`. The block length is the number of input points to use for each overlap-add computation.

`Hd = dfilt.fftfir(b)` returns a discrete-time, FFT, FIR filter, `Hd`, with numerator coefficients, `b` and block length, `len`=100.

`Hd = dfilt.fftfir` returns a default, discrete-time, FFT, FIR filter, `Hd`, with the numerator `b=1` and block length, `len`=100. This filter passes the input through to the output unchanged.

Note When you use a `dfilt.fftfir` object to filter, the input signal length must be an integer multiple of the object's block length, `len`. The resulting number of FFT points = (filter length + the block length - 1). The filter is most efficient if the number of FFT points is a power of 2.

The `fftfir` uses an overlap-add block processing algorithm, which is represented as follows,



where `len` is the block length and `M` is the length of the numerator-1, (`length(b)-1`), which is also the number of states. The output of each convolution is a block that is longer than the input block by a tail of (`length(b)-1`) samples. These tails overlap the next block and are added to it. The states reported by `dfilt.fftfir` are the tails of the final convolution.

Examples

Create an FFT FIR discrete-time filter with coefficients from a 30^{th} order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.fftfir(b);
% To obtain frequency domain coefficients
% used in filtering
Coeffs = fftcoeffs(Hd);
```

See Also

`dfilt` | `dfilt.dffir` | `dfilt.dfasymfir` | `dfilt.dffirt` | `dfilt.dfsymfir`

Introduced in R2011a

dfilt.latticeallpass

Discrete-time, lattice allpass filter

Syntax

Refer to `dfilt.latticeallpass` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.latticeallpass(k)` returns a discrete-time, lattice allpass filter object `hd`, with lattice coefficients, `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

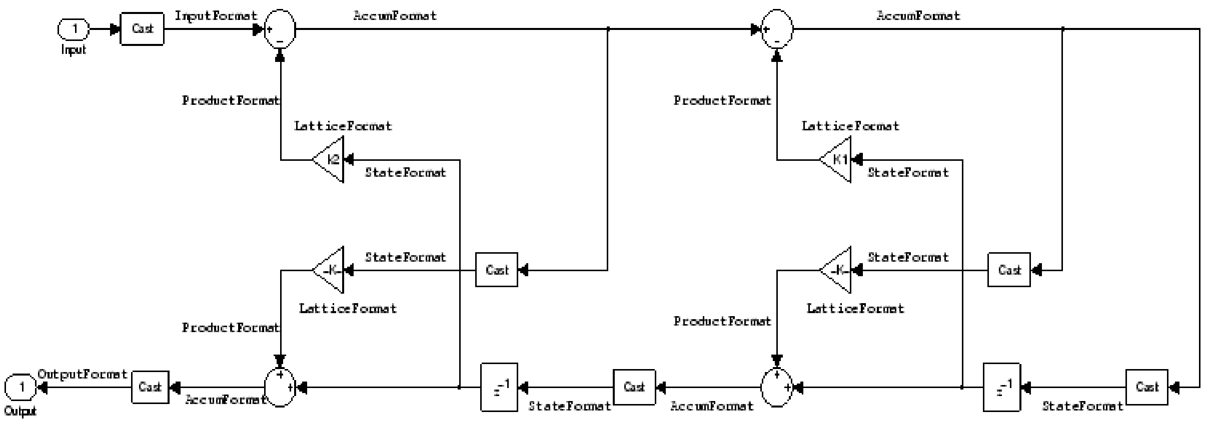
  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to “Arithmetic” on page 6-18.

`hd = dfilt.latticeallpass` returns a default, discrete-time, lattice allpass filter object `hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the allpass lattice filter implemented by `dfilt.latticeallpass`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word “format.” In this use, “format” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	`AccumWordLength`	`AccumFracLength`	`AccumMode`
InputFormat	`InputWordLength`	`InputFracLength`	None
LatticeFormat	`CoeffWordLength`	`LatticeFracLength`	`CoeffAutoScale`
OutputFormat	`OutputWordLength`	`OutputFracLength`	`OutputMode`

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
ProductFormat	`ProductWordLength`	`ProductFracLength`	`ProductMode`
StateFormat	`StateWordLength`	`StateFracLength`	`States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties `ProductFracLength`, `ProductWordLength`, and `ProductMode` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with the allpass lattice implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 6-2.

Property Name	Brief Description
`AccumFracLength`	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties

Property Name	Brief Description
	—DenAccumFracLength and NumAccumFracLength — that let you set the precision for numerator and denominator operations separately.
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/ buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the LatticeFracLength property value to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
Lattice	Any lattice structure coefficients. No default value.
LatticeFracLength	Sets the fraction length applied to the lattice coefficients.

Property Name	Brief Description
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductFracLength	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.

Property Name	Brief Description
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to `filtstates` in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Specify a third-order lattice allpass filter structure for a `dfilt` object `hd`, with the following code:

```
k = [.66 .7 .44];
hd=dfilt.latticeallpass(k);
% convert to fixed-point arithmetic
hd.arithmetic = 'fixed';
```

See Also

`dfilt` | `dfilt.latticear` | `dfilt.latticearma` | `dfilt.latticemamax` | `dfilt.latticemamin`

Introduced in R2011a

dfilt.latticear

Discrete-time, lattice, autoregressive filter

Syntax

Refer to `dfilt.latticear` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.latticear(k)` returns a discrete-time, lattice autoregressive filter object `hd`, with lattice coefficients, `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

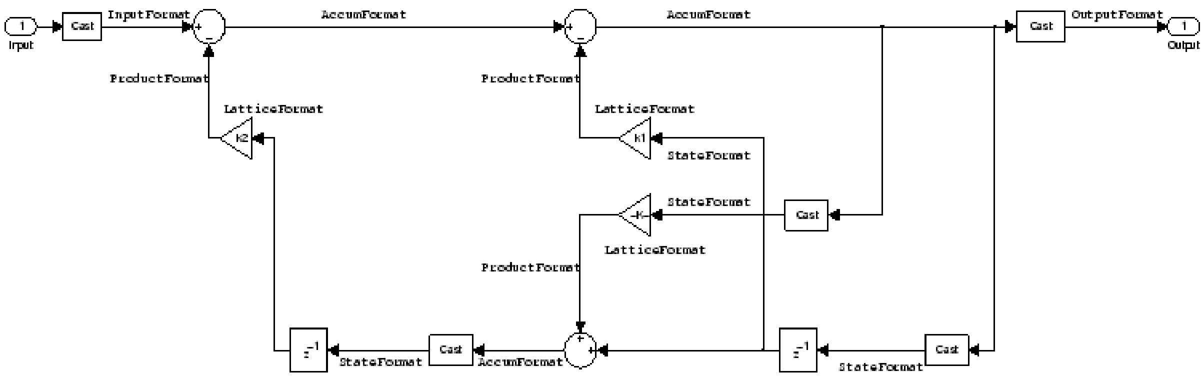
  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 6-18.

`hd = dfilt.latticear` returns a default, discrete-time, lattice autoregressive filter object `hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the autoregressive lattice filter implemented by `dfilt.latticear`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	`AccumWordLength`	`AccumFracLength`	`AccumMode`
InputFormat	`InputWordLength`	`InputFracLength`	None
LatticeFormat	`CoeffWordLength`	`LatticeFracLength`	`CoeffAutoScale`
OutputFormat	`OutputWordLength`	`OutputFracLength`	`OutputMode`
ProductFormat	`ProductWordLength`	`ProductFracLength`	`ProductMode`

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
StateFormat	`StateWordLength`	`StateFracLength`	`States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties `ProductFracLength`, `ProductWordLength`, and `ProductMode` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with the autoregressive lattice implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 6-2.

Property Name	Brief Description
`AccumFracLength`	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — `DenAccumFracLength` and `NumAccumFracLength` — that

Property Name	Brief Description
	let you set the precision for numerator and denominator operations separately.
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/ buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the LatticeFracLength to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering —gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
Lattice	Any lattice structure coefficients.
LatticeFracLength	Sets the fraction length applied to the lattice coefficients.

Property Name	Brief Description
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
ProductFracLength	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
`ProductWordLength`	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
`PersistentMemory`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.
`RoundMode`	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `ceil` - Round toward positive infinity. • `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • `fix` - Round toward zero. • `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`StateFracLength`	When you set `StateAutoScale` to `false`, you enable the `StateFracLength` property that lets you set the fraction length applied to interpret the filter states.

Property Name	Brief Description
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Specify a third-order lattice autoregressive filter structure for a dfilt object, hd, with the following code that creates a fixed-point filter.

```
k = [.66 .7 .44];
hd1=dfilt.latticear(k);
hd1.arithmetic='fixed';
specifyall(hd1);
```

See Also

dfilt | dfilt.latticeallpass | dfilt.latticearma | dfilt.latticemamax | dfilt.latticemamin

Introduced in R2011a

dfilt.latticearma

Discrete-time, lattice, autoregressive, moving-average filter

Syntax

Refer to `dfilt.latticearma` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.latticearma(k)` returns a discrete-time, lattice moving-average autoregressive filter object `hd`, with lattice coefficients, `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

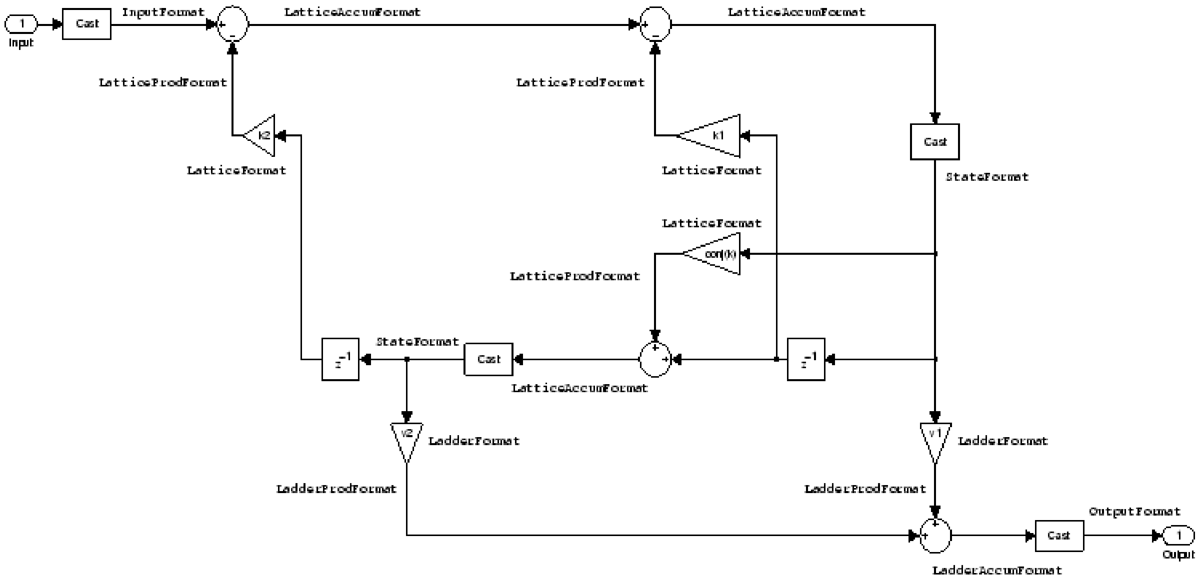
  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 6-18.

`hd dfilt.latticearma` returns a default, discrete-time, lattice moving-average, autoregressive filter object `hd`, with `k` = [ ]. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the autoregressive lattice filter implemented by `dfilt.latticearma`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word “format.” In this use, “format” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
InputFormat	`InputWordLength`	`InputFracLength`	None

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
LadderAccumFormat	`AccumWordLength`	`LadderAccumFracLength`	`AccumMode`
LadderFormat	`CoeffWordLength`	`LadderFracLength`	`CoeffAutoScale`
LadderProdFormat	`ProductWordLength`	`LadderProdFracLength`	`ProductMode`
LatticeAccumFormat	`AccumWordLength`	`LatticeAccum-FracLength`	`AccumMode`
LatticeFormat	`CoeffWordLength`	`LatticeFracLength`	`CoeffAutoScale`
LatticeProdFormat	`ProductWordLength`	`LatticeProdFracLength`	`ProductMode`
OutputFormat	`OutputWordLength`	`OutputFracLength`	`OutputMode`
StateFormat	`StateWordLength`	`StateFracLength`	`States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label LatticeProdFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that lattice coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the LatticeProdFormat refers to the properties `ProductWordLength`, `LatticeProdFracLength`, and `ProductMode` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with the autoregressive moving-average lattice implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 6-2.

Property Name	Brief Description
`AccumFracLength`	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — `DenAccumFracLength` and `NumAccumFracLength` — that let you set the precision for numerator and denominator operations separately.
`AccumMode`	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
`AccumWordLength`	Sets the word length used to store data in the accumulator/buffer.
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `LatticeFracLength` property to specify the precision used.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations

Property Name	Brief Description
	during filtering—gains, delays, sums, products, and input/output.
`InputFracLength`	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Specifies the word length applied to interpret input data.
`Ladder`	Stores the ladder coefficients for lattice ARMA (`dfilt.latticearma`) filters.
`LadderAccumFracLength`	Sets the fraction length used to interpret the output from sum operations that include the ladder coefficients. You can change this property value after you set `AccumMode` to `SpecifyPrecision`.
`LadderFracLength`	Determines the precision used to represent the ladder coefficients in ARMA lattice filters.
`Lattice`	Stores the lattice structure coefficients.
`LatticeFracLength`	Sets the fraction length applied to the lattice coefficients.
`OutputFracLength`	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `OutputMode` to `SpecifyPrecision`.
`OutputMode`	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • `AvoidOverflow` — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • `BestPrecision` — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • `SpecifyPrecision` — lets you set the word and fraction lengths used by the output data from filtering.
`OutputWordLength`	Determines the word length used for the output data.

Property Name	Brief Description
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductFracLength	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

See Also

`dfilt` | `dfilt.latticeallpass` | `dfilt.latticear` | `dfilt.latticemamin` | `dfilt.latticemamin`

Introduced in R2011a

dfilt.latticemamax

Discrete-time, lattice, moving-average filter with maximum phase

Syntax

Refer to `dfilt.latticemamax` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.latticemamax(k)` returns a discrete-time, lattice, moving-average filter object `hd`, with lattice coefficients `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

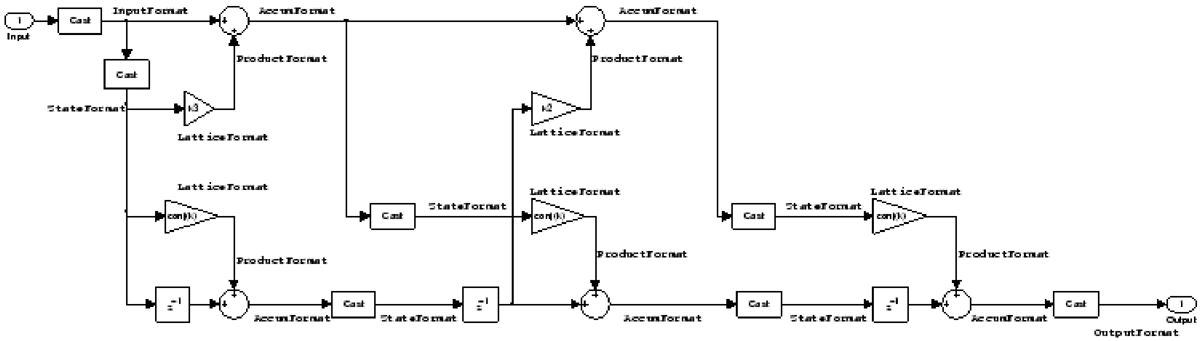
For more information about the property `Arithmetic`, refer to "Arithmetic" on page 6-18.

Note When the `k` coefficients define a maximum phase filter, the resulting filter in this structure is maximum phase. When your coefficients do not define a maximum phase filter, placing them in this structure does not produce a maximum phase filter.

`hd = dfilt.latticemamax` returns a default discrete-time, lattice, moving-average filter object `hd`, with `k` = [ ]. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the maximum phase implementation of a moving-average lattice filter implemented by `dfilt.latticemamax`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	`AccumWordLength`	`AccumFracLength`	`AccumMode`

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
InputFormat	`InputWordLength`	`InputFracLength`	None
LatticeFormat	`CoeffWordLength`	`LatticeFracLength`	`CoeffAutoScale`
OutputFormat	`OutputWordLength`	`OutputFracLength`	`OutputMode`
ProductFormat	`ProductWordLength`	`ProductFracLength`	`ProductMode`
StateFormat	`StateWordLength`	`StateFracLength`	`States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties `ProductFracLength`, `ProductWordLength`, and `ProductMode` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with the maximum phase, moving average lattice implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 6-2.

Property Name	Brief Description
AccumFracLength	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — DenAccumFracLength and NumAccumFracLength — that let you set the precision for numerator and denominator operations separately.
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the LatticeFracLength property to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering —gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.

Property Name	Brief Description
Lattice	Any lattice structure coefficients.
LatticeFracLength	Sets the fraction length applied to the lattice coefficients.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductFracLength	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Property Name	Brief Description
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

Examples

Specify a fourth-order lattice, moving-average, maximum phase filter structure for a dfilt object, hd, with the following code:

```
k = [.66 .7 .44 .33];
hd = dfilt.latticemamax(k);
```

See Also

dfilt | dfilt.latticeallpass | dfilt.latticear | dfilt.latticearma | dfilt.latticemamin

Introduced in R2011a

dfilt.latticemamin

Discrete-time, lattice, moving-average filter with minimum phase

Syntax

Refer to `dfilt.latticemamin` in Signal Processing Toolbox documentation.

Description

`hd = dfilt.latticemamin(k)` returns a discrete-time, lattice, moving-average, minimum phase, filter object `hd`, with lattice coefficients `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

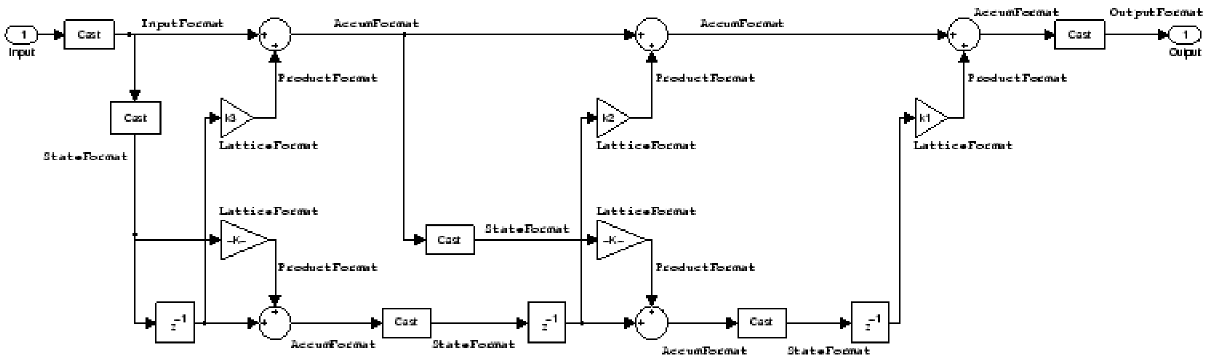
For more information about the property `Arithmetic`, refer to “Arithmetic” on page 6-18.

Note When the `k` coefficients define a minimum phase filter, the resulting filter in this structure is minimum phase. When your coefficients do not define a minimum phase filter, placing them in this structure does not produce a minimum phase filter.

`hd = dfilt.latticemamin` returns a default discrete-time, lattice, moving-average, minimum phase, filter object `hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the minimum phase implementation of a moving-average lattice filter implemented by `dfilt.latticemamin`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.



Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word “format.” In this use, “format” means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	`AccumWordLength`	`AccumFracLength`	`AccumMode`
InputFormat	`InputWordLength`	`InputFracLength`	None
LatticeFormat	`CoeffWordLength`	`LatticeFracLength`	`CoeffAutoScale`
OutputFormat	`OutputWordLength`	`OutputFracLength`	`OutputMode`
ProductFormat	`ProductWordLength`	`ProductFracLength`	`ProductMode`
StateFormat	`StateWordLength`	`StateFracLength`	`States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties `ProductFracLength`, `ProductWordLength`, and `ProductMode` that fully define the coefficient format after multiply (or product) operations.

Properties

In this table you see the properties associated with the minimum phase, moving average lattice implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 6-2.

Property Name	Brief Description
AccumFracLength	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — DenAccumFracLength and NumAccumFracLength — that let you set the precision for numerator and denominator operations separately.
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the LatticeFracLength property to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.

Property Name	Brief Description
Lattice	Any lattice structure coefficients.
LatticeFracLength	Sets the fraction length applied to the lattice coefficients.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
ProductFracLength	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Property Name	Brief Description
`Signed`	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`StateFracLength`	When you set `StateAutoScale` to `false`, you enable the `StateFracLength` property that lets you set the fraction length applied to interpret the filter states.
`States`	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to `filtstates` in Signal Processing Toolbox documentation or in the Help system.
`StateWordLength`	Sets the word length used to represent the filter states.

Examples

Specify a third-order lattice, moving-average, minimum phase, filter structure for a `dfilt` object, `hd`, with the following code:

```
k = [.66 .7 .44];
hd = dfilt.latticemamin(k);
```

See Also

`dfilt` | `dfilt.latticeallpass` | `dfilt.latticear` | `dfilt.latticearma` | `dfilt.latticemamax`

Introduced in R2011a

dfilt.parallel

Discrete-time, parallel structure filter

Syntax

Refer to `dfilt.parallel` in Signal Processing Toolbox documentation.

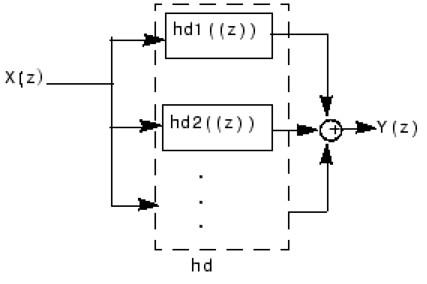
Description

`hd = dfilt.parallel(hd1,hd2,...)` returns a discrete-time filter object `hd`, which is a structure of two or more `dfilt` filter objects, `hd1`, `hd2`, and so on arranged in parallel.

You can also use the standard notation to combine filters into a parallel structure.

```
parallel(hd1,hd2,...)
```

In this syntax, `hd1`, `hd2`, and so on can be a mix of `dfilt` objects and other filtering objects.



`hd1`, `hd2`, and so on can be fixed-point filters. All filters in the parallel structure must be the same arithmetic format — double, single, or fixed. `hd`, the filter returned, inherits the format of the individual filters.

See Also

`dfilt` | `dfilt.cascade` | `dfilt.cascade` | `dfilt.parallel`

Introduced in R2011a

dfilt.scalar

Discrete-time, scalar filter

Syntax

Refer to `dfilt.scalar` in Signal Processing Toolbox documentation.

Description

`dfilt.scalar(g)` returns a discrete-time, scalar filter object with gain `g`, where `g` is a scalar.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to “Arithmetic” on page 6-18.

`dfilt.scalar` returns a default, discrete-time scalar gain filter object `hd`, with gain `1`.

Properties

In this table you see the properties associated with the scalar implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 6-2.

Property Name	Brief Description
`Arithmetic`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
`CoeffAutoScale`	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `CoeffFracLength` property to specify the precision used.
`CoeffFracLength`	Set the fraction length the filter uses to interpret coefficients. `CoeffFracLength` is always available, but it is read-only until you set `CoeffAutoScale` to `false`.
`CoeffWordLength`	Specifies the word length to apply to filter coefficients.
`FilterStructure`	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
`Gain`	Returns the gain for the scalar filter. Scalar filters do not alter the input data except by adding gain.
`InputFracLength`	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Specifies the word length applied to interpret input data.
`OutputFracLength`	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `OutputMode` to `SpecifyPrecision`.
`OutputMode`	Sets the mode the filter uses to scale the filtered data for output. You have the following choices:

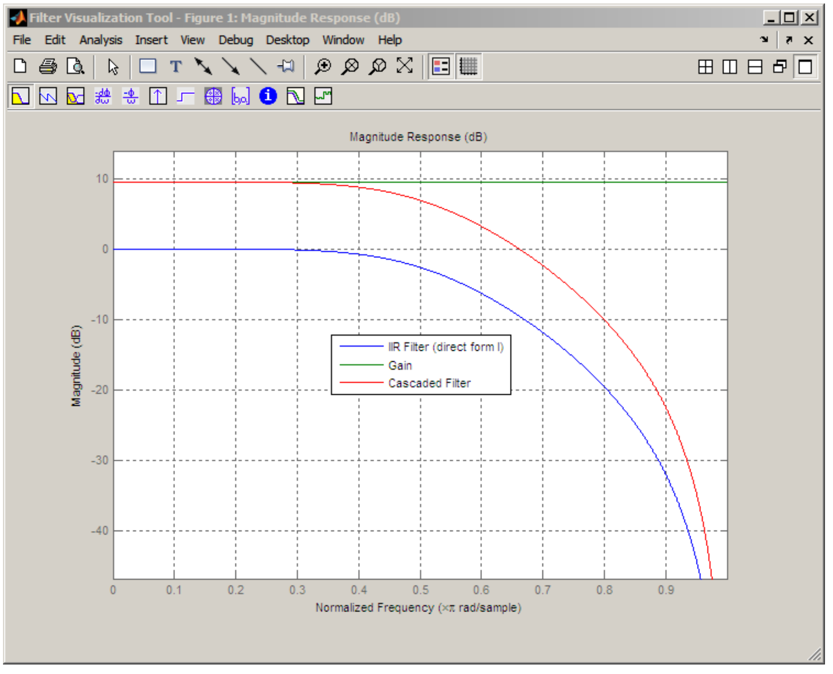
Property Name	Brief Description
	• AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.

Property Name	Brief Description
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.

Examples

Create a direct-form I filter object hd_filt and a scalar object with a gain of 3 hd_gain and cascade them together.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd_filt = dfilt.df1(b,a);
hd_gain = dfilt.scalar(3);
hd_cascade=cascade(hd_gain,hd_filt);
fvtool_handle = fvtool(hd_filt,hd_gain,hd_cascade);
legend(fvtool_handle,'IIR Filter (direct form I)',...
'Gain','Cascaded Filter');
```



To view the stages of the cascaded filter, use

```
hd.Stage(1)
```

and

```
hd.Stage(2)
```

See Also

dfilt | dfilt.cascade

Introduced in R2011a

dfilt.wdfallpass

Wave digital allpass filter

Syntax

```
hd = dfilt.wdfallpass(c)
```

Description

`hd = dfilt.wdfallpass(c)` constructs an allpass wave digital filter structure given the allpass coefficients in vector `c`.

Vector `c` must have, one, two, or four elements (filter coefficients). Filters with three coefficients are not supported. When you use `c` with four coefficients, the first and third coefficients must be 0.

Given the coefficients in `c`, the transfer function for the wave digital allpass filter is defined by

$$H(z) = \frac{c(n) + c(n-1)z^{-1} + \ldots + z^{-n}}{1 + c(1)z^{-1} + \ldots + c(n)z^{-n}}$$

Internally, the allpass coefficients are converted to wave digital filters for filtering. Note that `dfilt.wdfallpass` allows only stable filters. Also note that the leading coefficient in the denominator, a 1, does not need to be included in vector `c`.

Use the constructor `dfilt.cascadewdfallpass` to cascade `wdfallpass` filters.

To compare these filters to other similar filters, `dfilt.wdfallpass` and `dfilt.cascadewdfallpass` filters have the same number of multipliers as the non-wave digital filters `dfilt.allpass` and `dfilt.cascadeallpass`. However, the wave digital filters use fewer states and they may require more adders in the filter structure.

Wave digital filters are usually used to create other filters. This toolbox uses them to implement halfband filters, which the first example in Examples demonstrates. They are most often building blocks for filters.

Properties

In the next table, the row entries are the filter properties and a brief description of each property.

Property Name	Brief Description
AllpassCoefficients	Contains the coefficients for the allpass wave digital filter object
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.

Filter Structure

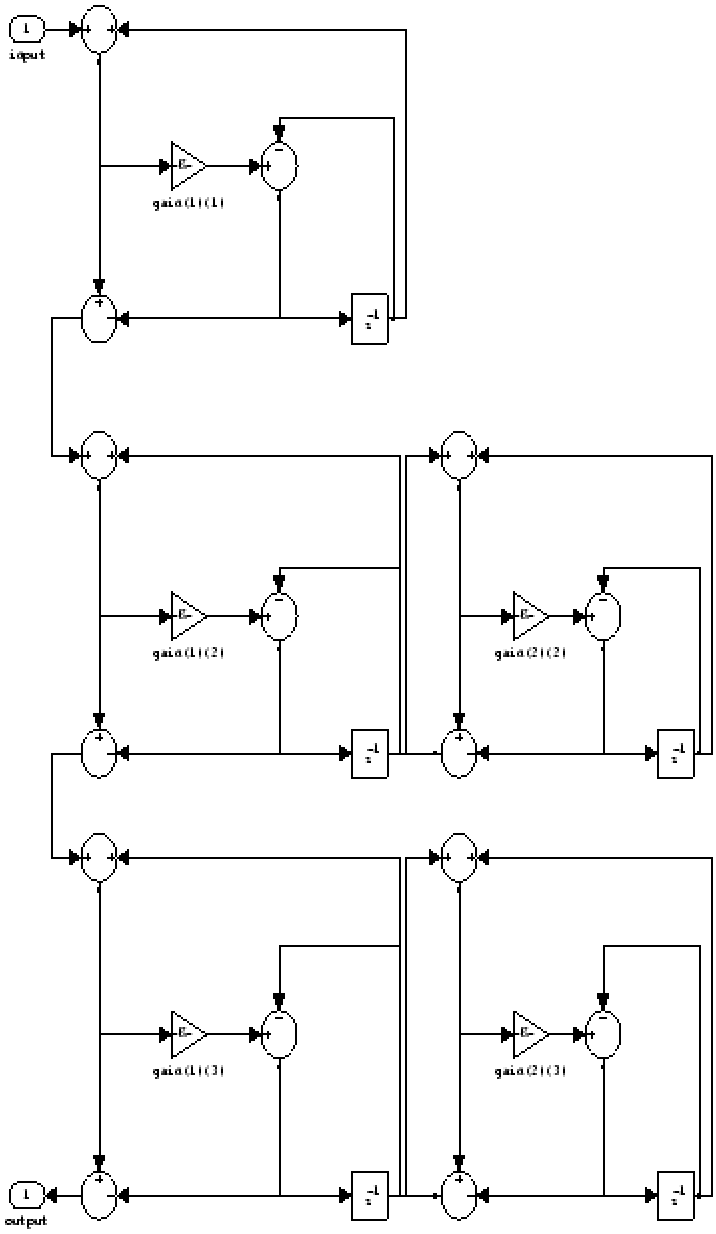
When you change the order of the wave digital filters in the cascade, the filter structure changes as well.

As shown in this example, realizemdl lets you see the filter structure used for your filter, if you have Simulink installed.

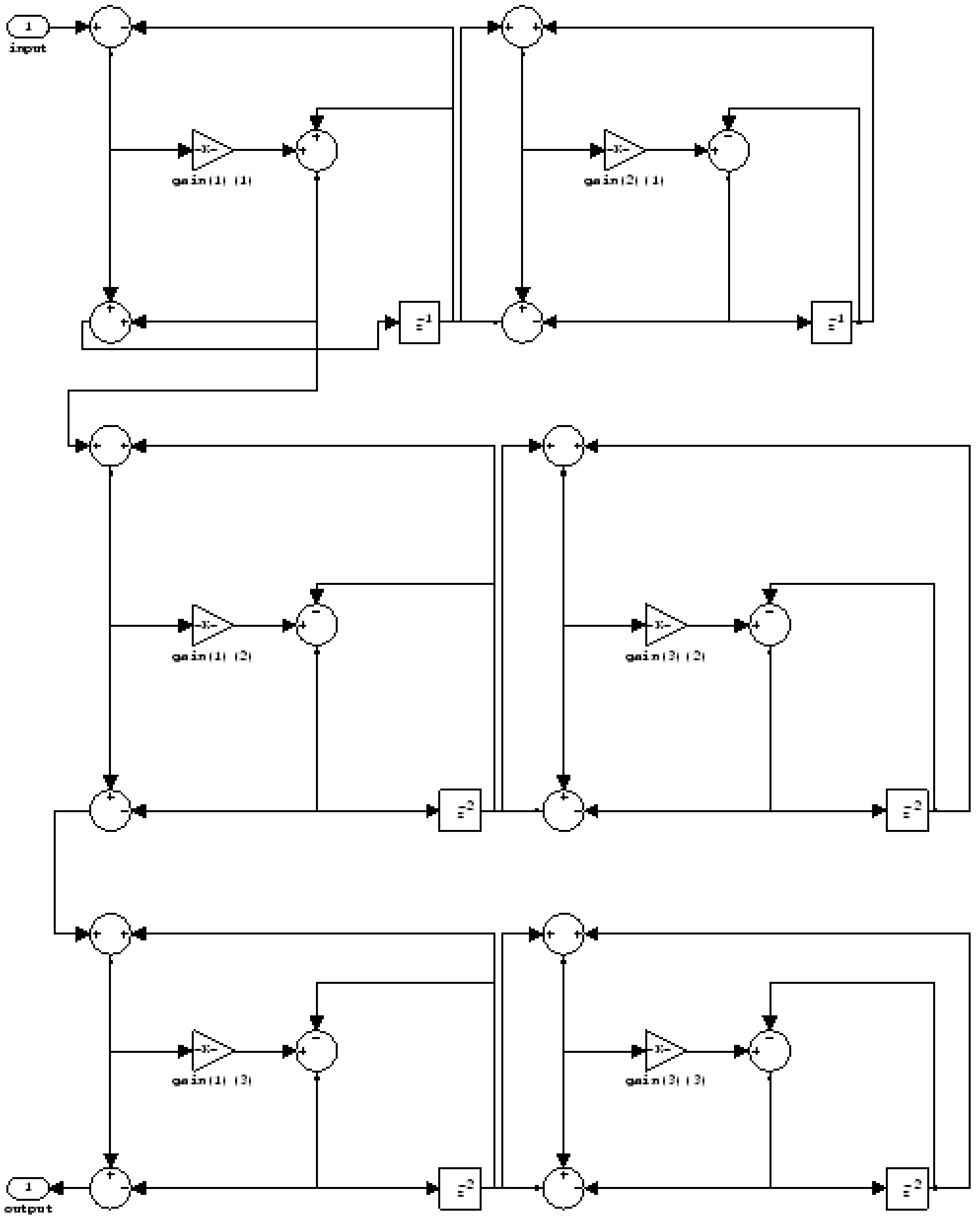
```
section11=0.8;
section12=[1.5,0.7];
section13=[1.8,0.9];
hd1=dfilt.cascadewdfallpass(section11,section12,section13);
```

```
section21=[0.8,0.4];
section22=[0,1.5,0,0.7];
section23=[0,1.8,0,0.9];
hd2=dfilt.cascadewdfallpass(section21,section22,section23);
% If you have Simulink
realizemdl(hd2)
```

hd1 has this filter structure with three sections.


input
gain(1)(1)
gain(1)(2)
gain(2)(2)
gain(1)(3)
gain(2)(3)
output

The filter structure for hd2 is somewhat different, with the different orders and interconnections between the three sections.



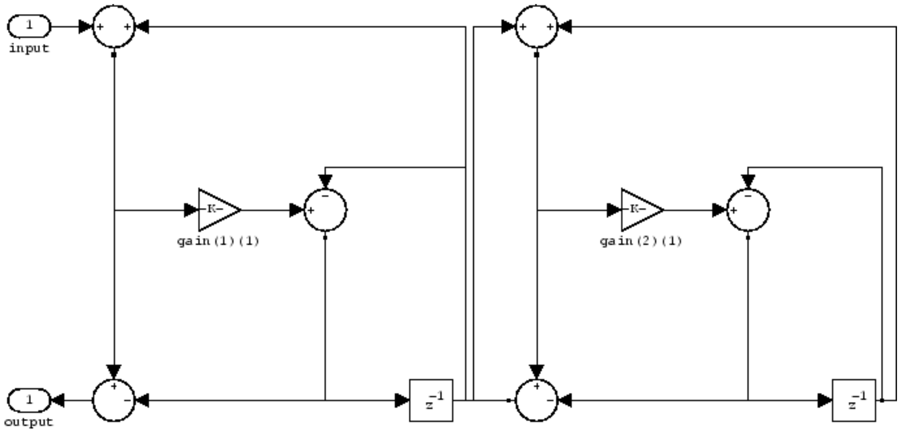
1
input
gain(1)(1)
gain(2)(1)
gain(1)(2)
gain(3)(2)
gain(1)(3)
gain(3)(3)
1
output

Examples

Construct a second-order wave digital allpass filter with two coefficients. Note that to use `realizemdl`, you must have Simulink.

```
c = [1.5,0.7];
hd = dfilt.wdfallpass(c);
```

With Simulink installed, `realizemdl` returns this structure for `hd`.



See Also

`dfilt` | `dfilt.allpass` | dsp.CICInterpolator | `dfilt.latticeallpass` | `dfilt.cascadewdfallpass` | `dfilt.cascadeallpass` | dsp.IIRHalfbandDecimator

Introduced in R2011a

disp

Filter properties and values

Syntax

```
disp(h)
```

Description

`disp(h)` lists the property names and property values of a filter object. In all ways, it is the same as leaving the semicolon off an on the command line, except that `disp` does not display the variable name.

Examples

Display the Properties of a Filter Object

```
FIRFilt = dsp.FIRFilter;
disp(FIRFilt);
```

```
  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: [0.5000 0.5000]
    InitialConditions: 0

  Use get to show all properties
```

See Also

`set`

Introduced in R2011a

double

Cast fixed-point filter to use double-precision arithmetic

Syntax

```
hd = double(h)
```

Description

`hd = double(h)` returns a new filter `hd` that has the same structure and coefficients as `h`, but whose arithmetic property is set to `double` to use double-precision arithmetic for filtering. `double(h)` is not the same as the `reffilter(h)` function:

- `hd`, the filter returned by `double` has the quantized coefficients of `h` represented in double-precision floating-point format
- The reference filter returned by `reffilter` has double-precision, floating-point coefficients that have not been quantized.

You might find `double(h)` useful to isolate the effects of quantizing the coefficients of a filter by using `double` to create a filter `hd` that operates in double-precision but uses the quantized filter coefficients.

Examples

Compare Fixed-point Output with Floating-Point Output

Construct a Lowpass Filter

```
h = dfilt.dffir(firgr(27,[0 .4 .6 1],...
[1 1 0 0]));
```

Set h to use fixed-point arithmetic to filter. Quantize the coeffs.

```
h.arithmetic = 'fixed';
```

Cast h to double-precision

```
hd = double(h);
```

Set up an input signal.

```
n = 0:99; x = sin(0.7*pi*n(:));
```

Fixed-point output.

```
y = filter(h,x);
```

Floating-point output.

```
yd = filter(hd,x);
```

Compare the Fixed-point output with Floating-point output

```
FixedFloatNormDiff=norm(yd-double(y),inf)
```

```
FixedFloatNormDiff =

   2.1234e-05
```

See Also

reffilter

Introduced in R2011a

dsp.util.getLogsArray

Package: dsp.util

Return logged signal as MATLAB array

Syntax

```
Array = dsp.util.getLogsArray(LogObject,Format2D,'SignalPath',PATH)
Array = dsp.util.getLogsArray(LogObject,Format2D,'SignalName',NAME)
```

Description

`Array = dsp.util.getLogsArray(LogObject,Format2D,'SignalPath',PATH)` returns a MATLAB array that contains a signal in `LogObject`. You must specify the `PATH` to the signal in `LogObject` using the `Name,Value` pair argument.

`Array = dsp.util.getLogsArray(LogObject,Format2D,'SignalName',NAME)` returns a MATLAB array that contains a signal in `LogObject`. You must specify the `NAME` of the signal in `LogObject` using the `Name,Value` pair argument.

Input Arguments

`LogObject`

Specify the name of the object that contains your logged signals. Valid classes for `LogObject` depend on the syntax you use:

- When you specify `PATH` as a dsp.util.SignalPath object, `LogObject` can be either a Simulink.SimulationData.Dataset or `Simulink.SimulationData.Signal` object.
- When you specify `PATH` as the full path to a block in a Simulink model, `LogObject` must be a `Simulink.SimulationData.Dataset` object.
- When you specify the `NAME` of a signal in `LogObject`, `LogObject` can be an object of class timeseries, `Simulink.SimulationData.Dataset`, or `Simulink.SimulationData.Signal`.

`Format2D`

Specify a logical value to determine whether the function formats 3-D logged signals as a 2-D or 3-D MATLAB array. When you set this property to `true`, the function uses the following formula to format the 3-D logged signal into a 2-D MATLAB array:

```
dim = size(signal);
ntimes = dim(1)*dim(3);
Array = reshape(permute(signal,[1 3 2]),[ntimes dim(2)]);
```

When you set this property to `false`, the function returns the logged signal without any reformatting.

`PATH`

Specify the path to the logged signal in `LogObject`. You can specify the path using a dsp.util.SignalPath object, or you can provide the full path to a block in your Simulink model. To get the full path to a block in your Simulink model, use the `gcb` command.

`NAME`

Specify the name of the signal in `LogObject`.

Output Arguments

`Array`

The output `Array` is a MATLAB array that contains the specified logged signal. When the input is a 3-D logged signal, the dimensions of `Array` depend on the value you specify for `Format2D`:

- When `Format2D` is `true`, `Array` is a 2-D MATLAB array.
- When `Format2D` is `false`, `Array` is a 3-D MATLAB array.

When the input is not a 3-D signal, the dimensions of the output `Array` match those of the input.

Examples

Note: To run the following examples, you must first load ex_logsout.mat which contains a `Simulink.SimulationData.Dataset` object. Alternatively, you can open and simulate the ex_log_utils Simulink model. Doing so will log signals and generate the necessary `ex_logsout` object.

Extract a unique signal named Signal3x4 from ex_logsout.

```
dsp.util.getLogsArray(ex_logsout, true, 'SignalName','Signal3x4')
```

Extract a unique signal named Signal3x4 from ex_logsout as a 3-D array.

```
dsp.util.getLogsArray(ex_logsout, false, 'SignalName','Signal3x4')
```

Find and extract a specific signal from multiple signals that have the same name.

Because `ex_logsout` contains multiple signals named `Signal2x4`, you must use the `dsp.util.getSignalPath` function to find the paths to each of those signals.

```
paths = dsp.util.getSignalPath(ex_logsout, 'Signal2x4')
% paths is a 2x1 array of dsp.util.SignalPath objects. Next, examine
% the BlockPath property of each paths object.
paths.BlockPath
% Find the signal path that corresponds to the logged signal you are
% interested in. For example paths(2). You can then use the
% dsp.util.getLogsArray function and provide the 'SignalPath' name-value
% pair argument.
dsp.util.getLogsArray(ex_logsout, true, 'SignalPath', paths(2))
```

Find and extract a signal from a bus.

Use the `dsp.util.getSignalPath` function to get paths to all the signals in the bus named `Bus1`.

```
buspaths = dsp.util.getSignalPath(ex_logsout, 'Bus1')
% buspaths is a 2x1 array of dsp.util.SignalPath objects. Examine the
% BusElement property of each buspaths object.
buspaths.BusElement
% Select a signal path. For example buspaths(1). This is the path to the
% signal named 'Signal3x4' in bus 'Bus' that is contained in bus 'Bus1'.
% Now that you have the path to the signal, call dsp.util.getLogsArray
```

```
% using the 'SignalPath' name-value pair argument.
dsp.util.getLogsArray(ex_logsout, true, 'SignalPath', buspaths(1))
```

See Also

`dsp.util.getSignalPath` | dsp.util.SignalPath |
`Simulink.SimulationData.BlockPath` | Simulink.SimulationData.Dataset |
`Simulink.SimulationData.Signal` | timeseries

Topics

"Export Signal Data Using Signal Logging" (Simulink)
"Configure a Signal for Logging" (Simulink)
"Migrate Scripts That Use ModelDataLogs API" (Simulink)

Introduced in R2011b

dsp.util.getSignalPath

Package: dsp.util

Paths to logged signals

Syntax

```
Path = dsp.util.getSignalPath(LogObject, SignalName)
```

Description

`Path = dsp.util.getSignalPath(LogObject, SignalName)` returns all paths to signals in `LogObject` with name `SignalName`. The output `Path` is a dsp.util.SignalPath object or an array of `dsp.util.SignalPath` objects.

Input Arguments

`LogObject`

Specify the name of the object that contains your logged signals. `LogObject` must be a Simulink.SimulationData.Dataset or `Simulink.SimulationData.Signal` object.

`SignalName`

Specify the name of a logged signal in `LogObject`.

Output Arguments

`Path`

The output `Path` contains the path to all signals named `SignalName` in `LogObject`.

- If `LogObject` contains a unique signal with name `SignalName`, the function returns a single dsp.util.SignalPath object.

- If `LogObject` contains more than one signal with the specified name, the function returns an array of dsp.util.SignalPath objects.

Examples

Note: To run the following examples, you must first load ex_logsout.mat which contains a `Simulink.SimulationData.Dataset` object. Alternatively, you can open and simulate the ex_log_utils Simulink model. Doing so will log signals and generate the necessary `ex_logsout` object.

Find and extract a specific signal from multiple signals that have the same name.

Because `ex_logsout` contains multiple signals named `Signal2x4`, you must use the `dsp.util.getSignalPath` function to find the paths to each of those signals.

```
paths = dsp.util.getSignalPath(ex_logsout, 'Signal2x4')
% paths is a 2x1 array of dsp.util.SignalPath objects. Next, examine
% the BlockPath property of each paths object.
paths.BlockPath
% Find the signal path that corresponds to the logged signal you are
% interested in. For example paths(2). You can then use the
% dsp.util.getLogsArray function and provide the 'SignalPath' name-value
% pair argument.
dsp.util.getLogsArray(ex_logsout, true, 'SignalPath', paths(2))
```

Find and extract a signal from a bus.

Use the `dsp.util.getSignalPath` function to get paths to all the signals in the bus named `Bus1`.

```
buspaths = dsp.util.getSignalPath(ex_logsout, 'Bus1')
% buspaths is a 2x1 array of dsp.util.SignalPath objects. Examine the
% BusElement property of each buspaths object.
buspaths.BusElement
% Select a signal path. For example buspaths(1). This is the path to the
% signal named 'Signal3x4' in bus 'Bus' that is contained in bus 'Bus1'.
% Now that you have the path to the signal, call dsp.util.getLogsArray
% using the 'SignalPath' name-value pair argument.
dsp.util.getLogsArray(ex_logsout, true, 'SignalPath', buspaths(1))
```

Tips

- To return the path to an unnamed signal in `LogObject`, set `SignalName` to the empty string (`''`).

See Also

`dsp.util.getLogsArray` | dsp.util.SignalPath | Simulink.SimulationData.Dataset | `Simulink.SimulationData.Signal` | `Simulink.SimulationData.updateDatasetFormatLogging`

Topics

"Export Signal Data Using Signal Logging" (Simulink)
"Configure a Signal for Logging" (Simulink)
"Migrate Scripts That Use ModelDataLogs API" (Simulink)

Introduced in R2011b

dsp.util.SignalPath class

Package: dsp.util

Properties of paths to signals

Description

`dsp.util.SignalPath` objects contain path information for signals in Simulink.SimulationData.Dataset objects. You get `Simulink.SimulationData.Dataset` objects when you use `Dataset` logging to log signals from a Simulink model.

Construction

`dsp.util.SignalPath` objects are returned by the `dsp.util.getSignalPath` function and can be used as input to the `dsp.util.getLogsArray` function.

Properties

`SignalName`

Contains the name of the signal output by the block at the specified `BlockPath`.

`BlockPath`

Provides the `Simulink.SimulationData.BlockPath` to the block in the Simulink model from which the signal originates.

`PortIndex`

Provides the output port index of the block from which the logged signal `SignalName` originates.

`BusElement`

Provides a string description of a signal in a logged bus. When `SignalPath` is not a logged bus, this property will be an empty string.

If the `SignalPath` object is a logged bus signal, the `BusElement` string will be formatted as follows:

- When the bus contains a nonbus signal, `BusElement` is the name of that signal.
- When the bus contains a nested bus that contains a nonbus signal, `BusElement` will be a dot-separated string consisting of the name of the nested bus followed by the name of the non-bus signal. For example: `nestbus.signal1`
- When the bus contains nested busses within nested busses to any depth, `BusElement` will be a dot-separated string. This string contains each of the nested bus names, and ends with the nonbus signal name. For example: `outernestedbus.innernestedbus.signal1`

See Also

`dsp.util.getLogsArray` | `dsp.util.getSignalPath` | `Simulink.SimulationData.BlockPath` | Simulink.SimulationData.Dataset

dsp_links

Identify whether blocks in model are current, deprecated, or obsolete

Syntax

```
dsp_links
dsp_links('modelname')
```

Description

`dsp_links` returns a structure with three elements that identify whether the DSP System Toolbox blocks in the current model are current, deprecated, or obsolete. Each element is one of the three block categories and contains a cell array of character vectors. Each character vector is the name of a library block in the current model.

`dsp_links('modelname')` returns the three-element structure for the specified model.

Examples

Name of the First Current Block

Display block support information for the specified model, and then find the name of the first current block.

```
sys = 'dspcochlear';
```

Load the dspcochlear model

```
load_system(sys)
```

Run dsp_links on the model

```
links = dsp_links(sys)
```

```
links =
```

```
  struct with fields:

      obsolete: {}
    deprecated: {}
       current: {1×23 cell}
```

Find the name of the first current block

```
links.current{1}
```

```
ans =

    'dspcochlear/Filter bank signal processing/Subsystem3/IIR Filter Bank/Digital Filte
```

Definitions

Obsolete Blocks

Obsolete blocks are blocks that the toolbox no longer supports. In some cases, these blocks no longer function properly.

Deprecated Blocks

Deprecated blocks are blocks that the toolbox still supports but are likely to become obsolete in a future release. Refer to the block reference page for suggested replacements.

Current Blocks

Current blocks are blocks that the toolbox supports and that represent the latest block functionality.

See Also

`liblinks`

Introduced before R2006a

dsplib

Open top-level DSP System Toolbox library

Syntax

```
dsplib
```

Description

`dsplib` opens the top-level DSP System Toolbox block library model.

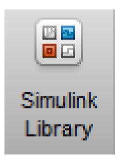
Examples

View and gain access to the DSP System Toolbox blocks:

```
dsplib
```

Alternatives

To view and gain access to the DSP System Toolbox blocks using the Simulink library browser:

- Type `simulink` at the MATLAB command line, and then expand the DSP System Toolbox node in the library browser.
- Click the Simulink icon 

from the MATLAB Toolstrip.

Introduced before R2006a

dspstartup

Configure Simulink environment for signal processing systems

Note: `dspstartup` will be removed in a future release. Use DSP Simulink model templates instead. For more information on DSP Simulink model templates, see Configure the Simulink Environment for Signal Processing Models.

Syntax

`dspstartup`

Description

`dspstartup` configures Simulink environment parameters with settings appropriate for a typical signal processing project. You can use the `dspstartup` function in the following ways:

- At the MATLAB command line. Doing so configures the Simulink environment in your current session for signal processing projects.
- By adding a call to the `dspstartup` function from your `startup.m` file. When you do so, MATLAB configures your Simulink environment for typical signal processing projects each time you launch MATLAB.

When the function successfully configures your Simulink environment, MATLAB displays the following message in the command window.

```
Changed default Simulink settings for signal processing
systems (dspstartup.m).
```

The `dspstartup.m` file executes the following commands.

```
set_param(0, ...
          'SingleTaskRateTransMsg','error', ...
          'multiTaskRateTransMsg', 'error', ...
          'Solver',                'fixedstepdiscrete', ...
```

```
'EnableMultiTasking',    'Off', ...
'StartTime',             '0.0', ...
'StopTime',              'inf', ...
'FixedStep',             'auto', ...
'SaveTime',              'off', ...
'SaveOutput',            'off', ...
'AlgebraicLoopMsg',      'error', ...
'SignalLogging',         'off');
```

Examples

Add a call to the `dspstartup` function from your `startup.m` file:

1. To find out if there is a `startup.m` file on your MATLAB path, run the following code at the MATLAB command line:

   ```
   which startup.m
   ```

2. If MATLAB returns `'startup.m' not found`, see "Specify Startup Options" (MATLAB) to learn more about the `startup.m` file.

   If MATLAB returns a path to your `startup.m` file, open that file for editing.

   ```
   edit startup.m
   ```

3. Add a call to the `dspstartup` function. Your `startup.m` file now resembles the following code sample:

   ```
   %STARTUP   Startup file
   %   This file is executed when MATLAB starts up, if it exists
   %   anywhere on the path.  In this case, the startup.m file
   %   runs the dspstartup.m file to configure the Simulink
   %   environment with settings appropriate for typical
   %   signal processing projects.

   dspstartup;
   ```

See Also

`startup`

Introduced before R2006a

dspunfold

Generates a multi-threaded MEX file from a MATLAB function

Syntax

```
dspunfold file
dspunfold options file
```

Description

`dspunfold file` generates a multi-threaded MEX file from the entry-point MATLAB function specified by `file`, using the DSP unfolding technology. DSP unfolding is a technique to improve throughput through parallelization. The multi-threaded MEX file leverages the multicore CPU architecture of the host computer and can improve speed significantly. In addition to the multi-threaded MEX file, the function generates a single-threaded MEX file, a self-diagnostic analyzer function, and the corresponding help files.

`dspunfold options file` generates a multi-threaded MEX file from the entry-point MATLAB function specified by `file`, using the function arguments specified by `options`.

Note: This function requires a MATLAB Coder license.

Examples

- "Multi-Threaded MEX File Generation Using DSP Unfolding"

Input Arguments

`options` — Function parameters

option value pairs

Option	Values	Description	Examples
`-args arguments`	Cell array	Argument types for the entry-point	The number of elements in the cell array must be the same as

Option	Values	Description	Examples
		MATLAB function, specified as a cell array. The cell array accepts numeric elements, the `coder.typeof` function, and the `coder.Constant` function. The generated multi-threaded MEX file is specialized to the size, class, and complexity of `arguments`.	the number of arguments that the entry-point MATLAB function expects. • `dspunfold fcn -args {ones(10,1), 5}` `dspunfold` extracts the type (size, class, and complexity) information from the elements in the `arguments` cell array. `fcn` is the entry-point MATLAB function. • `dspunfold fcn -args {coder.typeof(ones(10,1)), coder.typeof(5)}` `coder.typeof` is used to specify the types of the `fcn` arguments. • `dspunfold fcn -args {coder.Constant(ones(10,1)), coder.Constant(5)}` • `dspunfold fcn -args {}` By default, `arguments` is `{}`. An empty cell array `{}` indicates that `fcn` accepts no input arguments.

Option	Values	Description	Examples

`-o output`	Charact vector	Name of the output multi-threaded MEX file, specified as a character vector. If no `output` name is specified, the name of the generated multi-threaded MEX file is inherited from the input MATLAB function with an `'_mt'` suffix. `dspunfold` also adds a platform-specific extension to this name. In addition, `dspunfold` generates a single-threaded MEX file with an `'_st'` suffix, and a test bench file with an `'_analyzer'` suffix.	• No `output` name specified
			`dspunfold fcn`
			Files generated: `fcn_mt.mexw64`, `fcn_st.mexw64`, `fcn_analyzer.p`
			• `output` name specified
			`dspunfold fcn -o foo`
			Files generated: `foo.mexw64`, `foo_st.mexw64`, `foo_analyzer.p`

Option	Values	Description	Examples
`-s statelength`	Scalar integer greater than or equal to zero `auto`	State length of the algorithm in the entry-point MATLAB function, specified as a scalar integer greater than or equal to zero, or `auto`. By default, the `statelength` is `zero` frames, indicating that the algorithm is stateless. If at least one entry of `frameinputs` is `true`, `statelength` is considered in samples. For information on frames and samples, see "Sample- and Frame-Based Concepts" `-s auto` triggers automatic state length detection. In this mode, you must provide numeric inputs to the `arguments` cell array. These inputs detect the	• `dspunfold fcn -args {randn(10,1), randn(10,1), randn(10,1)} -s 3 -f [false, false, false]` State length is `three` frames. • `dspunfold fcn -args {randn(10,1), randn(10,1), randn(10,1)} -s 3 -f [true, false, false]` State length is `three` samples. State length is considered in samples, because at least one entry of the `-f` option is `true`. • `dspunfold fcn -args {randn(10,1), randn(10,1), randn(10,1)} -s auto` Automatic state length detection is invoked. • `dspunfold fcn -args {coder.typeof (randn(10,1)), coder.typeof(randn(10,1)), coder.typeof(randn(10,1))} -s auto` generates this error message: `The input argument 1 is of type coder.PrimitiveType which is not supported when using -s auto`

Option	Values	Description	Examples
		state length of the algorithm. You can input `coder.Constant` but not `coder.typeof`. When automatic state length detection is invoked, it is recommended that you provide random inputs to the `arguments` array. See "Automatic State Length Detection" on page 5-478	

Option	Values	Description	Examples
`-f frameinputs`	scalar logical vector of logical values	Frame status of input arguments for the entry-point MATLAB function, specified as one of `true` or `false`. • `true` — Input is in frames and can be subdivided into samples without changing the system behavior. • `false` — Input cannot be subdivided into samples without changing the system behavior. For example, you cannot subdivide the coefficients of a filter without changing the characteristics of the filter. By default, `frameinputs` is `false`.	• `dspunfold fcn -args {randn(10,1), randn(10,1), randn(10,1)} -s 3 -f true` All the inputs are marked as frames. State length is `three` samples. • `dspunfold fcn -args {randn(10,1), randn(10,1), randn(10,1)} -s 3 -f [true, false, false]` State length is `three` samples. • `dspunfold fcn -args {randn(10,1), randn(10,1), randn(10,1)} -s 3` The default value of `frameinputs` is `false`. State length is `three` frames.

Option	Values	Description	Examples
		`frameinputs` set to a scalar logical value sets the frame status of all the inputs simultaneously. To specify `statelength` in samples, set at least one entry of `frameinputs` to `true`. If `frameinputs` is not specified, the unit of `statelength` is frames.	
`-r repetition`	Positive integer	Repetition factor used to generate the multi-threaded MEX file, specified as a positive integer. The default value of `repetition` is `1`. See “Repetition Factor” on page 5-478.	`dspunfold fcn -args {randn(10,2), randn(20,2), randn(30,3)} -r 2`

Option	Values	Description	Examples
`-t threads`	Positive integer	Number of threads used by the multi-threaded MEX file, specified as a positive integer. The default value of `threads` is the number of physical CPU cores present on your machine. See "Threads" on page 5-478.	`dspunfold fcn -args {randn(10,1), randn(20,2), randn(30,3)} -t 4`
`-v verbose`	Scalar logical	Option to show verbose output during code generation, specified as `true` or `false`. The default is `true`.	• `dspunfold fcn -args {randn(10,1), randn(20,2), randn(30,3)} -v true` • `dspunfold fcn -args {randn(10,1), randn(20,2), randn(30,3)} -v false`

`file` — entry-point MATLAB function
character vector

Entry-point MATLAB function from which `dspunfold` generates the multi-threaded MEX file. The function must support code generation.

Example: `dspunfold fcn -args {randn(10,1),randn(10,2),randn(20,1)}`

`fcn` is the entry-point MATLAB function and `{randn(10,1),randn(10,2),randn(20,1)}` are its input arguments.

Output Files

When you invoke `dspunfold` on an entry-point MATLAB function, `dspunfold` generates the following files.

File	Value	Description	Examples
Multi-threaded MEX file	MEX file	Multi-threaded MEX file generated from the entry-point MATLAB function. The MEX file inherits the output name. If no output name is specified, the name of this file is inherited from the MATLAB function with an '_mt' suffix. A platform-specific extension is also added to the name.	• dspunfold fcn -o foo generates foo.mexw64 • dspunfold fcn generates fcn_mt.mexw64
Help file for the multi-threaded MEX file	MATLA file	MATLAB help file for the multi-threaded MEX file. The help file has the same name as the MEX file, but with an '.m' extension. To invoke the help file, type help <MEX filename> at the MATLAB command prompt. This help file displays information on how to invoke the MEX file, its syntax, latency, and types (size, class, and complexity) of the inputs to the MEX file. In addition, the help file documents the parameters used by dspunfold — Threads, Repetition, and State length. This information is useful when you are invoking the MEX file. The syntax to invoke the MEX file should be the same as the syntax shown in the help file.	• help foo • help fcn_mt

File	Value	Description	Examples
Single-threaded MEX file	MEX file	Single-threaded MEX file generated from the entry-point MATLAB function. The MEX file inherits the `output` name with an `'_st'` suffix. If no `output` name is specified, the name of this file is inherited from the MATLAB function with an `'_st'` suffix. A platform-specific extension is also added to the name. Use this file as a benchmark to compare against the speed of the multi-threaded MEX file.	• `dspunfold fcn -o foo` generates `foo_st.mexw64` • `dspunfold fcn` generates `fcn_st.mexw64`
Help file for the single-threaded MEX file	MATLAB file	MATLAB help file for the single-threaded MEX file. The help file has the same name as the MEX file, but with an '.m' extension. To invoke the help file, type `help <MEX filename>` at the MATLAB command prompt. The help file displays information on how to invoke the MEX file, its syntax, and types (size, class, and complexity) of the inputs to the MEX file. The syntax to invoke the MEX file should be the same as the syntax shown in the help file.	• `help foo_st` • `help fcn_st`

File	Value	Description	Examples
Self-diagnostic analyzer function	P-coded file	`report = function_analyzer (input 1, input 2,...input n)` measures the difference in speed between the multi-threaded MEX file and the single-threaded MEX file. This file verifies that the output values match. `report = function_analyzer('late` reports the latency of the multi-threaded MEX file introduced by DSP unfolding. `report` contains the following fields: • Latency — The value of the latency (in frames) • Speedup — The speedup difference between the multi-threaded MEX file and single-threaded MEX file. If you specified `latency` option, the value of this field is empty `[]`. • Pass — Logical value that shows if the outputs match between the generated multi-threaded MEX file and the single-threaded MEX file. If you specified `latency` option,	• Multiple frames with different values are specified along the first dimension Example 1: `report = foo_analyzer(randn(10*2,1), randn(20*2,2), randn(30*3,3))` Example 2: `report = foo_analyzer([randn(10,1);randn [randn(20,1);randn(20,1)], [randn(30,1);randn(30,1);randn` • `report = foo_analyzer('latency')`

File	Value	Description	Examples
		the value of this field is empty `[]`. The first dimension of the analyzer inputs must be a multiple of the first dimension of the corresponding inputs, given to the `-args` option. The other dimensions must match exactly. The analyzer inherits the `output` name with an `'_analyzer'` suffix. If no `output` name is specified, the name of this file is inherited from the MATLAB function with an `'_analyzer'` suffix.	

File	Value	Description	Examples
Help file for the self-diagnostic analyzer function	MATLA file	Help file for the self-diagnostic analyzer function. The help file has the same name as the MEX file, but with an '.m' extension. To invoke the help file, type `help <function_analyzer>` in MATLAB. The help file for the self-diagnostic analyzer function displays information on how to invoke the analyzer function, its syntax, and types (size, class, and complexity) of the inputs to the analyzer function. The syntax to invoke the analyzer function should be the same as the syntax shown in the help file.	`help foo_analyzer`

Limitations

General Limitations:

- On Windows and Linux, you must use a compiler that supports the Open Multiprocessing (OpenMP) application interface. See `http://www.mathworks.com/support/compilers/current_release/`.
- If the input MATLAB function has runtime errors, the errors are not caught when you run the multi-threaded MEX file. Before you use the `dspunfold` function, call `codegen` on the MATLAB function and make sure that the MEX file is generated successfully.
- If the generated code uses a large amount of memory to store the local variables, around `4 MB` on Windows platform, the generated multi-threaded MEX file can have unexpected behavior. This limit varies with each platform. As a workaround, reduce

the size of the input signals or restructure the MATLAB function to use less local memory.

- `dspunfold` does not support:

  - `varargin` and `varargout` inside the MATLAB function
  - variable-size inputs and outputs
  - P-coded entry-point MATLAB functions
  - cell arrays as inputs and outputs

Analyzer Limitations:

The following limitations apply to the analyzer function generated by the `dspunfold` function. For more information on the analyzer function, see 'Self-Diagnostic Analyzer' in the 'More About' section of `dspunfold`.

- If multiple frames of the analyzer input are identical, the analyzer might throw false positive pass results. It is recommended that you provide at least two different frames for each input of the analyzer.
- If the algorithm in the entry-point MATLAB function chooses its state length based on the input values, the analyzer might provide different *pass* results for different input values. For an example, see the `FIR_Mean` function in “Why Does the Analyzer Choose the Wrong State Length?”.
- If the input to the entry-point MATLAB function does affect the output immediately, the analyzer might throw false positive *pass* results. For an example, see the `Input_Output` function in “Why Does the Analyzer Choose a Zero State Length?”.
- If the output results of the multi-threaded MEX file and single-threaded MEX file match statistically but do not match numerically, the analyzer does not pass. Consider the `FilterNoise` function that follows, which filters a random noise signal with an FIR filter. The function calls `randn` from within itself to generate random noise. Hence, the output results of the `FilterNoise` function match statistically but not match numerically.

  ```
  function Output = FilterNoise(x)

  persistent FIRFilter
  if isempty(FIRFilter)
      FIRFilter = dsp.FIRFilter('Numerator',fir1(12,0.4));
  end
  Output = FIRFilter(x+randn(1000,1));
  ```

```
end
```

When you run the automatic state length detection tool run on `FilterNoise`, the tool detects an infinite state length. Because the tool cannot find a numerical match for a finite state length, it chooses an infinite state length.

```
dspunfold FilterNoise -args {randn(1000,1)} -s auto
```

```
Analyzing input MATLAB function FilterNoise
Creating single-threaded MEX file FilterNoise_st.mexw64
Searching for minimal state length (this might take a while)
Checking stateless ... Insufficient
Checking 1 ... Insufficient
Checking Infinite ... Sufficient
Checking 2 ... Insufficient
Minimal state length is Inf
Creating multi-threaded MEX file FilterNoise_mt.mexw64
Warning: The multi-threading was disabled due to performance considerations. This ha
equal to (Threads-1)*Repetition frames (3 frames in this case).
> In coder.internal.warning (line 8)
  In unfoldingEngine/BuildParallelSolution (line 25)
  In unfoldingEngine/generate (line 207)
  In dspunfold (line 234)
Creating analyzer file FilterNoise_analyzer
```

The algorithm does not need an infinite state. The state length of the FIR filter, hence the algorithm is `12`.

Call `dspunfold` with state length set to 12.

```
dspunfold FilterNoise -args {randn(1000,1)} -s 12 -f true
```

```
Analyzing input MATLAB function FilterNoise
Creating single-threaded MEX file FilterNoise_st.mexw64
Creating multi-threaded MEX file FilterNoise_mt.mexw64
Creating analyzer file FilterNoise_analyzer
```

Run the analyzer function.

```
FilterNoise_analyzer(randn(1000*4,1))
```

```
Analyzing multi-threaded MEX file FilterNoise_mt.mexw64  ...
Latency = 8 frames
Speedup = 0.5x
Warning: The output results of the multi-threaded MEX file FilterNoise_mt.mexw64 do
single-threaded MEX file FilterNoise_st.mexw64. Check that you provided the correct
```

```
when you generated the multi-threaded MEX file FilterNoise_mt.mexw64. For best pract
see the 'Tips' section in the dspunfold function reference page.
> In coder.internal.warning (line 8)
  In FilterNoise_analyzer
ans =
    Latency: 8
    Speedup: 0.4970
       Pass: 0
```

The analyzer looks for a numerical match and fails the verification, even though the generated multi-threaded MEX file is valid.

Speedup Limitations:

- If the entry-point MATLAB function contains code with low complexity, MATLAB overhead or multi-threaded MEX overhead overshadow any performance gains. In such cases, do not use `dspunfold`.
- If the number of operations in the input MATLAB function is small compared to the size of the input or output data, the multi-threaded MEX file does not provide any speedup gain. Sometimes, it can result in a speedup loss, even if the repetition value is increased. In such cases, do not use `dspunfold`.

More About

State Length

State length of the algorithm.

Most of the time, the state length used by `dspunfold` matches the state length of the algorithm in the entry-point MATLAB function. If the algorithm is simple, state length is easy to determine. For example, the state length of an FIR filter is the number of taps in the filter – `1`. In some scenarios, to optimize speedup, `dspunfold` chooses a state length that is different from the algorithm state length or the state length specified using the `-s` option. For example, when the state length is greater than ($threads - 1$) × *repetition* frames, `dspunfold` considers the state length to be infinite. Also, multi-threading gets disabled due to performance considerations.

Automatic State Length Detection

You can automatically detect the minimum state length for which the outputs of the multi-threaded MEX and single-threaded MEX match.

In complex algorithms, it is not easy to determine the state length analytically. In such scenarios, use the analyzer to compute the state length. When you set `-s` to `auto`, `dspunfold` invokes the analyzer. The analyzer computes the outputs for different state lengths and detects the minimum state length for which the outputs of the multi-threaded MEX file and single-threaded MEX file match. The analyzer uses the numeric value of the inputs given to `-args`. To detect the most efficient state length, provide random inputs to `-args`. In this mode, you cannot input `coder.typeof` to `arguments`. Due to the extra analysis this tool requires, the time to generate the MEX file increases.

When you use automatic state length detection on an algorithm with code paths that depend on the input values, use inputs that choose the code path with the longest state length. Also, the inputs must have an immediate effect on the output. If inputs choose a code path that triggers runtime errors, automatic state length detection stops, and so does the analyzer. Make sure that the MATLAB function supports code generation and does not have run-time errors for the inputs under test. Before invoking `dspunfold`, call `codegen` on the entry-point MATLAB function. In addition, simulate the entry-point MATLAB function to make sure it has no run-time errors.

Threads

The `-t` option specifies the number of threads used by the multi-threaded MEX file.

Increasing this value can improve the multi-threaded MEX speedup, at the cost of a larger latency. Decreasing this value reduces the latency and potentially decreases the multi-threaded MEX speedup.

Repetition Factor

Repetition factor is the number of consecutive frames processed by each thread in one processing step.

Increasing this value reduces the overhead per frame of data, potentially improving the speedup at the cost of larger latency. Decreasing this value reduces the latency, and potentially decreases the multi-threaded MEX speedup.

Self-Diagnostic Analyzer

The self-diagnostic analyzer function is a help tool that is generated with the MEX file. This function measures the speedup gain of the multi-threaded MEX file compared to the single-threaded MEX file. The analyzer function also verifies that the outputs of the multi-threaded MEX file and single-threaded MEX file match.

If you specify an incorrect state length value, the outputs usually do not match. To check for the numerical match between the multi-threaded MEX file and the single-threaded MEX file, provide at least two different frames for each input argument of the analyzer. The frames are appended along the first dimension. The analyzer alternates between these frames while verifying that the outputs match. Failure to provide multiple frames for each input can decrease the effectiveness of the analyzer and can lead to false positive verification results. In other words, the analyzer might produce *pass* `= 1` results even when an incorrect state length value is specified. The analyzer alternates through a maximum of 3 × (2 × *threads* × *repetition*) frames. If your algorithm requires more than 3 × (2 × *threads* × *repetition*) frames to verify the results, then the analyzer cannot verify accurately.

Tips

General

- Do not display plots, scopes, or execute other user interface operations from within the multi-threaded MEX file. The generated MEX file can have unexpected behavior.
- Do not use `coder.extrinsic` inside the input MATLAB function. The generated MEX file can have unexpected behavior.

When the state length is less than or equal to (*threads* `- 1`) × *repetition* frames:

- Do not use a random number inside the MATLAB function. The outputs of the single-threaded MEX file and the multi-threaded MEX file might not match. Also, the outputs of the consecutive executions of the multi-threaded MEX file might not match. The analyzer might not pass the numerical match verification.

  It is recommended that you generate the random number outside the entry-point MATLAB function and pass it as an argument to the function.
- Do not use global or persistent variables anywhere other than in the entry-point MATLAB function. For example, avoid using persistent variables in subfunctions. The

generated MEX file can produce inaccurate results. In general, global variables are not recommended.

- Do not access I/O resources from within the multi-threaded MEX file. The generated MEX file can have unexpected behavior. These resources include file writers and readers, UDP sockets, and audio players and recorders.
- Do not use functions with interactive inputs (for example, the keyboard) inside the multi-threaded MEX file. The generated MEX file can have unexpected behavior.

Workflow

- To generate a valid multi-threaded MEX file with the required speedup and latency, follow the "Workflow for Generating a Multi-Threaded MEX File using dspunfold".
- Before using `dspunfold`, call `codegen` on the entry-point MATLAB function and make sure that the function generates a MEX file successfully.
- After generating the multi-threaded MEX file using `dspunfold`, run the analyzer function. Make sure that the analyzer function passes. The exception to this rule is when the algorithm produces results that match statistically, but not numerically. In this exception, the analyzer function does not `pass`, even though the `dspunfold` function generates a valid multi-threaded MEX file. See 'Analyzer Limitations' for an example.
- For help on using the MEX file and analyzer, at the MATLAB command prompt, enter `help <mexfile name>` and `help <analyzer name>`.

State Length

- If you choose a state length that is greater than or equal to the value of the exact state length, the analyzer passes. If the analyzer fails, increase the state length, regenerate the MEX file, and verify again.
- If the state length is greater than `0`, the inputs marked as frames (through `-f` option) must all have the same dimensions.
- When generating the MEX file and running the analyzer, use inputs that invoke the same state length.

Automatic State Length Detection

When you set `-s` to `auto`:

- If the algorithm in the entry-point MATLAB function chooses a code path based on the input values, use inputs that choose the code path with the longest state length.
- Provide random inputs to `-args`.

- Choose inputs that have an immediate effect on the output. See "Why Does the Analyzer Choose a Zero State Length?".

Analyzer

- Make sure the outputs of the multi-threaded MEX file and the single-threaded MEX file do not contain `NaN` or an `Inf`. The analyzer cannot do numeric checks and returns `pass` as `false`. The automatic state length detection tool detects infinite state length and displays a warning

> **Warning:** The output results of the multi-threaded MEX file do not match the output results of the single-threaded MEX file even for Infinite state length. A possible reason is that input MATLAB function generates different output results between consecutive runs even for the same input values.

- Provide multiple frames with different values for each input of the analyzer.
  To improve the analyzer effectiveness, append successive frames along the first dimension.
- Provide inputs to the analyzer that lead to efficient code coverage.

Speedup

- To improve the speedup of the multi-threaded MEX file, specify the exact state length in samples. You can specify the state length in samples by setting at least one entry of `frameinputs` to `true`. The use of samples reduces the overhead and increases the speedup.
- To increase the speedup at the cost of larger latency, you can:
  - Increase the repetition factor. Use the `-r` option.
  - Increase the number of threads. Use the `-t` option.
- For each input that can be divided into samples without altering the algorithm behavior, set frame status to `true` using the `-f` option. The input is then considered in samples, which can increase the speedup of the generated multi-threaded MEX file.

Algorithms

The multi-threaded MEX file buffers multiple-input signal frames into a buffer of 2 × *threads* × *repetition* frames, where *threads* is the number of threads, and *repetition* is the repetition factor. The MEX file processes these frames simultaneously, using multiple cores. This process introduces some deterministic latency, where *latency* = 2 × *threads* × *repetition*. Latency is traded off with the speedup you might gain by increasing the number of threads or the repetition factor.

See Also

See Also

"How Is dspunfold Different from parfor?" | "Workflow for Generating a Multi-Threaded MEX File using dspunfold" | "Why Does the Analyzer Choose the Wrong State Length?" | "Why Does the Analyzer Choose a Zero State Length?" | "MATLAB Algorithm Acceleration" (MATLAB Coder)

Topics

"Multi-Threaded MEX File Generation Using DSP Unfolding"

Introduced in R2015b

ellip

Elliptic filter using specification object

Syntax

```
ellipFilter = design(d,'ellip','SystemObject',true)
ellipFilter = design(d,'ellip',designoption,value,designoption,...
value,...,'SystemObject',true)
```

Description

`ellipFilter = design(d,'ellip','SystemObject',true)` designs an elliptical IIR digital filter using the specifications supplied in the object `d`.

`ellipFilter = design(d,'ellip',designoption,value,designoption,... value,...,'SystemObject',true)` returns an elliptical or Cauer FIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `ellip`, refer to the command line help system. For example, to get specific information about using `ellip` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'ellip')
```

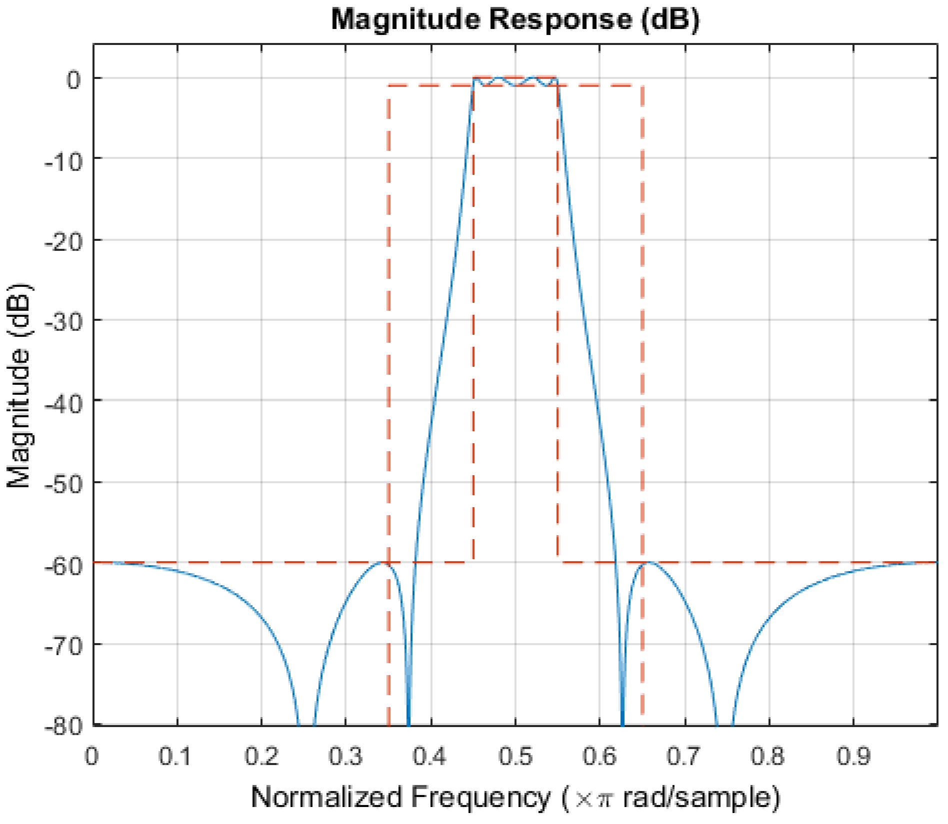
Examples

These examples demonstrate how to use `ellip` to design filters based on filter specification objects.

Design a Bandpass Filter

Construct the default bandpass filter specification object and design an elliptic filter.

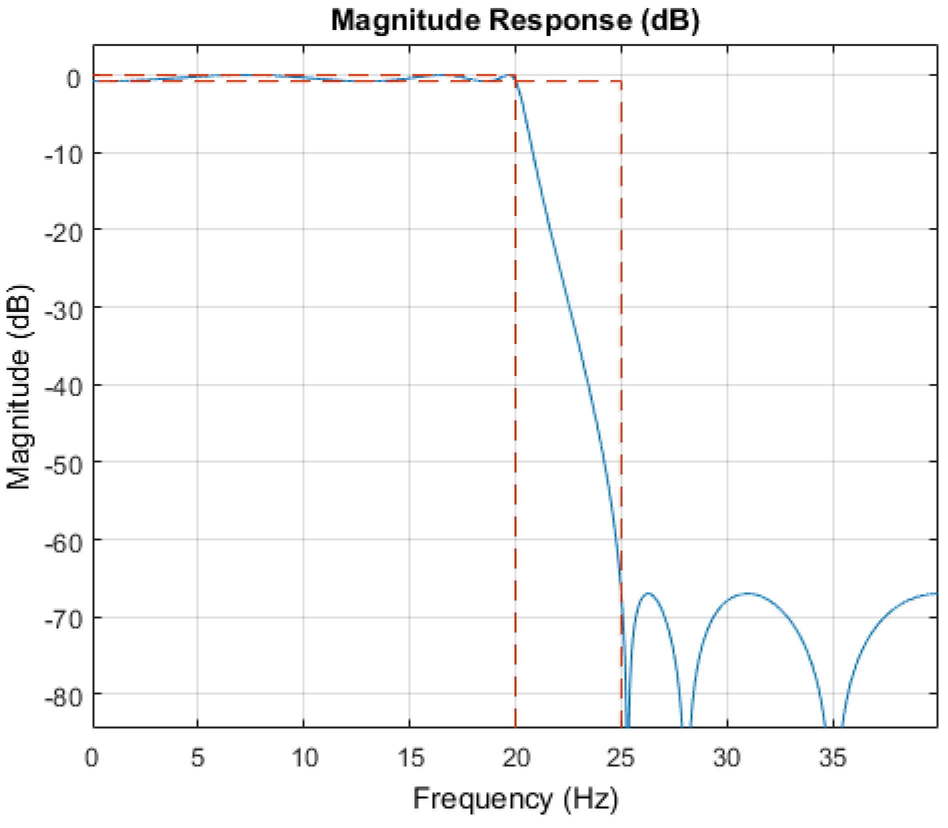
```
d = fdesign.bandpass;
bandpass = design(d,'ellip','matchexactly','both','SystemObject',true);
fvtool(bandpass);
```



Design Lowpass Filter

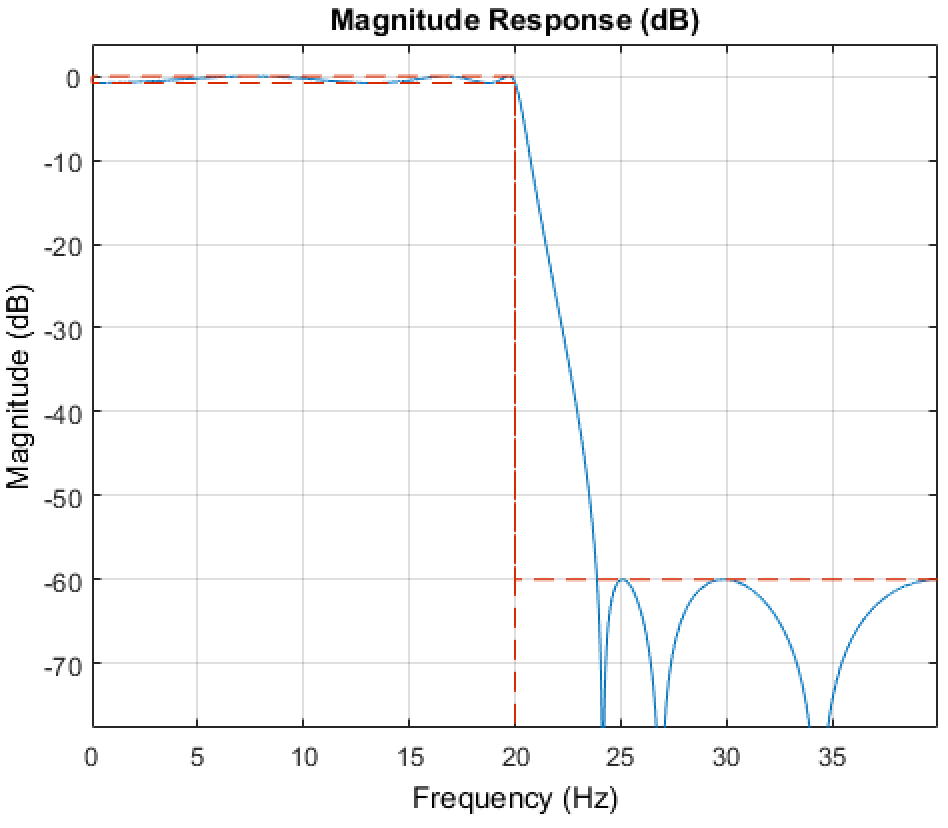
Construct a lowpass object with order, passband-edge frequency, stopband-edge frequency, and passband ripple specifications, and then design an elliptic filter.

```
d = fdesign.lowpass('n,fp,fst,ap',6,20,25,.8,80);
design(d,'ellip','SystemObject',true); % Starts fvtool to display the filter.
```



Construct a lowpass object with filter order, passband edge frequency, passband ripple, and stopband attenuation specifications, and then design an elliptic filter.

```
d = fdesign.lowpass('n,fp,ap,ast',6,20,.8,60,80);
design(d,'ellip','SystemObject',true); % Starts fvtool to display the filter.
```



See Also

`butter` | `cheby1` | `cheby2`

Introduced in R2011a

euclidfactors

Euclid factors for multirate filter

Compatibility

`mfilt` will be removed in a future release. See dsp.CICDecimator, dsp.CICInterpolator, dsp.FIRDecimator, dsp.FIRInterpolator, dsp.FilterCascade, dsp.FarrowRateConverter, dsp.FIRRateConverter, dsp.IIRHalfbandDecimator, or dsp.IIRHalfbandInterpolator instead.

Syntax

```
[lo,mo] = euclidfactors(hm)
```

Description

`[lo,mo] = euclidfactors(hm)` returns integer factors `lo` and `mo` such that (`lo`*L)-(`mo`*M) = -1. L and M are relatively prime and represent the interpolation and decimation factors of the multirate filter `hm`.

`euclidfactors` works with the multirate filter `mfilt.firsc`. You cannot return `lo` and `mo` for decimators or interpolators.

Examples

Use an FIR fractional decimator, with L = 5 and M = 7, to show what `euclidfactors` does.

Indeed, (`lo`*L)-(`mo`*M) = (4*5)-(3*7) = -1.

See Also

`polyphase` | `nstates`

Introduced in R2011a

equiripple

Equiripple single-rate FIR filter from specification object

Syntax

```
equiFilt = design(d,'equiripple','SystemObject',true)
equiFilt =
design(d,'equiripple',designoption,value,...,'SystemObject',true)
```

Description

`equiFilt = design(d,'equiripple','SystemObject',true)` designs an equiripple FIR digital filter using the specifications supplied in the object `d`. Equiripple filter designs minimize the maximum ripple in the passbands and stopbands.

When you use `equiripple` with Nyquist filter specification objects, you might encounter design cases where the filter design does not converge. Convergence errors occur mostly at large filter orders, or small transition widths, or large stopband attenuations. These specifications, alone or combined, can cause design failures. For more information, refer to `fdesign.nyquist` in the online Help system.

`equiFilt =`
`design(d,'equiripple',designoption,value,...,'SystemObject',true)` returns an equiripple FIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `equiripple`, refer to the command line help system. For example, to get specific information about using `equiripple` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'equiripple')
```

Examples

Design a Single-Rate Equiripple Filter

Design a single-rate equiripple filter from a halfband filter specification object. Notice the help command used to learn about the options for the specification object and method.

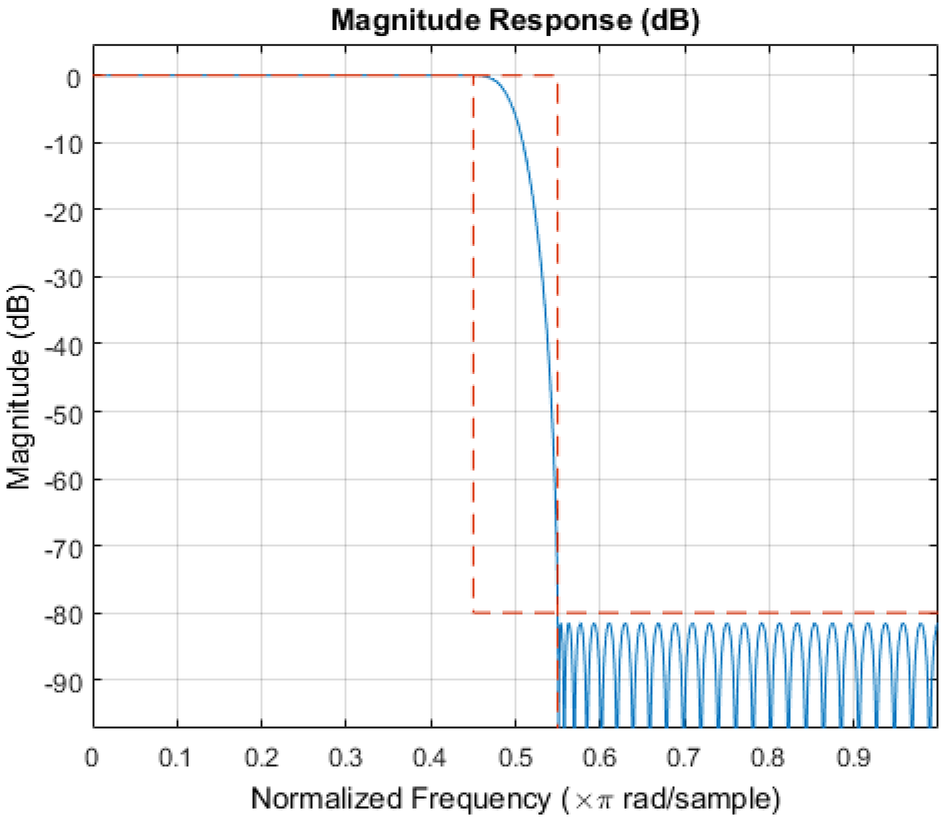
```
d = fdesign.halfband('tw,ast',0.1,80);
designmethods(d,'Systemobject',true)



Design Methods that support System objects for class fdesign.halfband (TW,Ast):


butter
ellip
iirlinphase
equiripple
kaiserwin


help(d,'equiripple')

 DESIGN Design a Equiripple FIR filter.
    HD = DESIGN(D, 'equiripple') designs a Equiripple filter specified by the
    FDESIGN object D, and returns the DFILT/MFILT object HD.

    HD = DESIGN(D, ..., 'SystemObject', true) implements the filter, HD,
    using a System object instead of a DFILT/MFILT object.

    HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a filter with the
    structure STRUCTURE.  STRUCTURE is 'dffir' by default and can be any of
    the following:

    'dffir'
    'dffirt'
    'dfsymfir'
    'fftfir'

    Some of the listed structures may not be supported by System object
    filters. Type validstructures(D, 'equiripple', 'SystemObject', true) to
    get a list of structures supported by System objects.
```

```
    HD = DESIGN(..., 'MinPhase', MPHASE) designs a minimum-phase filter
    when MPHASE is TRUE.  MPHASE is FALSE by default.

    HD = DESIGN(..., 'StopbandShape', SHAPE) designs a filter whose stopband
    has the shape defined by SHAPE.  SHAPE can be 'flat', '1/f', or 'linear'.
    SHAPE is 'flat' by default.

    HD = DESIGN(..., 'StopbandDecay', DECAY) specifies the decay to use when
    'StopbandShape' is not set to 'flat'.  When the shape is '1/f' this
    specifies the power that 1/f is raised.  When shaped is 'linear' this
    specifies the slope of the stopband in dB/rad/s.

    % Example #1 - Design a halfband lowpass equiripple filter with increased stopband
       TW = 0.1; % Transition Width
       Ast = 80; % Stopband Attenuation (dB)
       h  = fdesign.halfband('Type','Lowpass','TW,Ast',TW,Ast);
       Hd = design(h, 'equiripple', 'StopbandShape','linear','StopbandDecay',50);
       fvtool(Hd)
```

```
designopts(d,'equiripple')
```

```
ans =

  struct with fields:

    FilterStructure: 'dffir'
           MinPhase: 0
      StopbandShape: 'flat'
      StopbandDecay: 0
       SystemObject: 0
```

```
equiFilt = design(d,'equiripple','stopbandshape','flat','SystemObject',true);
fvtool(equiFilt);
```



The `fvtool` shows the equiripple nature of the filter.

Design an Equiripple FIR Filter with a Direct-Form Transposed Structure

This example designs an equiripple filter with a direct-form transposed structure by specifying the 'FilterStructure' argument. To set the design options for the filter, use the `designopts` method and options object `opts`.

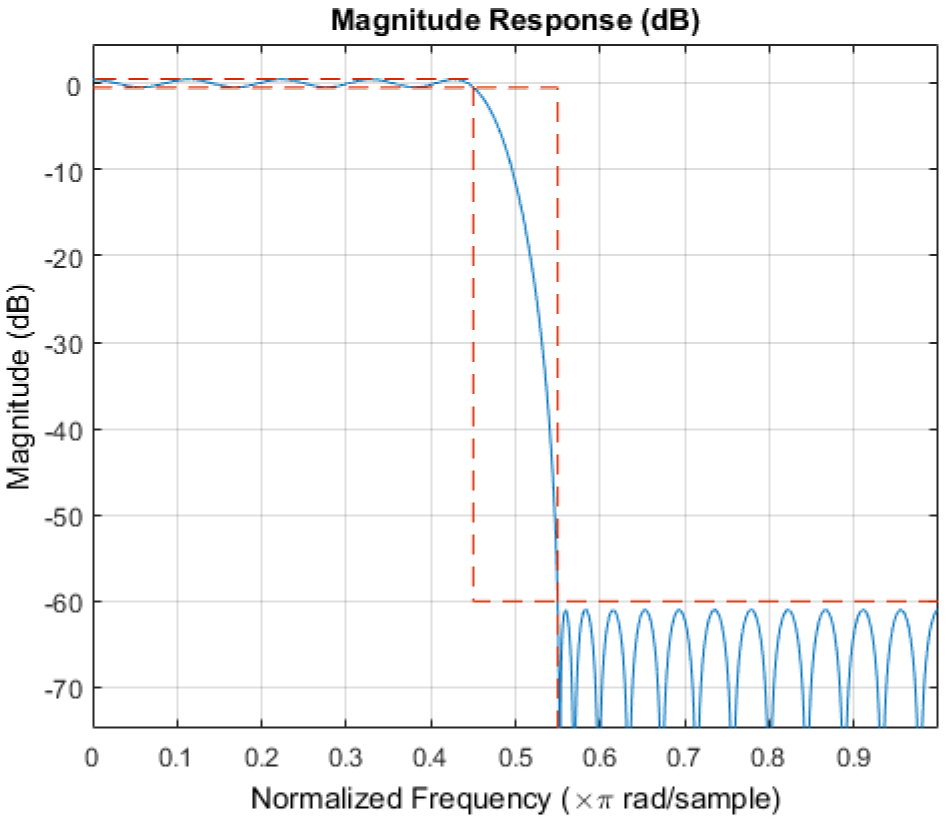
```
d = fdesign.lowpass('fp,fst,ap,ast');
opts = designopts(d,'equiripple');
opts.FilterStructure='dffirt';
opts.DensityFactor=20
```

```
opts = 

  struct with fields:

    FilterStructure: 'dffirt'
      DensityFactor: 20
           MinPhase: 0
           MaxPhase: 0
           MinOrder: 'any'
      StopbandShape: 'flat'
      StopbandDecay: 0
        UniformGrid: 1
       SystemObject: 0
```

```
firFilt = design(d,'equiripple','SystemObject',true,opts)
```

```
firFilt = 

  dsp.FIRFilter with properties:

            Structure: 'Direct form transposed'
      NumeratorSource: 'Property'
            Numerator: [1×43 double]
    InitialConditions: 0

  Use get to show all properties
```

```
fvtool(firFilt);
```



The MaxPhase design option for equripple FIR filters is currently only available for lowpass, highpass, bandpass, and bandstop filters.

See Also

See Also

Functions
fdesign.nyquist | firls | kaiserwin

Introduced in R2011a

fcfwrite

Write file containing filter coefficients

Syntax

```
fcfwrite(h)
fcfwrite(h,filename)
fcfwrite(...,'fmt')
```

Description

`fcfwrite(h)` writes a filter coefficient ASCII file in a folder you choose, or your current MATLAB working folder. `h` can be a single filter object or a vector of filter objects. On execution, `fcfwrite` opens the **Export Filter Coefficients to .FCF File** dialog box to let you assign a file name for the output file. You can choose the destination folder within this dialog as well.

The default file name is `untitled.fcf`. When you have DSP System Toolbox software, you can use `fcfwrite(h)` to write filter coefficient files for multirate filters, adaptive filters, and discrete-time filters.

`fcfwrite(h,filename)` writes the filter coefficients and general information to a text file called `filename` in your present MATLAB working folder and opens the file in the MATLAB editor for you to review or modify.

If you do not include a file extension in `filename`, `fcfwrite` adds the extension `fcf` to `filename`.

`fcfwrite(...,'fmt')` writes the filter coefficients in the format specified by the input argument `fmt`. Valid `fmt` values are `hex` for hexadecimal, `dec` for decimal, or `bin` for binary representation of the filter coefficients.

Examples

To demonstrate `fcfwrite`, create a fixed-point IIR filter at the command line, and then write the filter coefficients to a file named `iirfilter.fcf`.

```
d=fdesign.lowpass;
hd=design(d,'butter');
set(hd,'arithmetic','fixed');
fcfwrite(hd,'iirfilter.fcf');
```

Here is the output from `fcfwrite` as it appears in the MATLAB editor. Not shown here is the filename — `iirfilter.fcf` as specified and some comments at the top of the file.

```
%
%
% Coefficient Format: Decimal
%
% Discrete-Time IIR Filter (real)
% -------------------------------
% Filter Structure    : Direct-Form II, Second-Order
%                          Sections
% Number of Sections  : 13
% Stable              : Yes
% Linear Phase        : No
% Arithmetic          : fixed
% Numerator           : s16,13 -> [-4 4)
% Denominator         : s16,14 -> [-2 2)
% Scale Values        : s16,14 -> [-2 2)
% Input               : s16,15 -> [-1 1)
% Section Input       : s16,8 -> [-128 128)
% Section Output      : s16,10 -> [-32 32)
% Output              : s16,10 -> [-32 32)
% State               : s16,15 -> [-1 1)
% Numerator Prod      : s32,28 -> [-8 8)
% Denominator Prod    : s32,29 -> [-4 4)
% Numerator Accum     : s40,28 -> [-2048 2048)
% Denominator Accum   : s40,29 -> [-1024 1024)
% Round Mode          : convergent
% Overflow Mode       : wrap
% Cast Before Sum     : true


SOS matrix:
1  2  1  1  -0.22222900390625  0.88262939453125
1  2  1  1  -0.19903564453125  0.68621826171875
1  2  1  1  -0.18060302734375  0.5303955078125
1  2  1  1  -0.1658935546875   0.40570068359375
1  2  1  1  -0.154052734375    0.305419921875
1  2  1  1  -0.14453125        0.22479248046875
1  2  1  1  -0.136962890625    0.16015625
```

```
1   2   1   1   -0.13092041015625   0.10906982421875
1   2   1   1   -0.126220703125     0.06939697265625
1   2   1   1   -0.12274169921875   0.0399169921875
1   2   1   1   -0.12030029296875   0.01947021484375
1   2   1   1   -0.118896484375     0.0074462890625
1   1   0   1   -0.0592041015625    0

Scale Values:
0.41510009765625
0.371826171875
0.33746337890625
0.3099365234375
0.287841796875
0.27008056640625
0.25579833984375
0.2445068359375
0.23577880859375
0.22930908203125
0.22479248046875
0.22216796875
0.47039794921875
1
```

To write two or more filters out to one file, provide the filters as a vector to `fcfwrite`:

```
fcfwrite([hd hd1 hd2])
```

See Also

`dfilt`

Introduced in R2011a

filterDesigner

Open Filter Designer app

Syntax

`filterDesigner`

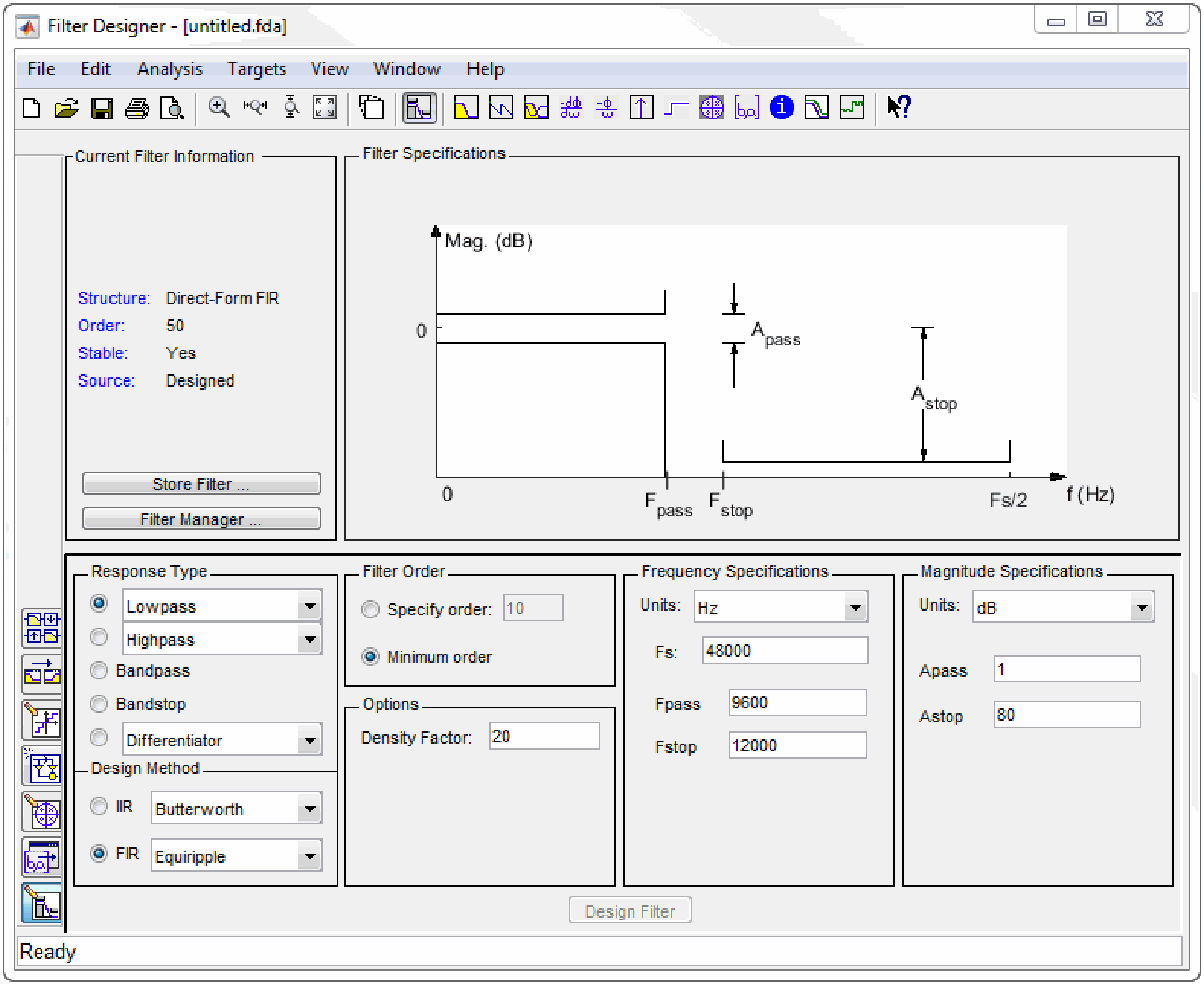
Description

`filterDesigner` opens the Filter Designer app. Use this tool to:

- Design filters
- Quantize filters (with DSP System Toolbox software installed)
- Analyze filters
- Modify existing filter designs
- Create multirate filters (with DSP System Toolbox software installed)
- Realize Simulink models of quantized, direct-form, FIR filters (with DSP System Toolbox software installed)
- Perform digital frequency transformations of filters (with DSP System Toolbox software installed)

Refer to "Use Filter Designer with DSP System Toolbox Software" for more information about using the analysis, design, and quantization features of filter designer. For general information about using filter designer, refer to "Using Filter Designer".

When you open the filter designer app and you have DSP System Toolbox software installed, filter designer incorporates features that are added by DSP System Toolbox software. With DSP System Toolbox software installed, filter designer lets you design and analyze quantized filters, as well as convert quantized filters to various filter structures, transform filters, design multirate filters, and realize models of filters.



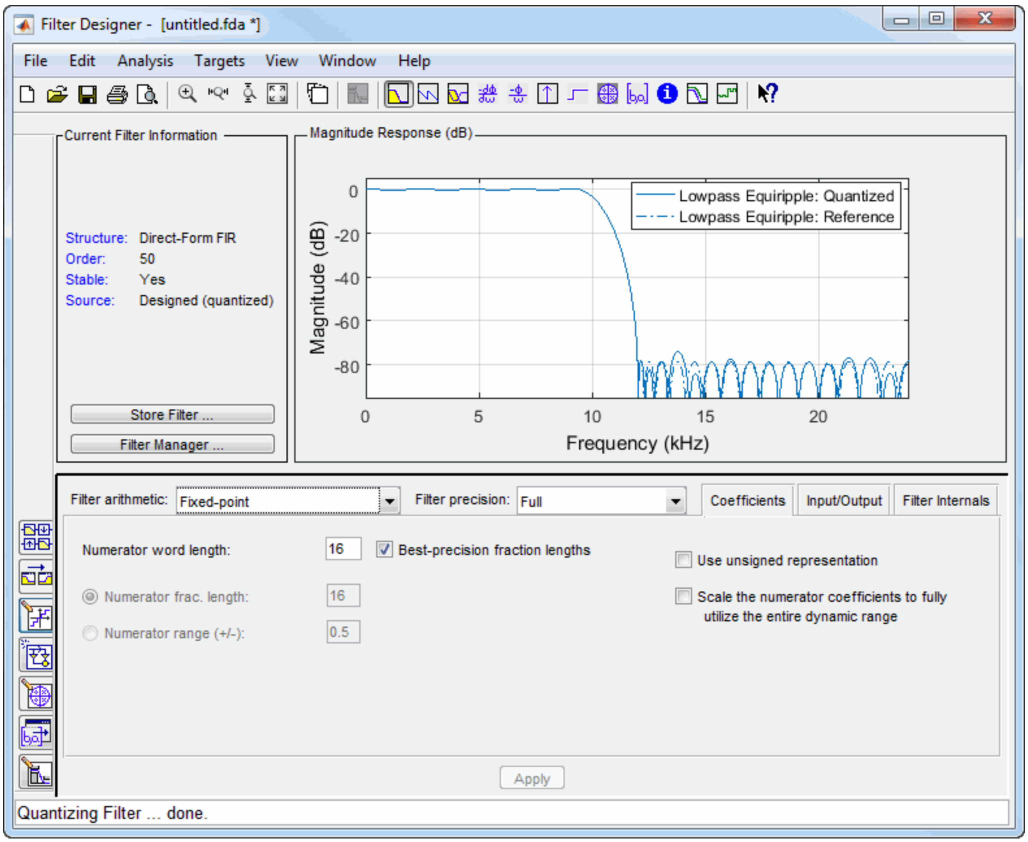
Use the buttons on the sidebar to configure the design area to use various tools in the filter designer app.

Set Quantization Parameters — provides access to the properties of the quantizers that compose a quantized filter. When you click **Set Quantization Parameters**, you see filter designer displaying the quantization options at the bottom of the dialog box (the design area), as shown in the figure.

Transform Filter — clicking this button opens the *Frequency Transformations* pane so you can use digital frequency transformations to change the magnitude response of your filter.

Create a multirate filter — clicking this button switches filter designer to multirate filter design mode so you can design interpolators, decimators, and fractional rate change filters.

Realize Model — starting from your quantized, direct-form, FIR filter, clicking this button creates a Simulink model of your filter structure in new model window.



Other options in the menu bar let you convert the filter structure to a new structure, change the order of second-order sections in a filter, or change the scaling applied to the filter, among many possibilities.

Examples

- "Use Filter Designer with DSP System Toolbox Software"

Tips

By incorporating many advanced filter design methods from DSP System Toolbox software, filter designer provides more design methods than the SPTool Filter Designer.

See Also

See Also

filterDesigner | `fvtool` | `sptool`

Introduced in R2011a

fdesign

Filter specification object

Syntax

```
filtSpecs = fdesign.response
filtSpecs = fdesign.response(spec)
filtSpecs = fdesign.response(...,Fs)
filtSpecs = fdesign.response(...,magunits)
```

Description

Filter Specification Objects

`filtSpecs = fdesign.`*`response`* returns a filter specification object, `filtSpecs`, of filter response *`response`*. To create filters from `filtSpecs`, use one of the design methods listed in "Using Filter Design Methods with Specification Objects" on page 5-509.

Note: Several of the filter response types described below are only available if your installation includes the DSP System Toolbox. The DSP System Toolbox significantly expands the functionality available for the specification, design, and analysis of filters.

Here is how you design filters using `fdesign`.

1. Use `fdesign.`*`response`* to construct a filter specification object.
2. Use `designmethods` to determine which filter design methods work for your new filter specification object.
3. Use `design` to apply your filter design method from step 2 to your filter specification object to construct a filter object.
4. Use FVTool to inspect and analyze your filter object.

Note `fdesign` does not create filters. `fdesign` returns a filter specification object that contains the specifications for a filter, such as the passband cutoff or attenuation in the stopband. To design a filter from a filter specification object `filtSpecs`, use `filtSpecs` with a filter design method such as `butter` — `IIRbutter = design(filtSpecs,'butter','SystemObject',true)`.

`response` can be one of the entries in the following table that specify the filter response desired, such as a bandstop filter or an interpolator.

fdesign Response Method	Description
`arbgrpdelay`	`fdesign.arbgrpdelay` creates an object to specify allpass arbitrary group delay filters. Requires the DSP System Toolbox
`arbmag`	`fdesign.arbmag` creates an object to specify IIR filters that have arbitrary magnitude responses defined by the input arguments.
`arbmagnphase`	`fdesign.arbmagnphase` creates an object to specify IIR filters that have arbitrary magnitude and phase responses defined by the input arguments. Requires the DSP System Toolbox.
`audioweighting`	`fdesign.audioweighting` creates a filter specification object for audio weighting filters. The supported audio weighting types are: A, C, C-message, ITU-T 0.41, and ITU-R 468-4 weighting. Requires the DSP System Toolbox
`bandpass`	`fdesign.bandpass` creates an object to specify bandpass filters.
`bandstop`	`fdesign.bandstop` creates an object to specify bandstop filters.
`ciccomp`	`fdesign.ciccomp` creates an object to specify filters that compensate for the CIC decimator or interpolator response curves. Requires the DSP System Toolbox.
`comb`	`fdesign.comb` creates an object to specify a notching or peaking comb filter. Requires the DSP System Toolbox.
`decimator`	`fdesign.decimator` creates an object to specify decimators. Requires the DSP System Toolbox

fdesign Response Method	Description
`differentiator`	`fdesign.differentiator` creates an object to specify an FIR differentiator filter.
`fracdelay`	`fdesign.fracdelay` creates an object to specify fractional delay filters. Requires the DSP System Toolbox.
`halfband`	`fdesign.halfband` creates an object to specify halfband filters. Requires the DSP System Toolbox.
`highpass`	`fdesign.highpass` creates an object to specify highpass filters.
`hilbert`	`fdesign.hilbert` creates an object to specify an FIR Hilbert transformer.
`interpolator`	`fdesign.interpolator` creates an object to specify interpolators. Requires the DSP System Toolbox.
`isinchp`	`fdesign.isinchp` creates an object to specify an inverse sinc highpass filter. Requires the DSP System Toolbox.
`isinclp`	`fdesign.isinclp` creates an object to specify an inverse sinc lowpass filters. Requires the DSP System Toolbox.
`lowpass`	`fdesign.lowpass` creates an object to specify lowpass filters.
`notch`	`fdesign.notch` creates an object to specify notch filters. Requires the DSP System Toolbox.
`nyquist`	`fdesign.nyquist` creates an object to specify Nyquist filters. Requires the DSP System Toolbox.
`octave`	`fdesign.octave` creates an object to specify octave and fractional octave filters. Requires the DSP System Toolbox.
`parameq`	`fdesign.parameq` creates an object to specify parametric equalizer filters. Requires the DSP System Toolbox.
`peak`	`fdesign.peak` creates an object to specify peak filters. Requires the DSP System Toolbox.
`polysrc`	`fdesign.polysrc` creates an object to specify polynomial sample-rate converter filters. Requires the DSP System Toolbox.

fdesign Response Method	Description
`rsrc`	`fdesign.rsrc` creates an object to specify rational-factor sample-rate convertors. Requires the DSP System Toolbox.

Use the `doc fdesign.`*`response`* syntax at the MATLAB prompt to get help on a specific structure. Using `doc` in a syntax like

```
doc fdesign.lowpass
doc fdesign.bandstop
```

gets more information about the lowpass or bandstop structure objects.

Each `response` has a property `Specification` that defines the specifications to use to design your filter. You can use defaults or specify the `Specification` property when you construct the specifications object.

Using the `Specification` property, you can provide filter constraints such as the filter order or the passband attenuation to use when you construct your filter from the specification object.

Properties

`fdesign` returns a filter specification object. Every filter specification object has the following properties.

Property Name	Default Value	Description
`Response`	Depends on the chosen type	Defines the type of filter to design, such as an interpolator or bandpass filter. This is a read-only value.
`Specification`	Depends on the chosen type	Defines the filter characteristics used to define the desired filter performance, such as the cutoff frequency `Fc` or the filter order `N`.
`Description`	Depends on the filter type you choose	Contains descriptions of the filter specifications used to define the object, and the filter specifications

Property Name	Default Value	Description
		you use when you create a filter from the object. This is a read-only value.
`NormalizedFrequency`	Logical `true`	Determines whether the filter calculation uses normalized frequency from 0 to 1, or the frequency band from 0 to $F_s/2$, the sampling frequency. Accepts either `true` or `false` without single quotation marks. Audio weighting filters do not support normalized frequency.

In addition to these properties, filter specification objects may have other properties as well, depending on whether they design single-rate filters or multirate filters.

Added Properties for Multirate Filters	Description
`DecimationFactor`	Specifies the amount to decrease the sampling rate. Always a positive integer.
`InterpolationFactor`	Specifies the amount to increase the sampling rate. Always a positive integer.
`PolyphaseLength`	Polyphase length is the length of each polyphase subfilter that composes the decimator or interpolator or rate-change factor filters. Total filter length is the product of `pl` and the rate change factors. `pl` must be an even integer.

`filtSpecs = fdesign.`*`response`*`(spec)`. In `spec`, you specify the variables to use that define your filter design, such as the passband frequency or the stopband attenuation. The specifications are applied to the filter design method you choose to design your filter.

For example, when you create a default lowpass filter specification object, `fdesign.lowpass` sets the passband frequency `Fp`, the stopband frequency `Fst`, the stopband attenuation `Ast`, and the passband ripple `Ap`:

```
filtSpecs = fdesign.lowpass
```

Use without a terminating semicolon to display the filter specifications.

The default specification `'Fp,Fst,Ap,Ast'` is only one of the possible specifications for `fdesign.lowpass`. To see all available specifications:

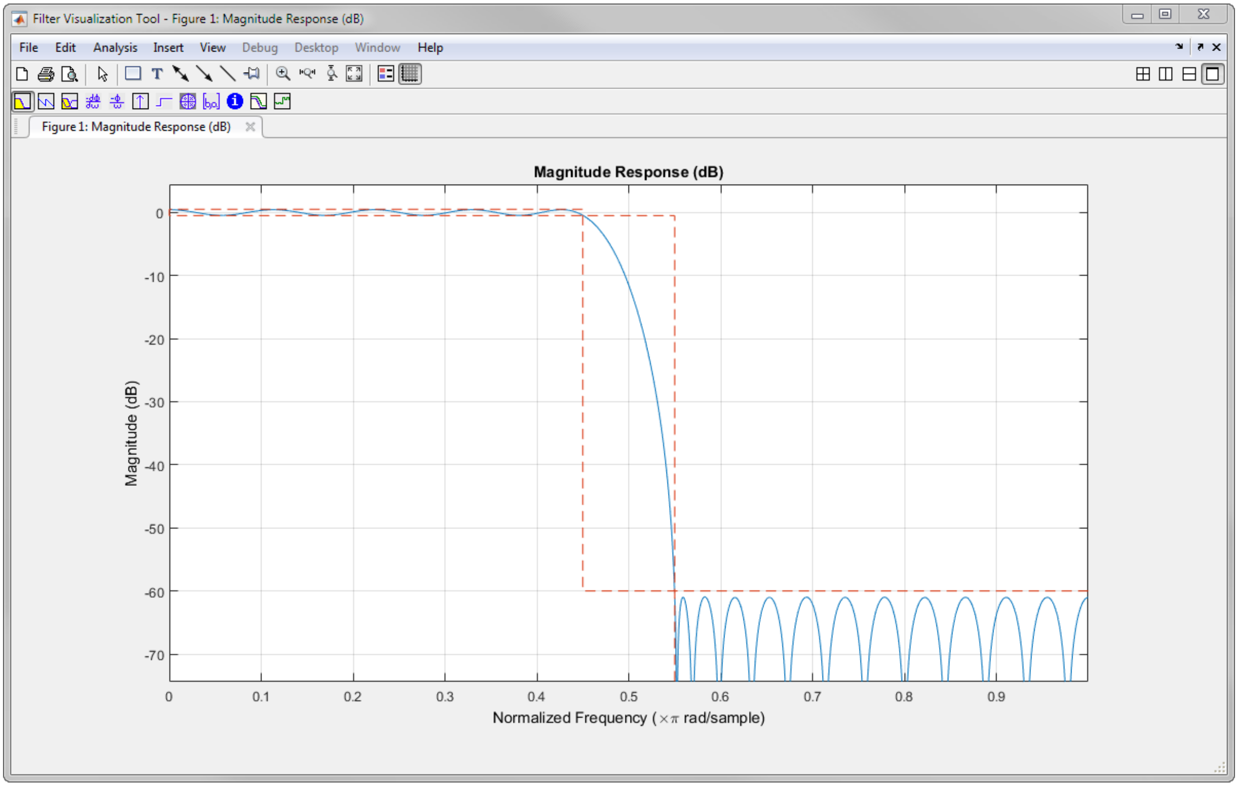
```
filtSpecs = fdesign.lowpass;
set(filtSpecs,'Specification')
```

The DSP System Toolbox software supports all available specification options. The Signal Processing Toolbox supports a subset of the specification options. See the reference pages for the filter specification object to determine which specification option your installation supports.

One important note is that the specification option you choose determines which design methods apply to the filter specifications object.

Specifications that do not contain the filter order result in minimum order designs when you invoke the `design` method:

```
filtSpecs = fdesign.lowpass;      % Specification is 'Fp,Fst,Ap,Ast'
FIReq = design(filtSpecs,'equiripple','SystemObject',true);
length(FIReq.Numerator)           % Returns 43. The filter order is 42
fvtool(FIReq)                     % View magnitude
```



`filtSpecs = fdesign.`*`response`*`(...,Fs)` specifies the sampling frequency in Hz to use in the filter specifications. The sampling frequency is a scalar trailing all other input arguments. If you specify a sampling frequency, all frequency specifications are in Hz.

`filtSpecs = fdesign.`*`response`*`(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of the following options:

- `'linear'` — specify the magnitude in linear units
- `'dB'` — specify the magnitude in decibels
- `'squared'` — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Using Filter Design Methods with Specification Objects

After you create a filter specification object, you use a filter design method to implement your filter with a selected algorithm. Use `designmethods` to determine valid design methods for your filter specification object.

```
filtSpecs = fdesign.lowpass('N,Fc,Ap,Ast',10,0.2,0.5,40);
designmethods(filtSpecs)
% Design FIR equiripple filter
FIReq = design(filtSpecs,'equiripple','SystemObject',true);
```

When you use any of the design methods without providing an output argument, the resulting filter design appears in FVTool by default.

Along with filter design methods, `fdesign` works with supporting methods that help you create filter specification objects or determine which design methods work for a given specifications object.

Supporting Function	Description
`setspecs`	Set all of the specifications simultaneously.
`designmethods`	Return the design methods.
`designopts`	Return the input arguments and default values that apply to a specifications object and method

You can set filter specification values by passing them after the `Specification` argument, or by passing the values without the `Specification`.

Filter object constructors take the input arguments in the same order as `setspecs` and `Specification`. Enter `doc setspecs` at the prompt for more information about using `setspecs`.

When the first input to `fdesign` is not a valid `Specification` option like `'N,Fc'`, `fdesign` assumes that the input argument is a filter specification and applies it using the default `Specification` option — `'Fp,Fst,Ap,Ast'` for a lowpass object, for example.

Examples

Design of Lowpass Filter

This example requires the DSP System Toolbox product.

Design a lowpass filter to be used on a signal sampled at 96 kHz. The passband of the filter extends up to 20 kHz. The stopband of the filter starts at 24 kHz. Specify a passband ripple of 0.01 dB and a stopband attenuation of 80 dB. Determine automatically the order required to meet the specifications.

Set up the filter design specifications and determine the available design algorithms.

```
Fs = 96e3;
Fpass = 20e3;
Fstop = 24e3;
Apass = 0.01;
Astop = 80;

filtSpecs = fdesign.lowpass(Fpass,Fstop,Apass,Astop,Fs);
designmethods(filtSpecs)
```

```
Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

Design an equiripple FIR filter and an elliptic IIR filter that meet the specifications. Measure the designs to verify that the filters satisfy the constraints.

```
lpFIR = design(filtSpecs,'equiripple','SystemObject',true);
lpIIR = design(filtSpecs,'ellip','SystemObject',true);

FIRmeas = measure(lpFIR)
IIRmeas = measure(lpIIR)
```

```
FIRmeas =

Sample Rate      : 96 kHz
Passband Edge    : 20 kHz
```

```
3-dB Point       : 21.4297 kHz
6-dB Point       : 21.8447 kHz
Stopband Edge    : 24 kHz
Passband Ripple  : 0.0092309 dB
Stopband Atten.  : 80.6014 dB
Transition Width : 4 kHz


IIRmeas =

Sample Rate      : 96 kHz
Passband Edge    : 20 kHz
3-dB Point       : 20.5524 kHz
6-dB Point       : 20.7138 kHz
Stopband Edge    : 24 kHz
Passband Ripple  : 0.01 dB
Stopband Atten.  : 80 dB
Transition Width : 4 kHz
```

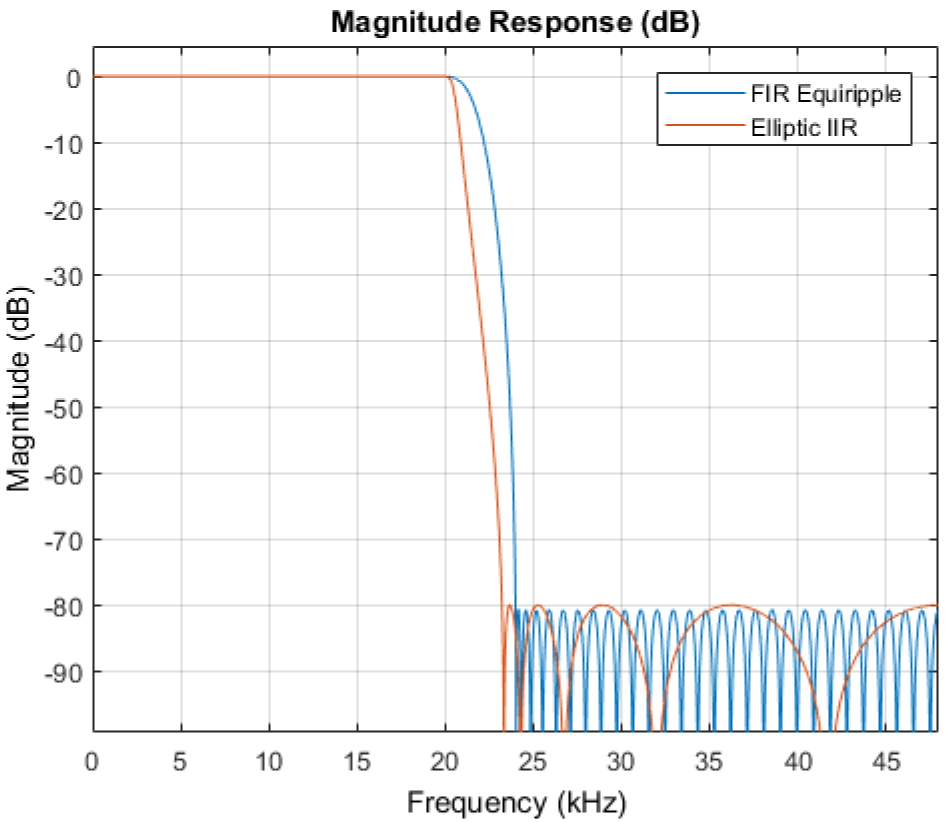
Estimate and display the computational cost of each filter. The equiripple FIR filter requires many more coefficients than the elliptic IIR filter.

```
FIRcost = cost(lpFIR)
IIRcost = cost(lpIIR)
```

```
FIRcost =

  struct with fields:

                  NumCoefficients: 101
                        NumStates: 100
    MultiplicationsPerInputSample: 101
          AdditionsPerInputSample: 100


IIRcost =

  struct with fields:

                  NumCoefficients: 20
                        NumStates: 10
    MultiplicationsPerInputSample: 20
          AdditionsPerInputSample: 20
```

Visualize the resulting designs with FVTool to compare their properties.

```
fvtool(lpFIR,lpIIR,'Fs',Fs);
legend('FIR Equiripple','Elliptic IIR')
```



Design of Lowpass Decimator

This example requires the DSP System Toolbox product.

Design a 100-tap FIR lowpass decimator filter that reduces the sample rate of a signal from 60 kHz to 20 kHz. The passband of the filter extends up to 6 kHz. Specify a passband ripple of 0.01 dB and a stopband attenuation of 100 dB.

```
Fs = 60e3;
N = 99;
Fpass = 6e3;
Apass = 0.01;
Astop = 100;
M = Fs/20e3;

filtSpecs = fdesign.decimator(M,'lowpass','N,Fp,Ap,Ast',N,Fpass,Apass,Astop,Fs);
```

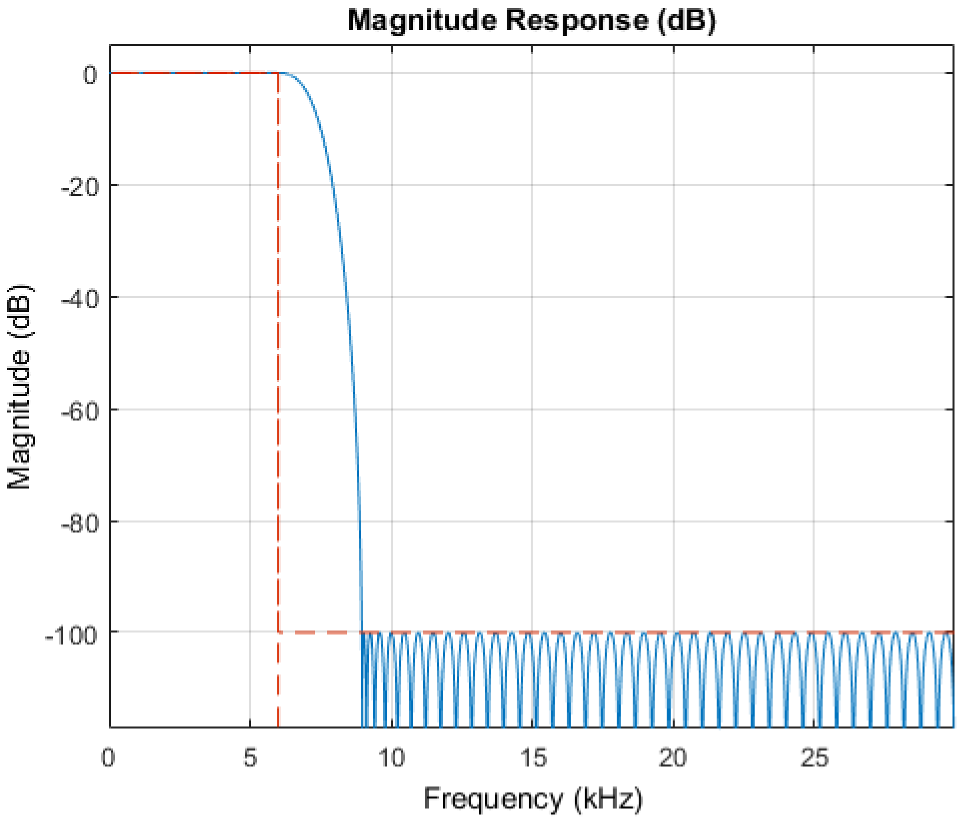
Design the filter and display information about it.

```
decimFIR = design(filtSpecs,'equiripple','SystemObject',true);
info(decimFIR)
```

```
ans =

  10×56 char array

    'Discrete-Time FIR Multirate Filter (real)               '
    '-----------------------------------------               '
    'Filter Structure   : Direct-Form FIR Polyphase Decimator'
    'Decimation Factor  : 3                                  '
    'Polyphase Length   : 34                                 '
    'Filter Length      : 100                                '
    'Stable             : Yes                                '
    'Linear Phase       : Yes (Type 2)                       '
    '                                                        '
    'Arithmetic         : double                             '
```

Visualize the magnitude response of the filter.

```
fvtool(decimFIR,'Fs',Fs)
```



Lowpass Butterworth Filter Specification and Design

Use a filter specification object to construct a lowpass Butterworth filter with the default specification, `'Fp,Fst,Ap,Ast'`. Specify a passband edge frequency of 0.4π rad/sample, a stopband frequency of 0.5π rad/sample, a passband ripple of 1 dB, and a stopband attenuation of 80 dB.

```
filtSpecs = fdesign.lowpass(0.4,0.5,1,80);
```

Determine which design methods apply to `filtSpecs`.

```
designmethods(filtSpecs)
```

```
Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

You can use `filtSpecs` and the `butter` design method to design a Butterworth filter. Determine the available design options.

```
designoptions(filtSpecs,'butter')
```

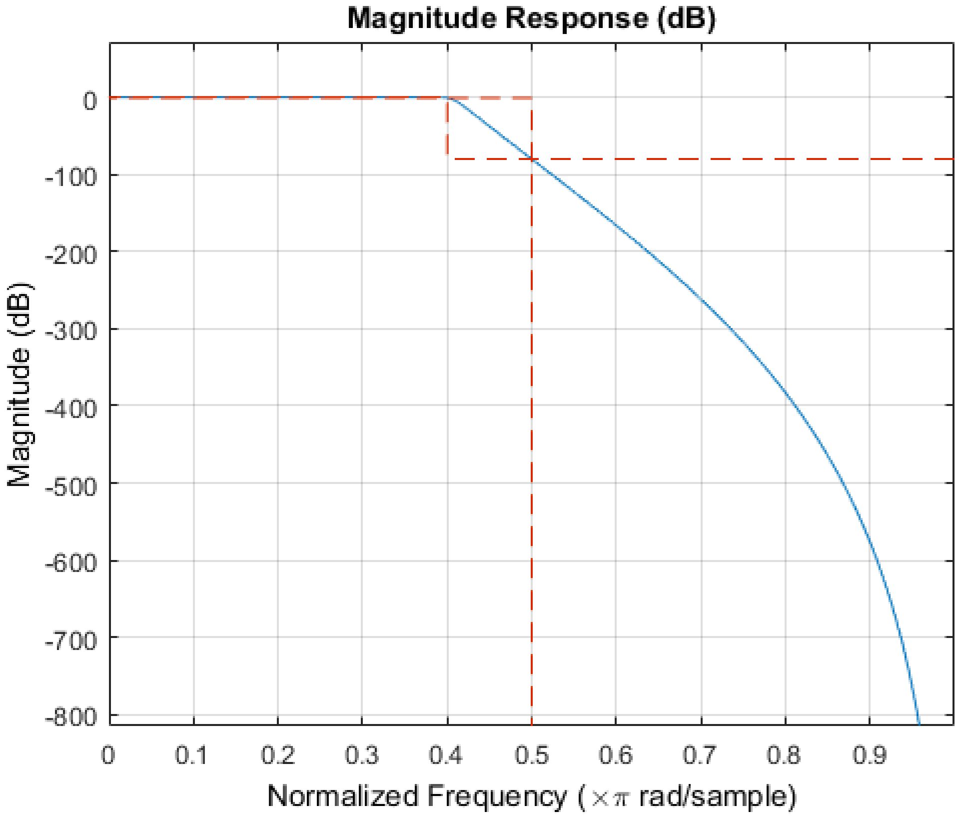
```
ans =

  struct with fields:

           FilterStructure: {1×6 cell}
              SOSScaleNorm: 'ustring'
              SOSScaleOpts: 'fdopts.sosscaling'
              MatchExactly: {'passband'  'stopband'}
              SystemObject: 'bool'
    DefaultFilterStructure: 'df2sos'
       DefaultMatchExactly: 'stopband'
       DefaultSOSScaleNorm: ''
       DefaultSOSScaleOpts: [1×1 fdopts.sosscaling]
       DefaultSystemObject: 0
```

The filter order necessary to meet a set of design constraints must also be rounded up to an integer value. This loosens some of the constraints, and, as a consequence, some design specifications are met while others are exceeded. The `'MatchExactly'` option allows you to match the passband or stopband exactly while exceeding the specification for the other band. Design the filter so that it matches the passband exactly.

```
IIRbutter = design(filtSpecs,'butter','MatchExactly','passband', ...
    'SystemObject',true);
```

Use FVTool to visualize the magnitude response of the filter.

```
fvtool(IIRbutter)
```



If you have the DSP System Toolbox software installed, the preceding figure appears with the filter specification mask.

See Also

See Also

Apps
Filter Designer

Functions
`designmethods` | `designopts` | `filterBuilder` | `fvtool`

Introduced in R2009a

fdesign.arbgrpdelay

Arbitrary group delay filter specification object

Syntax

```
D = fdesign.arbgrpdelay(SPEC)
D = fdesign.arbgrpdelay(SPEC,SPEC1,SPEC2,...)
D = fdesign.arbgrpdelay(N,F,Gd)
D = fdesign.arbgrpdelay(...,Fs)
```

Description

Arbitrary group delay filters are allpass filters you can use for correcting phase distortion introduced by other filters. `fdesign.arbgrpdelay` uses an iterative least *p*-th norm optimization procedure to minimize the phase response error [1].

`D = fdesign.arbgrpdelay(SPEC)` specifies an allpass arbitrary group delay filter with the `Specification` property set to `SPEC`. See "Input Arguments" on page 5-519 for a description of supported specifications.

`D = fdesign.arbgrpdelay(SPEC,SPEC1,SPEC2,...)` initializes the allpass arbitrary group delay filter specification object with specifications `SPEC1,SPEC2,...`. See SPEC for a description of supported specifications.

`D = fdesign.arbgrpdelay(N,F,Gd)` specifies an allpass arbitrary group delay filter. The filter order is equal to `N`, frequency vector equal to `F`, and group delay vector equal to `Gd`. See SPEC for a description of the filter order, frequency vector, and group delay vector inputs. See the example "Design an Allpass Filter With an Arbitrary Group Delay" on page 5-523 for an example using this syntax.

`D = fdesign.arbgrpdelay(...,Fs)` specifies the sampling frequency in hertz as a trailing scalar. If you do not specify a sampling frequency, all frequencies are normalized frequencies and group delay values are in samples. If you specify a sampling frequency, group delay values are in seconds.

Tips

If your arbitrary group delay design produces the error `Poorly conditioned Hessian matrix`, attempt one or more of the following:

- Set the `MaxPoleRadius` IIR lp norm design option to some number less than 1. Set this option when you `design` your filter with the syntax:

  ```
  design(d,'iirlpnorm','MaxPoleRadius',0.95)
  ```
  See the "Design an Arbitrary Group Delay Filter" on page 5-521 and "Multiband Delay Equalization" on page 5-524 examples for the use of the `MaxPoleRadius` design option.
- Reduce the order of your filter design.

Input Arguments

SPEC

Filter specification. `SPEC` is one of the following two options. The entries are not case sensitive.

- `'N,F,Gd'`
- `'N,B,F,Gd'`

The filter specifications are defined as follows:

- `N` — Filter order. This value must be an even positive integer. The numerator and denominator orders are both equal to `N`. "Allpass Systems" on page 5-527 explains why the numerator and denominator filter orders are equal and the order must be even in `fdesign.arbgrpdelay`.
- `F` — Frequency vector for the group delay specifications. The elements of the frequency vector must increase monotonically. If you do not specify a sampling frequency, `Fs`, in hertz, the frequencies are normalized frequencies. For a single-band design, the first element of the normalized frequency vector must be zero and the last element must be 1. These correspond to 0 and π radians/sample respectively. For multiband designs, the union of the frequency vectors must range from [0,1].

  If you specify a sampling frequency, `Fs`, the first element of the frequency vector in a single-band design must be 0. The last element must be the Nyquist frequency, Fs/2. For multiband designs, the union of the frequency vectors must range from [0,Fs/2].

- `Gd` — Group delay vector. A vector with nonnegative elements equal in length to the frequency vector, `F`. The elements of `Gd` specify the nonnegative group delay at the corresponding element of the frequency vector, `F`.

  If you do not specify a sampling frequency, `Fs`, in Hertz, the group delays are in samples. If you specify a sampling frequency, the group delays are in seconds.
- `B` — Number of frequency bands. If you use this specification, you must specify a frequency and group delay vector for each band. The union of the frequency vectors must range from [0,1] in normalized frequency, or [0,Fs/2] when a sampling frequency is specified. The elements in the union of the frequency bands must be monotonically increasing.

  For example:

  ```
  filtorder = 14;
  freqband1 = [0 0.1 0.4]; grpdelay1 = [1 2 3];
  freqband2 = [0.5 0.8 1]; grpdelay2 = [3 2 1];
  D = fdesign.arbgrpdelay('N,B,F,Gd',filtorder,2,freqband1,grpdelay1,freqband2,grpdela
  ```

Default: `'N,F,Gd'`

`Fs`

Sampling frequency. Specify the sampling frequency as a trailing positive scalar after all other input arguments. Specifying a sampling frequency forces the group delay units to be in seconds. If you specify a sampling frequency, the first element of the frequency vector must be 0. The last element must be the Nyquist frequency, Fs/2.

Output Arguments

`D`

Filter specification object. An allpass arbitrary group delay filter specification object containing the following modifiable properties: `Specification`, `NormalizedFrequency`, `FilterOrder`, `Frequencies`, and `GroupDelay`.

Use the `normalizefreq` method to change the `NormalizedFrequency` property after construction.

Examples

Design an Arbitrary Group Delay Filter

Construct a signal consisting of two discrete-time windowed sinusoids (wave packets) with disjoint time support to illustrate frequency dispersion. One discrete-time sinusoid has a frequency of pi/2 radians/sample and the other has a frequency of pi/4 radians/sample. There are 9 periods of the higher-frequency sinusoid that precede 5 periods of the lower-frequency signal.

Create the signal.

```
x = zeros(300,1);
x(1:36) = cos(pi/2*(0:35)).*hamming(36)';
x(40:40+39) = cos(pi/4*(0:39)).*hamming(40)';
```

Create an arbitrary group delay filter that delays the higher-frequency wave packet by approximately 100 samples.

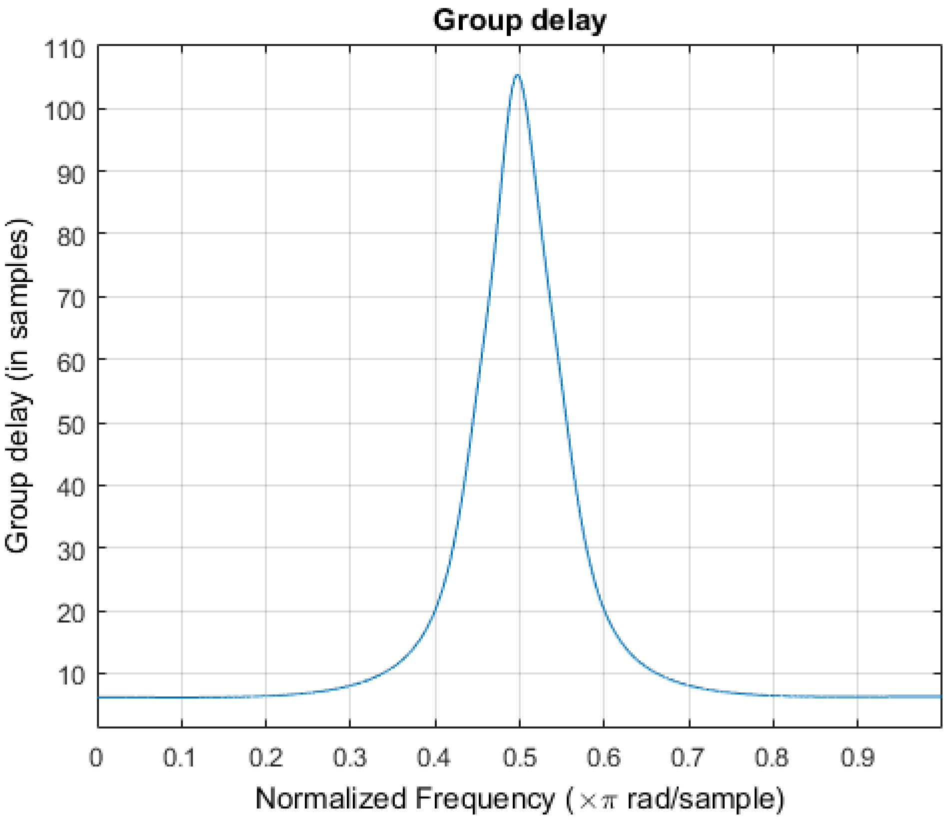
```
N = 18;
f = 0:.1:1;
gd = ones(size(f));
```

Delay pi/2 radians/sample by 100 samples

```
gd(6) = 100;
d = fdesign.arbgrpdelay(N,f,gd);
Hd = design(d,'iirlpnorm','MaxPoleRadius',0.9,'SystemObject',true);
```

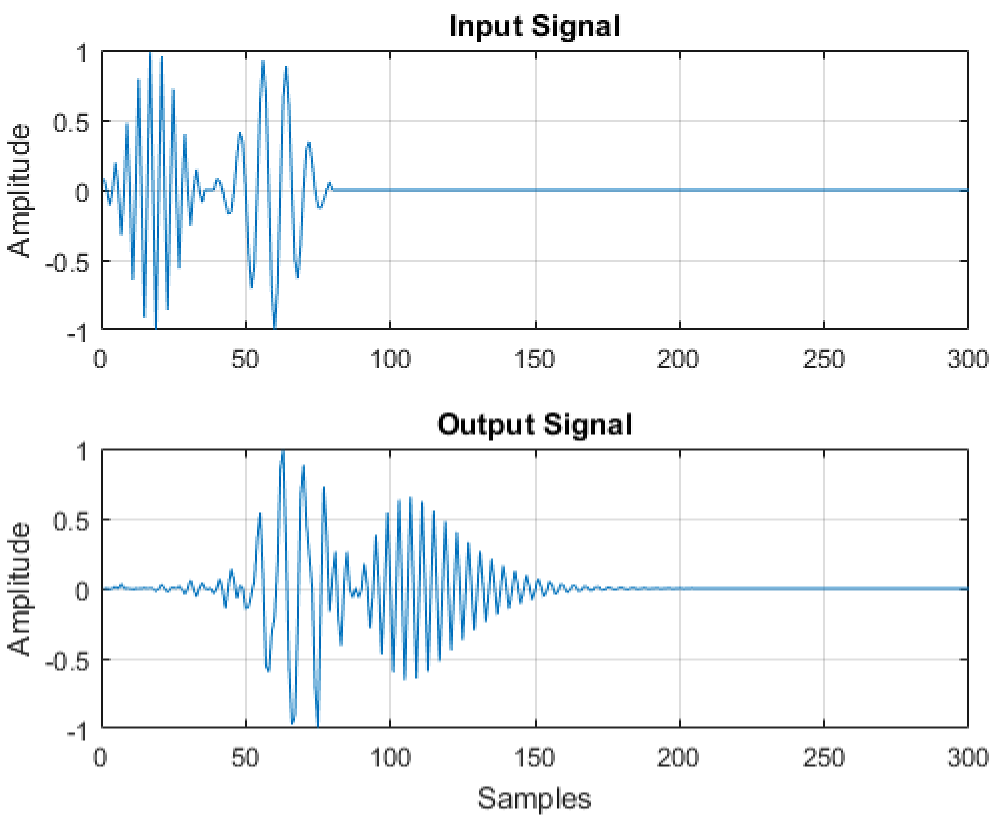
Visualize the group delay

```
fvtool(Hd,'analysis','grpdelay');
```



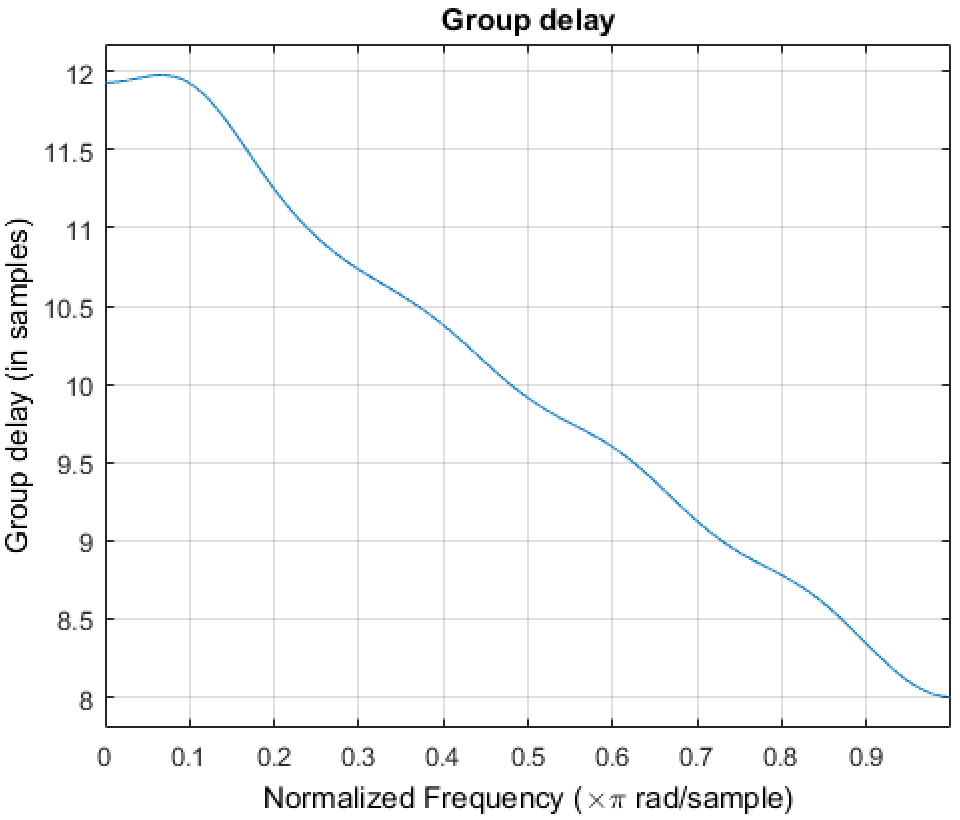
Filter the input signal with the arbitrary group delay filter and illustrate the frequency dispersion. The high-frequency wave packet, which initially preceded the low-frequency wave packet, now occurs later because of the nonconstant group delay.

```
y = Hd(x);
subplot(211)
plot(x); title('Input Signal');
grid on; ylabel('Amplitude');
subplot(212);
plot(y); title('Output Signal'); grid on;
xlabel('Samples'); ylabel('Amplitude');
```



Design an Allpass Filter With an Arbitrary Group Delay

```
N = 10;
f = [0 0.02 0.04 0.06 0.08 0.1 0.25 0.5 0.75 1];
g = [5 5 5 5 5 5 4 3 2 1];
w = [2 2 2 2 2 2 1 1 1 1];
hgd = fdesign.arbgrpdelay(N,f,g);
Hgd = design(hgd,'iirlpnorm','Weights',w,'MaxPoleRadius',0.95,...
    'SystemObject',true);
fvtool(Hgd,'Analysis','grpdelay') ;
```

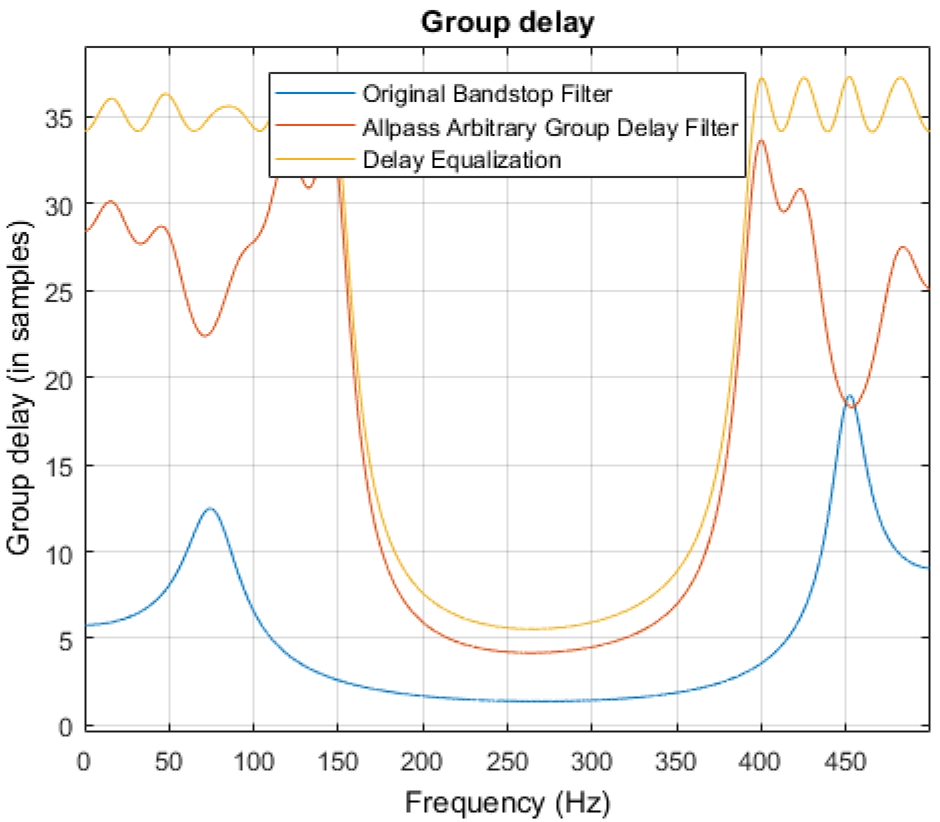


Multiband Delay Equalization

Perform multiband delay equalization outside the stopband.

```
Fs = 1e3;
Hcheby2 = design(fdesign.bandstop('N,Fst1,Fst2,Ast',10,150,400,60,Fs),'cheby2',...
    'SystemObject',true);
f1 = 0.0:0.5:150; % Hz
g1 = grpdelay(Hcheby2,f1,Fs).'/Fs; % seconds
f2 = 400:0.5:500; % Hz
g2 = grpdelay(Hcheby2,f2,Fs).'/Fs; % seconds
maxg = max([g1 g2]);
% Design an arbitrary group delay allpass filter to equalize the group
% delay of the bandstop filter. Use an order 18 multiband design and specify
```

```
% two bands.
hgd = fdesign.arbgrpdelay('N,B,F,Gd',18,2,f1,maxg-g1,f2,maxg-g2,Fs);
Hgd = design(hgd,'iirlpnorm','MaxPoleRadius',0.95,'SystemObject',true);
Hcascade = cascade(Hcheby2,Hgd);
hft = fvtool(Hcheby2,Hgd,Hcascade,'Analysis','grpdelay','Fs',Fs);
    legend(hft,'Original Bandstop Filter','Allpass Arbitrary Group Delay Filter',...
    'Delay Equalization', 'Location','North');
```



Definitions

Group Delay in Discrete-Time Filter Design

The frequency response of a rational discrete-time filter is:

$$H(e^{j\omega}) = \frac{B(e^{j\omega})}{A(e^{j\omega})}$$

The argument of the frequency response as a function of the angle, ω, is referred to as the *phase response*.

The negative of the first derivative of the argument with respect to ω is the group delay.

$$\tau(\omega) = -\frac{d}{d\omega} Arg(H(e^{j\omega}))$$

Systems with nonlinear phase responses have nonconstant group delay, which causes dispersion of the frequency components of the signal. You may not want this phase distortion even if the magnitude distortion introduced by the filter produces the desired effect. See “Design an Arbitrary Group Delay Filter” on page 5-521 for an illustration of frequency dispersion resulting from nonconstant group delay.

In these cases, you can cascade the frequency-selective filter with an allpass filter that compensates for the group delay. This process is often referred to as *delay equalization*.

Allpass Systems

The general form of an allpass system function with a real-valued impulse response is:

$$H_{ap}(z) = \prod_{k=1}^{M} \frac{z^{-1} - d_k}{1 - d_k z^{-1}} \prod_{k=1}^{N} \frac{(z^{-1} - c_k)(z^{-1} - c_k^*)}{(1 - c_k z^{-1})(1 - c_k^* z^{-1})}$$

where the d_k denote the real-valued poles and the c_k denote the complex-valued poles, which occur in conjugate pairs.

The preceding equation demonstrates that allpass systems with real-valued impulse responses have *2N+M* zeros and poles. The poles and zeros occur in pairs with reciprocal magnitudes. The filter order is always the same for the numerator and denominator.

Because `fdesign.arbgrpdelay` uses robust second-order section (biquad) filter structures to implement the allpass arbitrary group delay filter, the filter order must be even.

Algorithms

`fdesign.arbgrpdelay` uses a least *p*-th norm iterative optimization described in [1].

Alternatives

`iirgrpdelay` — Returns an allpass arbitrary group delay filter. The `iirgrpdelay` function returns the numerator and denominator coefficients. This behavior differs from that of `fdesign.arbgrpdelay`, which returns the filter in second-order sections. `iirgrpdelay` accepts only normalized frequencies.

References

[1] Antoniou, A. *Digital Signal Processing: Signals, Systems, and Filters.*, New York:McGraw-Hill, 2006, pp. 719–771.

See Also

`fdesign` | `design` | `iirgrpdelay`

Topics

"Design a Filter in Fdesign — Process Overview"

Introduced in R2011b

fdesign.arbmag

Arbitrary response magnitude filter specification object

Syntax

```
D= fdesign.arbmag
D= fdesign.arbmag(SPEC)
D = fdesign.arbmag(SPEC,specvalue1,specvalue2,...)
D = fdesign.arbmag(specvalue1,specvalue2,specvalue3)
D = fdesign.arbmag(...,Fs)
```

Description

`D= fdesign.arbmag` constructs an arbitrary magnitude filter specification object `D`.

`D= fdesign.arbmag(SPEC)` initializes the `Specification` property to `SPEC`. The input argument `SPEC` must be one of the entries shown in the following table. Specification entries are not case sensitive.

Note: Specification entries marked with an asterisk require the DSP System Toolbox software.

- `'N,F,A'` — Single band design (default)
- `'F,A,R'` — Single band minimum order design *
- `'N,B,F,A'` — Multiband design
- `'N,B,F,A,C'` — Constrained multiband design *
- `'B,F,A,R'` — Multiband minimum order design *
- `'Nb,Na,F,A'` — Single band design *
- `'Nb,Na,B,F,A'` — Multiband design *

The `SPEC` entries are defined as follows:

- A — Amplitude vector. Values in A define the filter amplitude at frequency points you specify in f, the frequency vector. If you use A, you must use F as well. Amplitude values must be real. For complex values designs, use fdesign.arbmagnphase.
- B — Number of bands in the multiband filter
- C — Constrained band flag. This enables you to constrain the passband ripple in your multiband design. You cannot constrain the passband ripple in all bands simultaneously.
- F — Frequency vector. Frequency values in specified in F indicate locations where you provide specific filter response amplitudes. When you provide F, you must also provide A.
- N — Filter order for FIR filters and the numerator and denominator orders for IIR filters.
- Nb — Numerator order for IIR filters
- Na — Denominator order for IIR filter designs
- R — Ripple

By default, this method assumes that all frequency specifications are supplied in normalized frequency.

Specifying Frequency and Amplitude Vectors

F and A are the input arguments you use to define the filter response desired. Each frequency value you specify in F must have a corresponding response value in A. The following table shows how F and A are related.

Define the frequency vector F as [0 0.25 0.3 0.4 0.5 0.6 0.7 0.75 1.0]

Define the response vector A as [1 1 0 0 0 0 0 1 1]

These specifications connect F and A as shown here:

F (Normalized Frequency)	A (Response Desired at F)
0	1
0.25	1
0.3	0
0.4	0
0.5	0

F (Normalized Frequency)	A (Response Desired at F)
0.6	0
0.7	0
0.75	1
1.0	1

Different specifications can have different design methods available. Use `designmethods` to get a list of design methods available for a given specification and filter specification object.

Use `designopts` to get a list of design options available for a filter specification object and a given design method. Enter `help(D,METHOD)` to get detailed help on the available design options for a given design method.

`D = fdesign.arbmag(SPEC,specvalue1,specvalue2,...)` initializes the specifications with `specvalue1`, `specvalue2`. Use `get(D,'Description')` for descriptions of the various specifications `specvalue1`, `specvalue2`, ... `specvalueN`.

`D = fdesign.arbmag(specvalue1,specvalue2,specvalue3)` uses the default specification `'N,F,A'`, setting the filter order, filter frequency vector, and the amplitude vector to the values `specvalue1`, `specvalue2`, and `specvalue3`.

`D = fdesign.arbmag(...,Fs)` specifies the sampling frequency in Hz. All other frequency specifications are also assumed to be in Hz when you specify `Fs`.

Examples

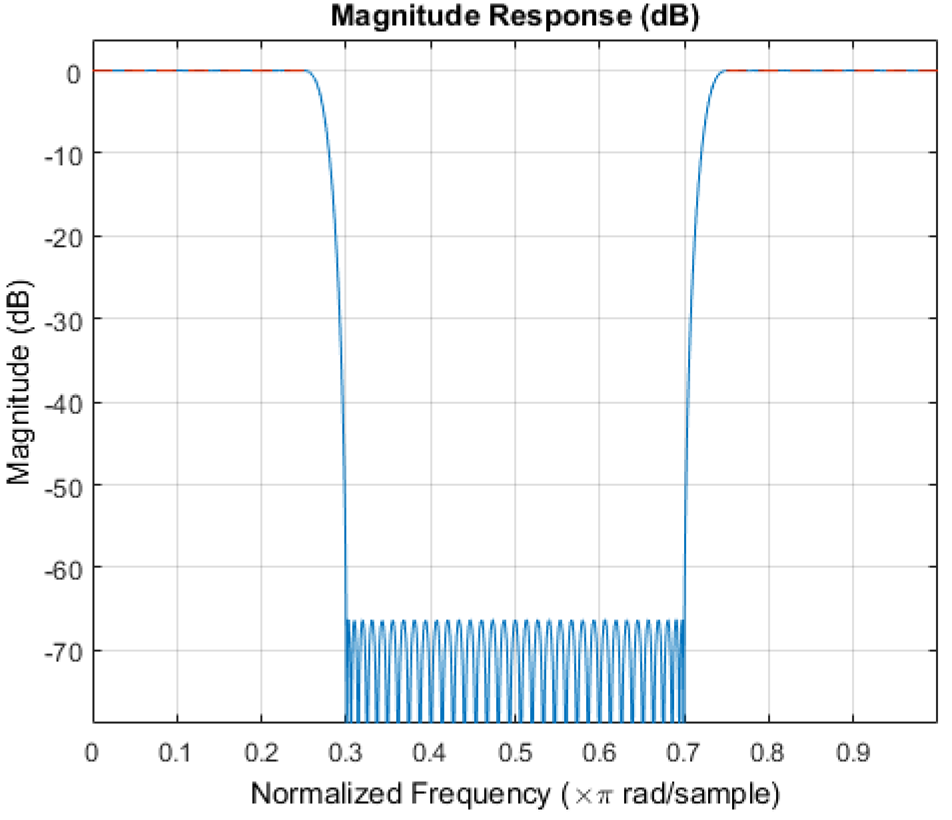
Design Multiband Arbitrary-Magnitude Filter

Use `fdesign.arbmag` to design a three-band filter.

- Define the frequency vector `F` = [0 0.25 0.3 0.4 0.5 0.6 0.7 0.75 1.0].
- Define the response vector `A` = [1 1 0 0 0 0 0 1 1].

```
N = 150;
B = 3;
F = [0 .25 .3 .4 .5 .6 .7 .75 1];
A = [1 1 0 0 0 0 0 1 1];
A1 = A(1:2);
A2 = A(3:7);
```

```
A3 = A(8:end);
F1 = F(1:2);
F2 = F(3:7);
F3 = F(8:end);
d = fdesign.arbmag('N,B,F,A',N,B,F1,A1,F2,A2,F3,A3);
Hd = design(d);
fvtool(Hd)
```



A response with two passbands -- one roughly between 0 and 0.25 and the second between 0.75 and 1 -- results from the mapping between F and A.

Design Single-Band Arbitrary-Magnitude Filter

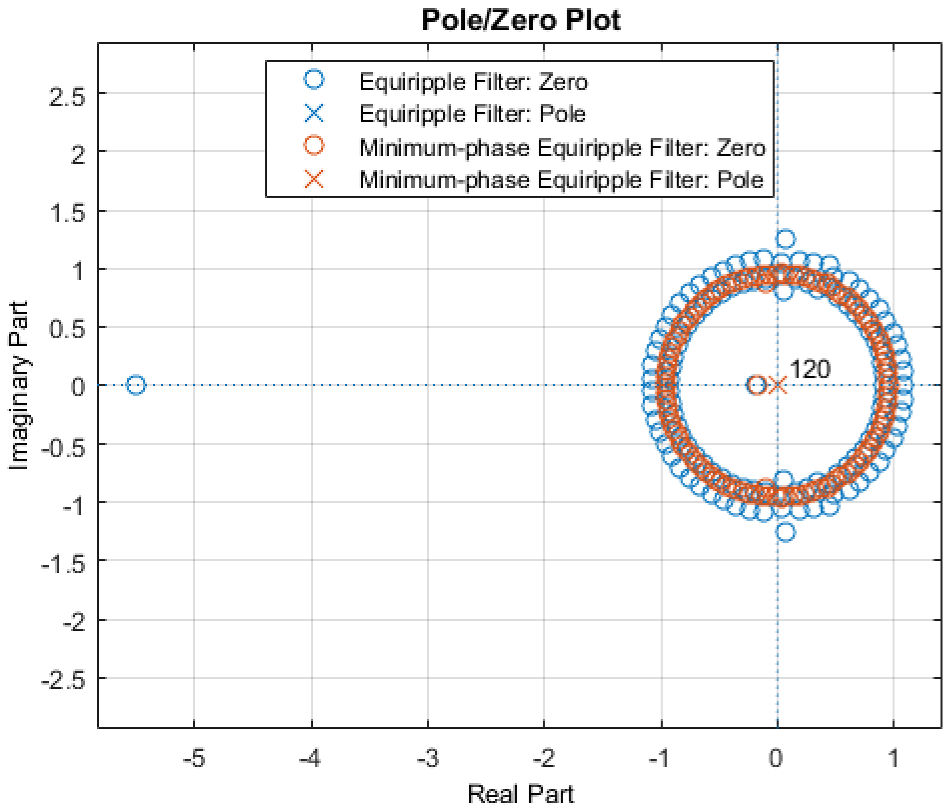
Use `fdesign.arbmag` to design a single band equiripple filter.

Specify 100 frequency points.

```
n = 120;
f = linspace(0,1,100);

as = ones(1,100)-f*0.2;
absorb = [ones(1,30),1-0.6*bohmanwin(10)',ones(1,5), ...
    1-0.5*bohmanwin(8)',ones(1,47)];
a = as.*absorb;

d = fdesign.arbmag('N,F,A',n,f,a);
hd1 = design(d,'equiripple');
```

Design a minimum-phase equiripple filter. Visualize the poles and zeros of the two filters.

```
hd2 = design(d,'equiripple','MinPhase',true);

hfvt = fvtool(hd1,hd2,'Analysis','polezero');
legend(hfvt,'Equiripple Filter','Minimum-phase Equiripple Filter')
```



Design Multiband Minimum-Order Arbitrary-Magnitude Filter

Use `fdesign.arbmag` to design a multiband minimum order filter.

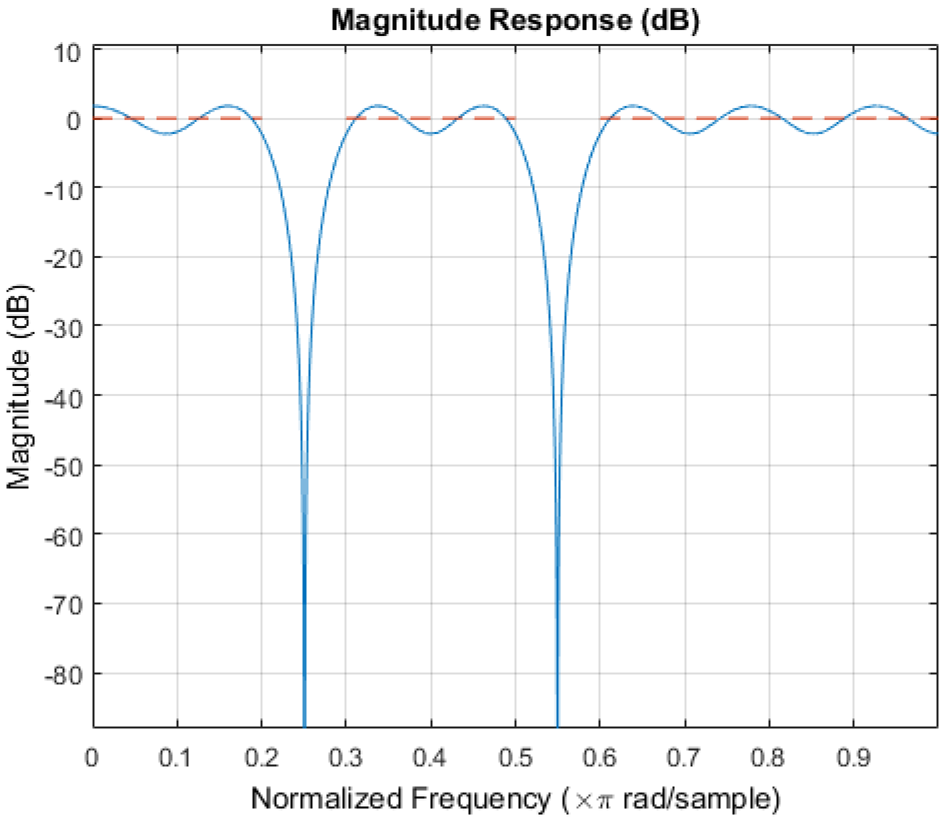
Place the notches at 0.25π and 0.55π rad/sample.

```
d = fdesign.arbmag('B,F,A,R');
d.NBands = 5;
d.B1Frequencies = [0 0.2];
d.B1Amplitudes = [1 1];
d.B1Ripple = 0.25;
d.B2Frequencies = 0.25;
d.B2Amplitudes = 0;
d.B3Frequencies = [0.3 0.5];
```

```
d.B3Amplitudes = [1 1];
d.B3Ripple = 0.25;
d.B4Frequencies = 0.55;
d.B4Amplitudes = 0;
d.B5Frequencies = [0.6 1];
d.B5Amplitudes = [1 1];
d.B5Ripple = 0.25;
Hd = design(d,'equiripple');
```

Visualize the frequency response of the resulting filter.

```
fvtool(Hd)
```



Design Multiband Constrained Arbitrary-Magnitude Filter

Use `fdesign.arbmag` to design a multiband constrained FIR filter.

Force the frequency response at 0.15π rad/sample to 0 dB.

```
d = fdesign.arbmag('N,B,F,A,C',82,2);
d.B1Frequencies = [0 0.06 0.1];
d.B1Amplitudes = [0 0 0];
d.B2Frequencies = [0.15 1];
d.B2Amplitudes = [1 1];
```

Design a filter with no constraints.

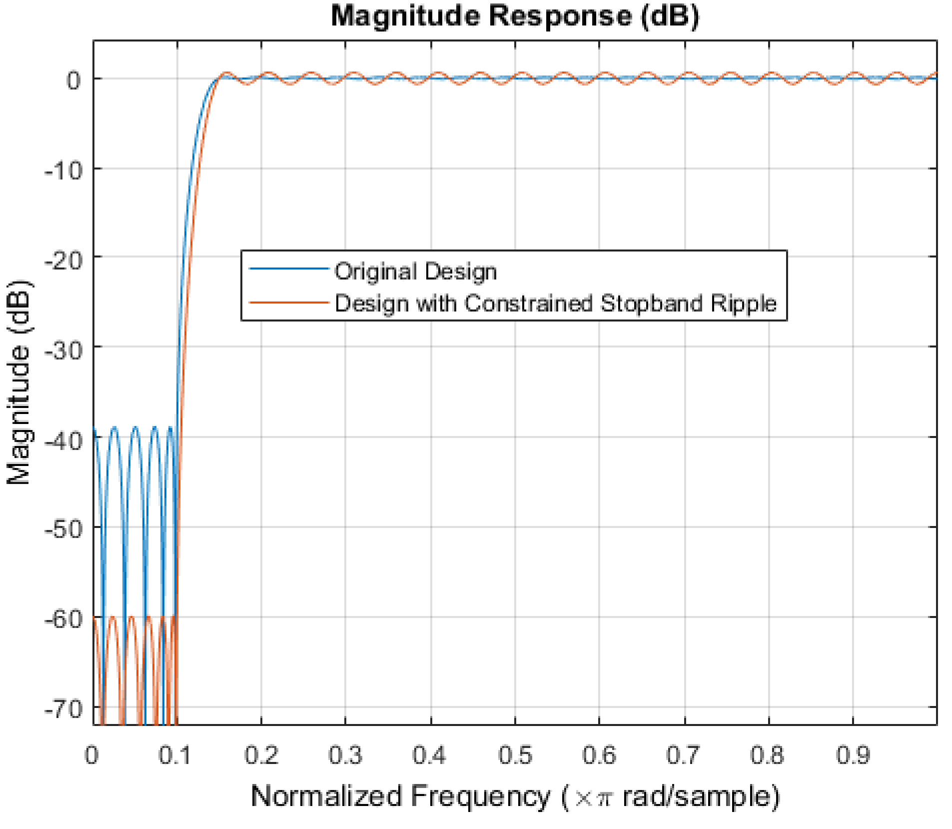
```
Hd1 = design(d,'equiripple','B2ForcedFrequencyPoints',0.15);
```

Add a constraint to the first band to increase attenuation.

```
d.B1Constrained = true;
d.B1Ripple = 0.001;
Hd2 = design(d,'equiripple','B2ForcedFrequencyPoints',0.15);
```

Visualize the frequency response.

```
hfvt = fvtool(Hd1,Hd2);
legend(hfvt,'Original Design','Design with Constrained Stopband Ripple')
```



See Also

design | designmethods | fdesign

Introduced in R2009a

fdesign.arbmagnphase

Arbitrary response magnitude and phase filter specification object

Syntax

```
d = fdesign.arbmagnphase
d = fdesign.arbmagnphase(specification)
d = fdesign.arbmagnphase(specification,specvalue1,specvalue2,...)
d = fdesign.arbmagnphase(specvalue1,specvalue2,specvalue3)
d = fdesign.arbmagnphase(...,fs)
```

Description

`d = fdesign.arbmagnphase` constructs an arbitrary magnitude filter specification object `d`.

`d = fdesign.arbmagnphase(specification)` initializes the `Specification` property for specifications object `d` to `specification`. The input argument `specification` must be one of the choices shown in the following table. Specification options are not case sensitive.

Specification	Description of Resulting Filter
`n,f,h`	Single band design (default). FIR and IIR (`n` is the order for both numerator and denominator).
`n,b,f,h`	FIR multiband design where `b` defines the number of bands.
`nb,na,f,h`	IIR single band design.

The following table describes the specification arguments.

Argument	Description
`b`	Number of bands in the multiband filter.
`f`	Frequency vector. Frequency values specified in `f` indicate locations where you provide specific filter response amplitudes. When you provide `f` you must also provide `h` which contains the response values.

Argument	Description
h	Complex frequency response values.
n	Filter order for FIR filters and the numerator and denominator orders for IIR filters (when not specified by nb and na).
nb	Numerator order for IIR filters.
na	Denominator order for IIR filter designs.

By default, this method assumes that all frequency specifications are supplied in normalized frequency.

Specifying f and h

f and h are the input arguments you use to define the filter response desired. Each frequency value you specify in f must have a corresponding response value in h. This example creates a filter with two passbands (b = 4) and shows how f and h are related. This example is for illustration only. It is not an actual filter.

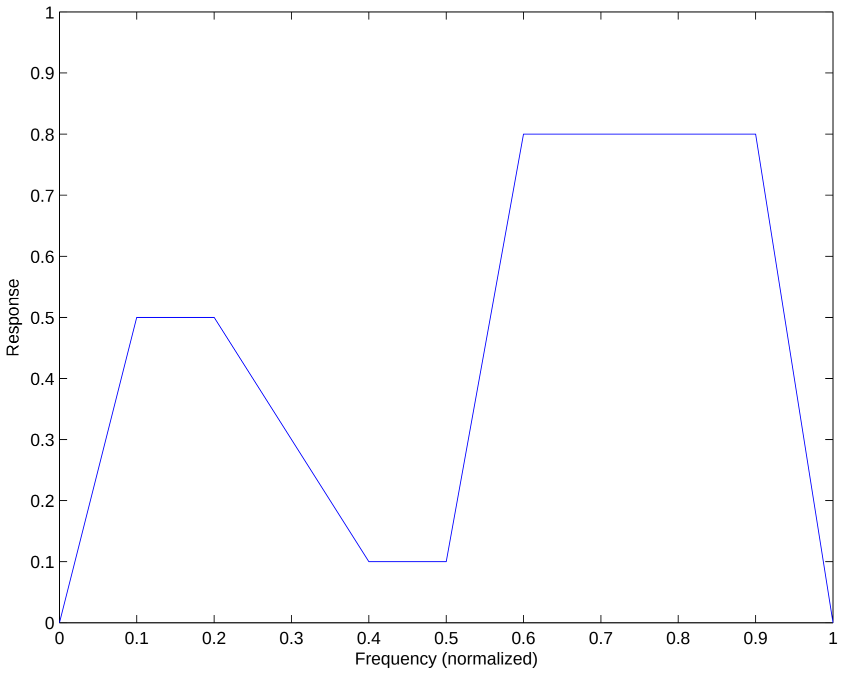
Define the frequency vector f as `[0 0.1 0.2 0.4 0.5 0.6 0.9 1.0]`

Define the response vector h as `[0 0.5 0.5 0.1 0.1 0.8 0.8 0]`

These specifications connectf and h as shown in the following table.

f (Normalized Frequency)	h (Response Desired at f)
0	0
0.1	0.5
0.2	0.5
0.4	0.1
0.5	0.1
0.6	0.8
0.9	0.8
1.0	0.0

A response with two passbands—one roughly between 0.1 and 0.2 and the second between 0.6 and 0.9—results from the mapping between f and h. Plotting f and h yields the following figure that resembles a filter with two passbands.



The second example in Examples shows this plot in more detail with a complex filter response for `h`. In the example, `h` uses complex values for the response.

Different specification types often have different design methods available. Use `designmethods(d)` to get a list of design methods available for a given specification option and specifications object.

`d = fdesign.arbmagnphase(specification,specvalue1,specvalue2,...)` initializes the filter specification object with `specvalue1`, `specvalue2`, and so on. Use `get(d,'description')` for descriptions of the various specifications `specvalue1`, `specvalue2`, ...`spec`*n*.

`d = fdesign.arbmagnphase(specvalue1,specvalue2,specvalue3)` uses the default specification option `n,f,h`, setting the filter order, filter frequency vector, and the complex frequency response vector to the values `specvalue1`, `specvalue2`, and `specvalue3`.

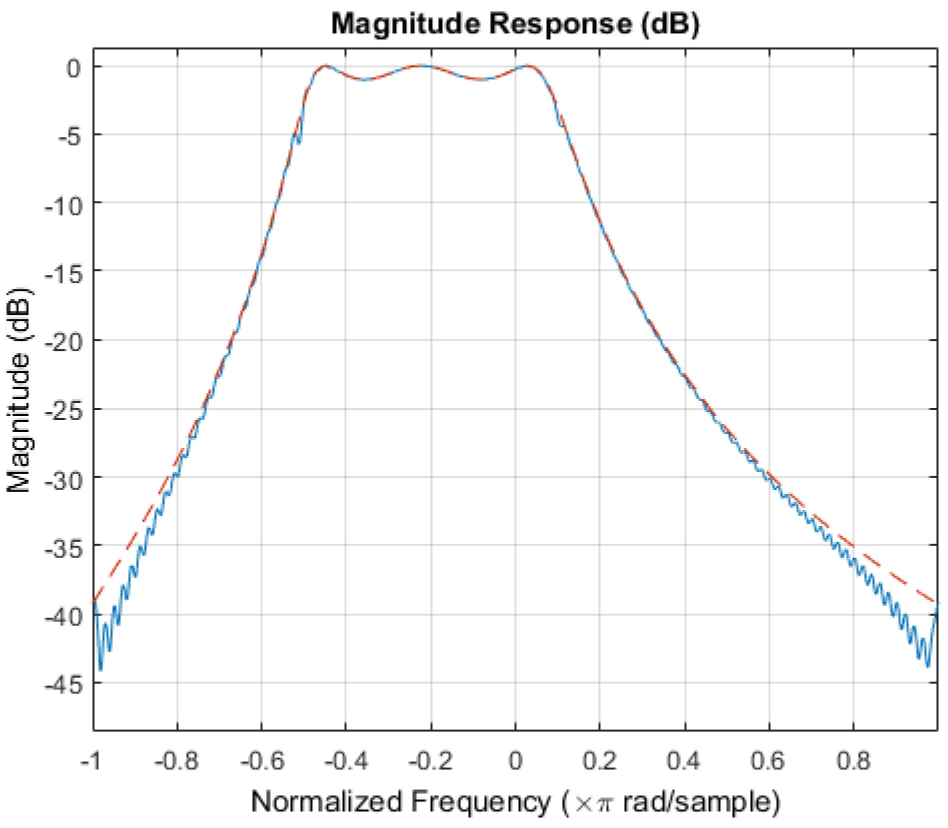
`d = fdesign.arbmagnphase(...,fs)` specifies the sampling frequency in Hz. All other frequency specifications are also assumed to be in Hz when you specify `fs`.

Examples

Construct a Complex Analog Filter

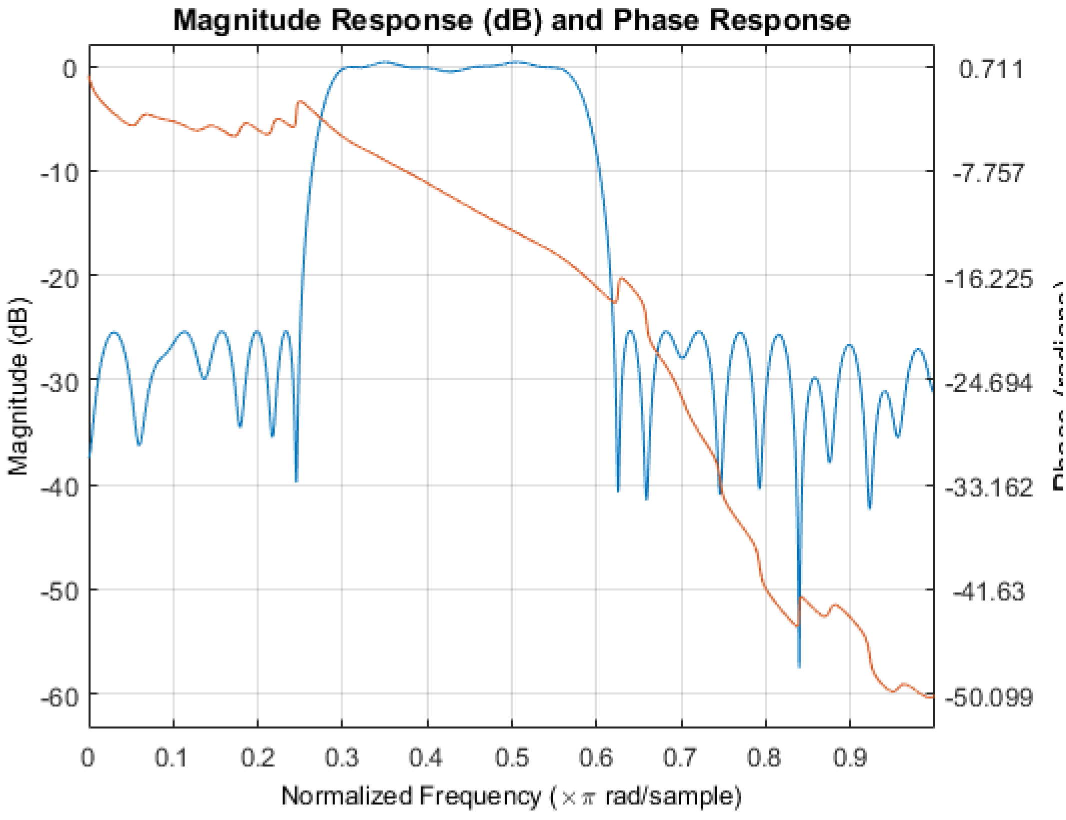
Use `fdesign.arbmagnphase` to model a complex analog filter.

```
d = fdesign.arbmagnphase('n,f,h',100); % N=100, f and h set to defaults.
design(d,'freqsamp','SystemObject',true);
```



For a more complex example, design a bandpass filter with low group delay by specifying the desired delay and using f and h to define the filter bands.

```
n = 50;     % Group delay of a linear phase filter would be 25.
gd = 12;    % Set the desired group delay for the filter.
f1=linspace(0,.25,30); % Define the first stopband frequencies.
f2=linspace(.3,.56,40);% Define the passband frequencies.
f3=linspace(.62,1,30); % Define the second stopband frequencies.
h1 = zeros(size(f1));  % Specify the filter response at the freqs in f1.
h2 = exp(-1j*pi*gd*f2); % Specify the filter response at the freqs in f2.
h3 = zeros(size(f3));  % Specify the response at the freqs in f3.
d=fdesign.arbmagnphase('n,b,f,h',50,3,f1,h1,f2,h2,f3,h3);
D = design(d,'equiripple','SystemObject',true);
fvtool(D,'Analysis','freq');
```



See Also

fdesign | design | designmethods | setspecs

Introduced in R2011a

fdesign.audioweighting

Audio weighting filter specification object

Syntax

```
HAwf = fdesign.audioweighting
HAwf = fdesign.audioweighting(spec)
HAwf = fdesign.audioweighting(spec,specvalue1,specvalue2)
HAwf = fdesign.audioweighting(specvalue1,specvalue2)
HAwf = fdesign.audioweighting(...,Fs)
```

Description

Supported audio weighting filter types are: A weighting, C weighting, C-message, ITU-T 0.41, and ITU-R 468–4 weighting.

`HAwf = fdesign.audioweighting` constructs an audio weighting filter specification object *HAwf* with a weighting type of A and a filter class of 1. Use the `design` method and set the `'SystemObject'` flag to `true`, to instantiate a System object based on the specifications in *HAwf*. Use `designmethods` to find valid filter design methods. Because the standards for audio weighting filters are in Hz, normalized frequency specifications are not supported for `fdesign.audioweighting` objects. The default sampling frequency for A weighting, C weighting, C-message, and ITU-T 0.41 filters is 48 kHz. The default sampling frequency for the ITU-R 468–4 filter is 80 kHz. If you invoke the `normalizefreq` method, a warning is issued when you instantiate the System object and the default sampling frequencies in Hz are used.

`HAwf = fdesign.audioweighting(spec)` returns an audio weighting filter specification object using default values for the specification string in *spec*. The following are valid entries for *spec*. The entries are not case sensitive.

- `'WT,Class'` (default *spec*)

  The `'WT,Class'` specification is valid for A weighting and C weighting filters of class 1 or 2.

  The weighting type is specified by the string: `'A'` or `'C'`. The class is the scalar 1 or 2.

The default values for `'WT,Class'` are `'A',1`.

- `'WT'`

  The `'WT'` specification is valid for C-message (default), ITU-T 0.41, and ITU-R 468–4 weighting filters.

  The weighting type is specified by the string: `'Cmessage'`, `'ITUT041'`, or `'ITUR4684'`.

`HAwf = fdesign.audioweighting(spec,specvalue1,specvalue2)` constructs an audio weighting filter specification object `HAwf` and sets its specifications at construction time.

`HAwf = fdesign.audioweighting(specvalue1,specvalue2)` constructs an audio weighting filter specification object `HAwf` with the specification `'WT,Class'` using the values you provide. The valid weighting types are `'A'` or `'C'`.

`HAwf = fdesign.audioweighting(...,Fs)` specifies the sampling frequency in Hz. The sampling frequency is a scalar trailing all other input arguments.

Input Arguments

Parameter Name/Value Pairs

`'WT'`

Weighting type

The weighting type defines the frequency response of the filter. The valid weighting types are: A weighting, C weighting, C-message, ITU-T 0.41, and ITU-R 468–4 weighting. The weighting types are described in “Definitions” on page 5-550.

`'Class'`

Filter Class

Filter class is only applicable for A weighting and C weighting filters. The filter class describes the frequency-dependent tolerances specified in the relevant standards [1],

[2]. There are two possible class values: 1 and 2. Class 1 weighting filters have stricter tolerances than class 2 filters. The filter class value does not affect the design. The class value is only used to provide a specification mask in `fvtool` for the analysis of the filter design.

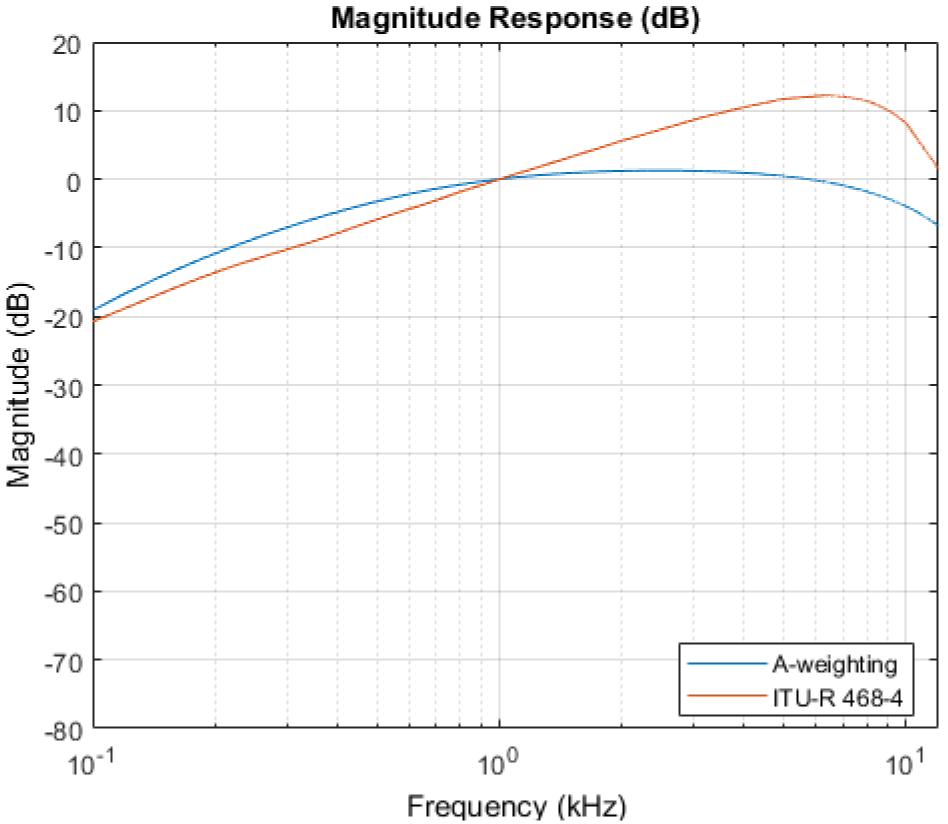
Default: 1

Examples

Compare Class 1 A and ITU-R 468-4 Weighting Filters

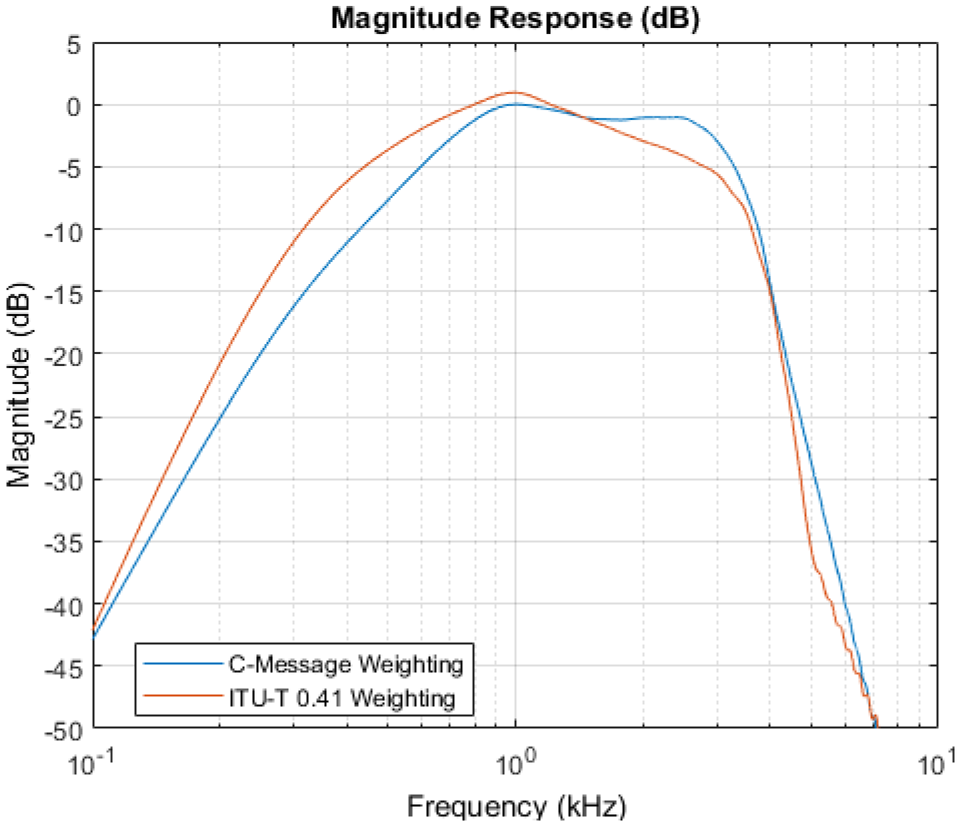
Compare class 1 A weighting and ITU-R 468-4 filters between 0.1 and 12 kHz. % Sampling frequency is 44.1 kHz

```
HawfA = fdesign.audioweighting('WT,Class','A',1,44.1e3);
HawfITUR = fdesign.audioweighting('WT','ITUR4684',44.1e3);
Afilter = design(HawfA,'SystemObject',true);
ITURfilter = design(HawfITUR,'SystemObject',true);
hfvt = fvtool(Afilter,ITURfilter);
axis([0.1 12 -80 20]);
legend(hfvt,'A-weighting','ITU-R 468-4');
```



Compare C-message and ITU-T 0.41 Weighting Filters

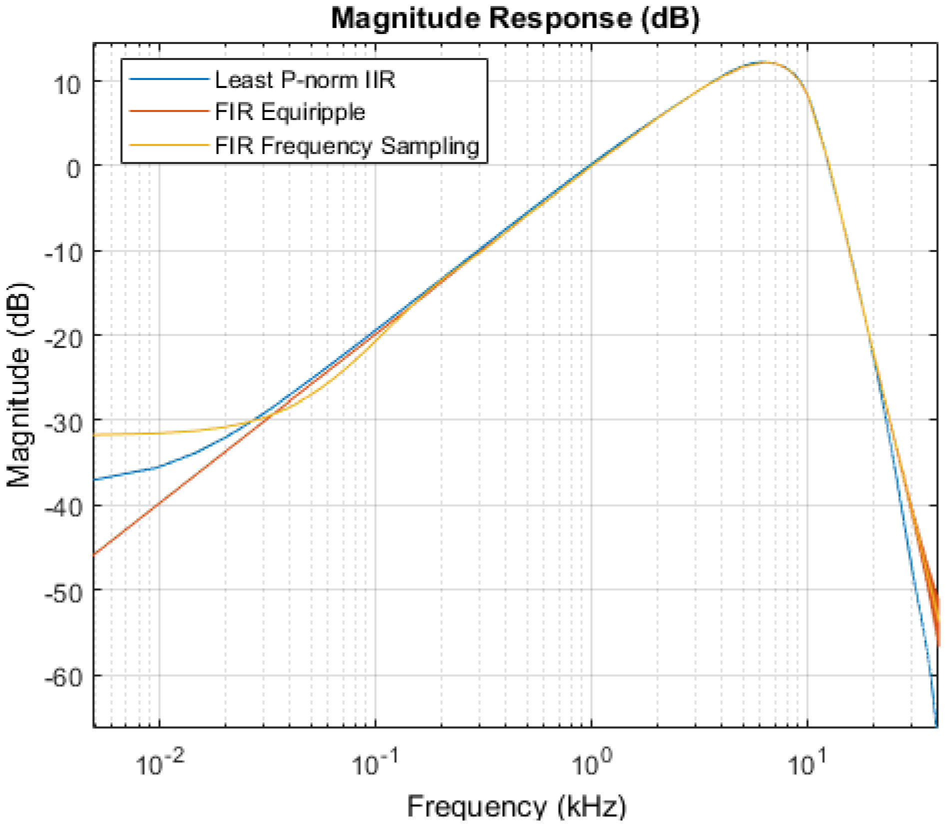
```
hCmessage = fdesign.audioweighting('WT','Cmessage',24e3);
hITUT = fdesign.audioweighting('WT','ITUT041',24e3);
dCmessage = design(hCmessage,'SystemObject',true);
dITUT = design(hITUT,'SystemObject',true);
hfvt = fvtool(dCmessage,dITUT);
legend(hfvt,'C-Message Weighting','ITU-T 0.41 Weighting');
axis([0.1 10 -50 5]);
```



Construct an ITU-R 468-4 Weighting Filter

Construct an ITU-R 468-4 filter using all available design methods.

```
HAwf = fdesign.audioweighting('WT','ITUR4684');
ValidDesigns = designmethods(HAwf);
% returns iirlpnorm,equiripple,freqsamp in cell array
D = design(HAwf,'all','SystemObject',true); % returns all designs
hfvt = fvtool(D{1},D{2},D{3});
legend(hfvt,'Least P-norm IIR','FIR Equiripple',...,
'FIR Frequency Sampling')
```



Definitions

A weighting

The specifications for the A weighting filter are found in ANSI standard S1.42-2001. The A weighting filter is based on the 40–phon Fletcher-Munson equal loudness contour [3]. One phon is equal to one dB sound pressure level (SPL) at one kHz. The Fletcher-Munson equal loudness contours are designed to account for frequency and level dependent differences in the perceived loudness of tonal stimuli. The 40–phon contour reflects the fact that the human auditory system is more sensitive to frequencies around 1–2 kHz than lower and higher frequencies. The filter roll off is more pronounced at lower frequencies and more modest at higher frequencies. While A weighting is based on the equal loudness contour for low-level (40–phon) tonal stimuli, it is commonly used in the United States for assessing potential health risks associated with noise exposure to narrowband and broadband stimuli at high levels.

C weighting

The specifications for the C weighting filter are found in ANSI standard S1.42-2001. The C weighting filter approximates the 100–phon Fletcher-Munson equal loudness contour for tonal stimuli. The C weighting magnitude response is essentially flat with 3–dB frequencies at 31.5 Hz and 8000 Hz. While C weighting is not as common as A weighting, sound level meters frequently have a C weighting filter option.

C-message

The specifications for the C–message weighting filter are found in Bell System Technical Reference, PUB 41009. C-message weighting filters are designed for measuring the impact of noise on telecommunications circuits used in speech transmission [6]. C-message weighting filters are commonly used in North America, while the ITU-T 0.41 filter is more commonly used outside of North America.

ITU-R 468–4

The specifications for the ITU-R 486–4 weighting filter are found in the International Telecommunication Union Recommendation ITU-R BS.468-4. ITU-R 486–4 is an equal loudness contour weighting filter. Unlike the A weighting filter, the ITU-R 468–4 filter describes subjective loudness judgements for broadband stimuli [4]. A common criticism of the A weighting filter is that it underestimates the loudness judgement of real-world stimuli particularly in the frequency band from about 1–9 kHz. A comparison of A weighting and ITU-R 468–4 weighting curves shows that the ITU-R 468–4 curve applies more gain between 1 and 10 kHz with a peak difference of approximately 12 dB around 6–7 kHz.

ITU-T 0.41

The specifications for the ITU-T 0.41 filter are found in the ITU-T Recommendation 0.41. ITU-T 0.41 weighting filters are designed for measuring the impact of noise on telecommunications circuits used in speech transmission [5]. ITU-T 0.41 weighting filters are commonly used outside of North America, while the C-message weighting filter is more common in North America.

References

[1] *American National Standard Design Response of Weighting Networks for Acoustical Measurements*, ANSI S1.42-2001, Acoustical Society of America, New York, NY, 2001.

[2] *Electroacoustics Sound Level Meters Part 1: Specifications*, IEC 61672-1, First Edition 2002-05.

[3] Fletcher, H. and W.A. Munson. “Loudness, its definition, measurement and calculation.” *Journal of the Acoustical Society of America*, Vol. 5, 1933, pp. 82–108.

[4] *Measurement of Audio-Frequency Noise Voltage Level in Sound Broadcasting*, International Telecommunication Union Recommendation ITU-R BS.468-4, 1986.

[5] *Psophometer for Use on Telephone-Type Circuits*, ITU-T Recommendation 0.41.

[6] *Transmission Parameters Affecting Voiceband Data Transmission-Measuring Techniques*, Bell System Technical Reference, PUB 41009, 1972.

See Also

`design` | `designmethods` | `fdesign` | `fvtool`

Topics

Audio Weighting Filters Example
“Design a Filter in Fdesign — Process Overview”

Introduced in R2011a

fdesign.bandpass

Bandpass filter specification object

Syntax

```
D = fdesign.bandpass
D = fdesign.bandpass(SPEC)
D = fdesign.bandpass(spec,specvalue1,specvalue2,...)
D = fdesign.bandpass(specvalue1,specvalue2,specvalue3,
specvalue4,...specvalue4,specvalue5,specvalue6)
D = fdesign.bandpass(...,Fs)
D = fdesign.bandpass(...,MAGUNITS)
```

Description

`D = fdesign.bandpass` constructs a bandpass filter specification object `D`, applying default values for the properties `Fstop1`, `Fpass1`, `Fpass2`, `Fstop2`, `Astop1`, `Apass`, and `Astop2` — one possible set of values you use to specify a bandpass filter.

`D = fdesign.bandpass(SPEC)` constructs object `D` and sets its `Specification` property to `SPEC`. Entries in the `SPEC` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `SPEC` are shown below and used to define the bandpass filter. These entries are not case sensitive.

Note: Specifications marked with an asterisk require the DSP System Toolbox software.

- `'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'` (default `spec`)
- `'N,F3dB1,F3dB2'`
- `"N,F3dB1,F3dB2,Ap'` *
- `'N,F3dB1,F3dB2,Ast'` *
- `'N,F3dB1,F3dB2,Ast1,Ap,Ast2'` *
- `'N,F3dB1,F3dB2,BWp` *

- 'N,F3dB1,F3dB2,BWst' *
- 'N,Fc1,Fc2'
- 'N,Fc1,Fc2,Ast1,Ap,Ast2'
- 'N,Fp1,Fp2,Ap'
- 'N,Fp1,Fp2,Ast1,Ap,Ast2'
- 'N,Fst1,Fp1,Fp2,Fst2'
- 'N,Fst1,Fp1,Fp2,Fst2,C' *
- 'N,Fst1,Fp1,Fp2,Fst2,Ap' *
- 'N,Fst1,Fst2,Ast'
- 'Nb,Na,Fst1,Fp1,Fp2,Fst2' *

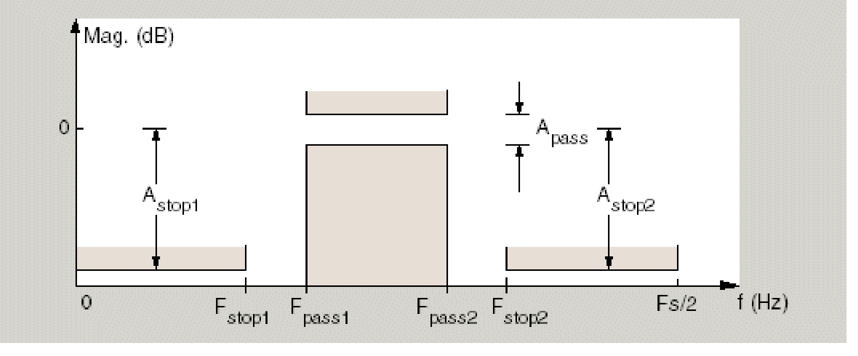
The filter specifications are defined as follows:

- Ap — amount of ripple allowed in the pass band. Also called Apass.
- Ast1 — attenuation in the first stop band in decibels (the default units). Also called Astop1.
- Ast2 — attenuation in the second stop band in decibels (the default units). Also called Astop2.
- BWp — bandwidth of the filter passband. Specified in normalized frequency units.
- BWst — bandwidth of the filter stopband. Specified in normalized frequency units.
- C — Constrained band flag. This enables you to specify passband ripple or stopband attenuation for fixed-order designs in one or two of the three bands.

  In the specification 'N,Fst1,Fp1,Fp2,Fst2,C', you cannot specify constraints in both stopbands and the passband simultaneously. You can specify constraints in any one or two bands.
- F3dB1 — cutoff frequency for the point 3 dB point below the passband value for the first cutoff. Specified in normalized frequency units. (IIR filters)
- F3dB2 — cutoff frequency for the point 3 dB point below the passband value for the second cutoff. Specified in normalized frequency units. (IIR filters)
- Fc1 — cutoff frequency for the point 6 dB point below the passband value for the first cutoff. Specified in normalized frequency units. (FIR filters)
- Fc2 — cutoff frequency for the point 6 dB point below the passband value for the second cutoff. Specified in normalized frequency units. (FIR filters)

- `Fp1` — frequency at the edge of the start of the pass band. Specified in normalized frequency units. Also called Fpass1.
- `Fp2` — frequency at the edge of the end of the pass band. Specified in normalized frequency units. Also called Fpass2.
- `Fst1` — frequency at the edge of the start of the first stop band. Specified in normalized frequency units. Also called Fstop1.
- `Fst2` — frequency at the edge of the start of the second stop band. Specified in normalized frequency units. Also called Fstop2.
- `N` — filter order for FIR filters. Or both the numerator and denominator orders for IIR filters when na and nb are not provided.
- `Na` — denominator order for IIR filters
- `Nb` — numerator order for IIR filters

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like `Fst1` and `Fp1` are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a bandpass filter specification object change depending on the `Specification`. Use `designmethods` to determine which design methods apply to an object and the `Specification` property value.

Use `designopts` to determine the design options for a given design method. Enter `help(D,METHOD)` at the MATLAB command line to obtain detailed help on the design options for a given design method, `METHOD`.

`D = fdesign.bandpass(spec,specvalue1,specvalue2,...)` constructs an object `D` and sets its specifications at construction time.

`D = fdesign.bandpass(specvalue1,specvalue2,specvalue3, specvalue4,...specvalue4,specvalue5,specvalue6)` constructs `D`with the default `Specification` property, using the values you provide as input arguments for `specvalue1,specvalue2,specvalue3,specvalue4,specvalue4,specvalue5, specvalue6` and `specvalue7`.

`D = fdesign.bandpass(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`D = fdesign.bandpass(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units
- `'dB'` — specify the magnitude in dB (decibels)
- `'squared'` — specify the magnitude in power units

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

Bandpass Filtering of Sinusoids

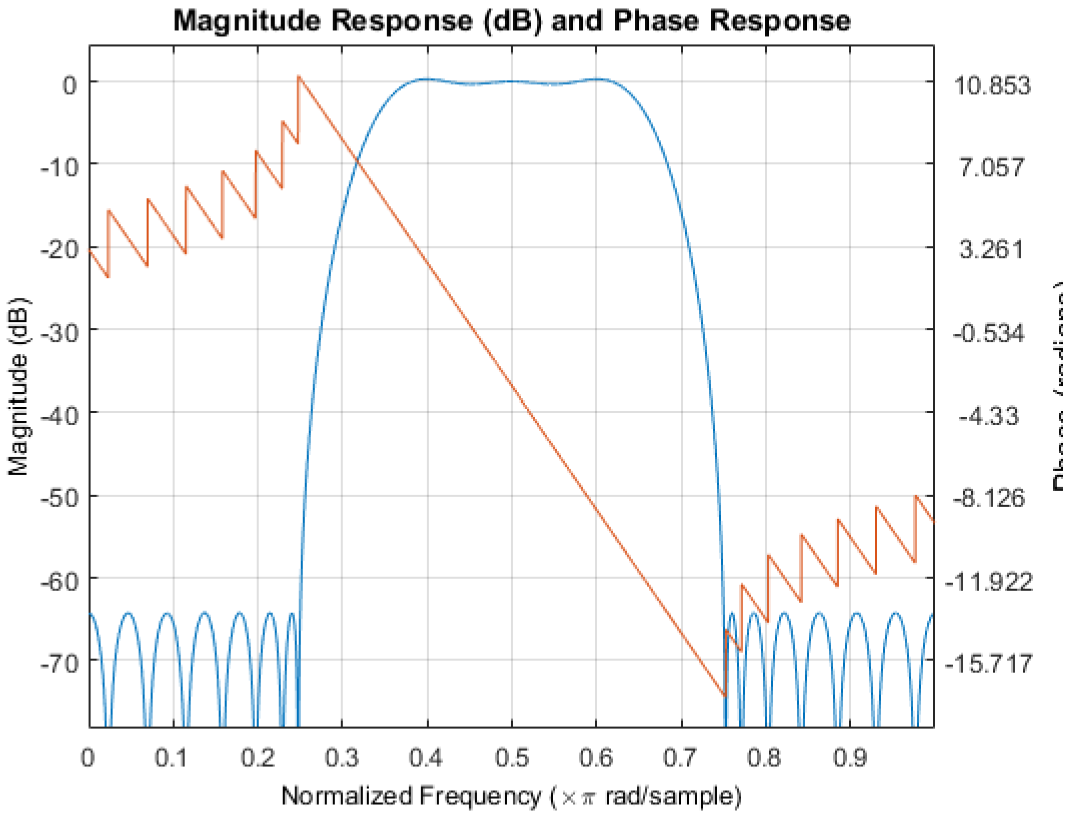
Filter a discrete-time signal with a bandpass filter. The signal is a sum of three discrete-time sinusoids, $\pi/8$, $\pi/2$, and 3π/4 rad/sample.

```
n = 0:159;
x = cos(pi/8*n)+cos(pi/2*n)+sin(3*pi/4*n);
```

Design an FIR equiripple bandpass filter to remove the lowest and highest discrete-time sinusoids. View the frequency response of the filter.

```
d = fdesign.bandpass('Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2', ...
    1/4,3/8,5/8,6/8,60,1,60);
```

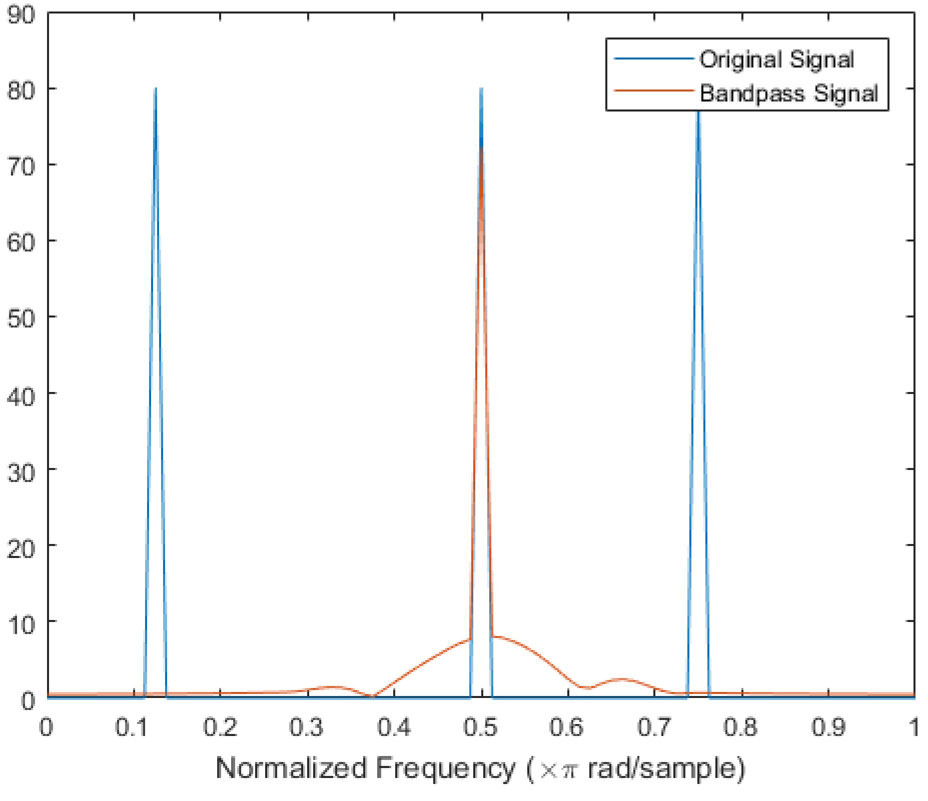
```
Hd = design(d,'equiripple');

freqz(Hd)
```



Apply the filter to the discrete-time signal. Plot the original and filtered signals in the frequency domain.

```
y = filter(Hd,x);
freq = 0:(2*pi)/length(x):pi;
xdft = fft(x);
ydft = fft(y);

plot(freq/pi,abs(xdft(1:length(x)/2+1)))
hold on
```

```
plot(freq/pi,abs(ydft(1:length(x)/2+1)))
hold off
xlabel('Normalized Frequency (\times\pi rad/sample)')
legend('Original Signal','Bandpass Signal')
```

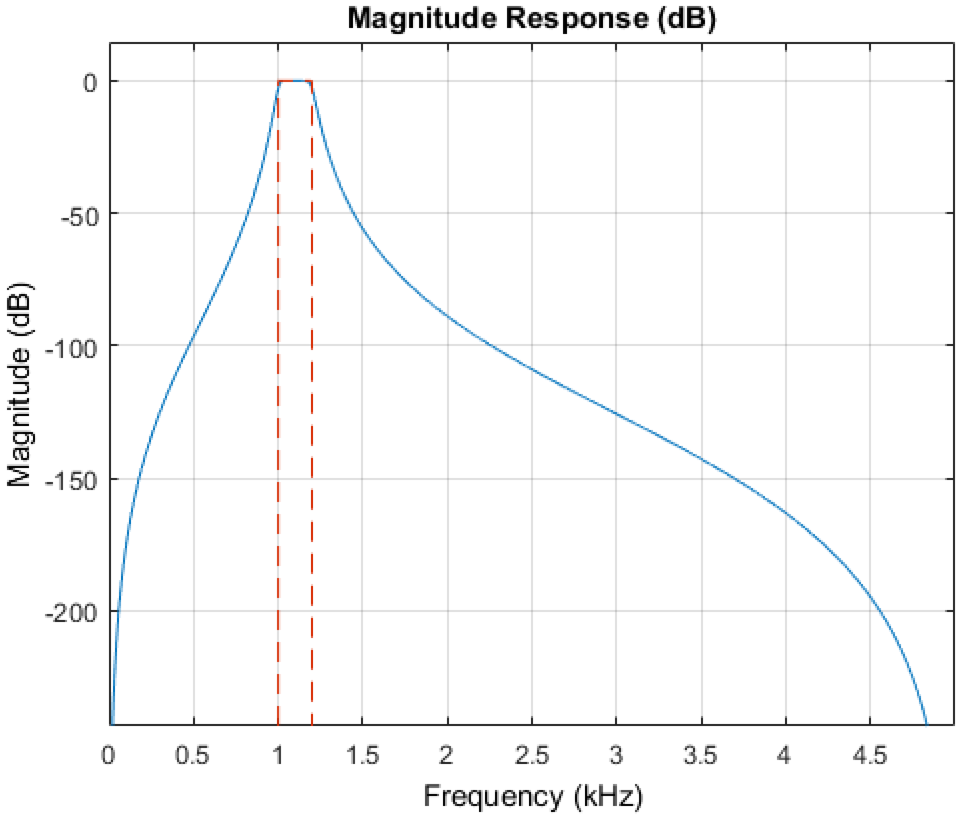


Butterworth Bandpass Filter

Design an IIR Butterworth filter of order 10 with 3-dB frequencies of 1 and 1.2 kHz. The sample rate is 10 kHz. Visualize the frequency response of the filter.

```
d = fdesign.bandpass('N,F3dB1,F3dB2',10,1e3,1.2e3,1e4);
Hd = design(d,'butter');
```

```
fvtool(Hd)
```



Equiripple FIR Passband Filter

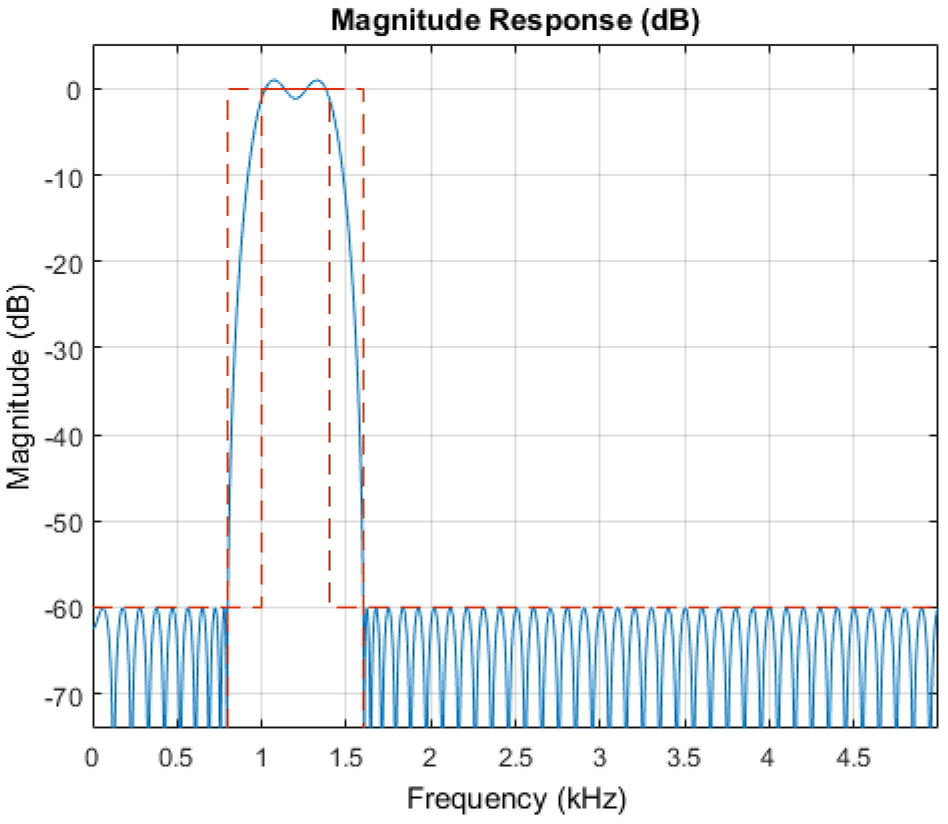
Design a constrained-band FIR equiripple filter of order 100 with a passband of [1, 1.4] kHz. Both stopband attenuation values are constrained to 60 dB. The sample rate is 10 kHz.

```
d = fdesign.bandpass('N,Fst1,Fp1,Fp2,Fst2,C',100,800,1e3,1.4e3,1.6e3,1e4);
d.Stopband1Constrained = true;
d.Astop1 = 60;
d.Stopband2Constrained = true;
d.Astop2 = 60;
```

```
Hd = design(d,'equiripple');
```

Visualize and measure the magnitude response of the filter.

```
fvtool(Hd)
measure(Hd)


ans =

Sample Rate               : 10 kHz
First Stopband Edge       : 800 Hz
First 6-dB Point          : 946.7621 Hz
First 3-dB Point          : 975.1807 Hz
First Passband Edge       : 1 kHz
Second Passband Edge      : 1.4 kHz
Second 3-dB Point         : 1.4248 kHz
Second 6-dB Point         : 1.4533 kHz
Second Stopband Edge      : 1.6 kHz
First Stopband Atten.     : 60.0614 dB
Passband Ripple           : 2.1443 dB
Second Stopband Atten.    : 60.0399 dB
First Transition Width    : 200 Hz
Second Transition Width   : 200 Hz
```



The passband ripple is slightly over 2 dB. Because the design constrains both stopbands, you cannot constrain the passband ripple.

See Also

See Also

`fdesign` | `fdesign.bandstop` | `fdesign.highpass` | `fdesign.lowpass`

Introduced in R2009a

fdesign.bandstop

Bandstop filter specification object

Syntax

```
D = fdesign.bandstop
D = fdesign.bandstop(SPEC)
D = fdesign.bandstop(SPEC,specvalue1,specvalue2,...)
D = fdesign.bandstop(specvalue1,specvalue2,specvalue3,specvalue4,...
specvalue5,specvalue6,specvalue7)
D = fdesign.bandstop(...,Fs)
D = fdesign.bandstop(...,MAGUNITS)
```

Description

`D = fdesign.bandstop` constructs a bandstop filter specification object `D`, applying default values for the properties `Fpass1`, `Fstop1`, `Fstop2`, `Fpass2`, `Apass1`, `Astop1` and `Apass2`.

`D = fdesign.bandstop(SPEC)` constructs object `D` and sets the `Specification` property to `SPEC`. Entries in `SPEC` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `SPEC` are shown below. These entries are not case sensitive.

Note: Specification options marked with an asterisk require the DSP System Toolbox software.

- `'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'` (default spec)
- `'N,F3dB1,F3dB2'`
- `'N,F3dB1,F3dB2,Ap'` *
- `'N,F3dB1,F3dB2,Ap,Ast'` *
- `'N,F3dB1,F3dB2,Ast'` *

- 'N,F3dB1,F3dB2,BWp' *
- 'N,F3dB1,F3dB2,BWst' *
- 'N,Fc1,Fc2'
- 'N,Fc1,Fc2,Ap1,Ast,Ap2'
- 'N,Fp1,Fp2,Ap'
- 'N,Fp1,Fp2,Ap,Ast'
- 'N,Fp1,Fst1,Fst2,Fp2'
- 'N,Fp1,Fst1,Fst2,Fp2,C' *
- 'N,Fp1,Fst1,Fst2,Fp2,Ap' *
- 'N,Fst1,Fst2,Ast'
- 'Nb,Na,Fp1,Fst1,Fst2,Fp2' *

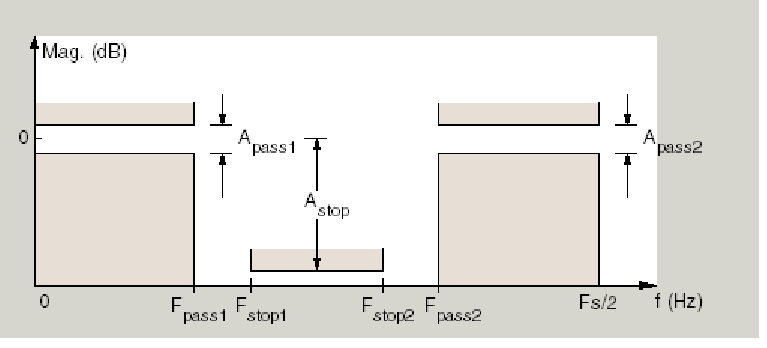
The filter specifications are defined as follows:

- Ap — amount of ripple allowed in the passband in decibels (the default units). Also called Apass.
- Ap1 — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass1.
- Ap2 — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass2.
- Ast — attenuation in the first stopband in decibels (the default units). Also called Astop1.
- BWp — bandwidth of the filter passband. Specified in normalized frequency units.
- BWst — bandwidth of the filter stopband. Specified in normalized frequency units.
- C — Constrained band flag. This enables you to specify passband ripple or stopband attenuation for fixed-order designs in one or two of the three bands.

  In the specification 'N,Fp1,Fst1,Fst2,Fp2,C', you cannot specify constraints simultaneously in both passbands and the stopband. You can specify constraints in any one or two bands.
- F3dB1 — cutoff frequency for the point 3 dB point below the passband value for the first cutoff.
- F3dB2 — cutoff frequency for the point 3 dB point below the passband value for the second cutoff.

- `Fc1` — cutoff frequency for the point 6 dB point below the passband value for the first cutoff. (FIR filters)
- `Fc2` — cutoff frequency for the point 6 dB point below the passband value for the second cutoff. (FIR filters)
- `Fp1` — frequency at the start of the pass band. Also called Fpass1.
- `Fp2` — frequency at the end of the pass band. Also called Fpass2.
- `Fst1` — frequency at the end of the first stop band. Also called Fstop1.
- `Fst2` — frequency at the start of the second stop band. Also called Fstop2.
- `N` — filter order.
- `Na` — denominator order for IIR filters.
- `Nb` — numerator order for IIR filters.

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like `Fp1` and `Fst1` are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a bandstop filter specification object change depending on the `Specification`. Use `designmethods` to determine which design methods apply to an object and the `Specification` property value.

Use `designopts` to determine the design options for a given design method. Enter `help(D,METHOD)` at the MATLAB command line to obtain detailed help on the design options for a given design method, `METHOD`.

`D = fdesign.bandstop(SPEC,specvalue1,specvalue2,...)` constructs an object `D` and sets its specifications at construction time.

`D = fdesign.bandstop(specvalue1,specvalue2,specvalue3,specvalue4,... specvalue5,specvalue6,specvalue7)` constructs an object `D` with the default `Specification` property, using the values you provide in `specvalue1,specvalue2,specvalue3,specvalue4,specvalue5, specvalue6` and `specvalue7`.

`D = fdesign.bandstop(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency. If you specify the sampling frequency as a trailing scalar, all frequencies in the specifications are in Hz as well.

`D = fdesign.bandstop(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units
- `'dB'` — specify the magnitude in dB (decibels)
- `'squared'` — specify the magnitude in power units

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

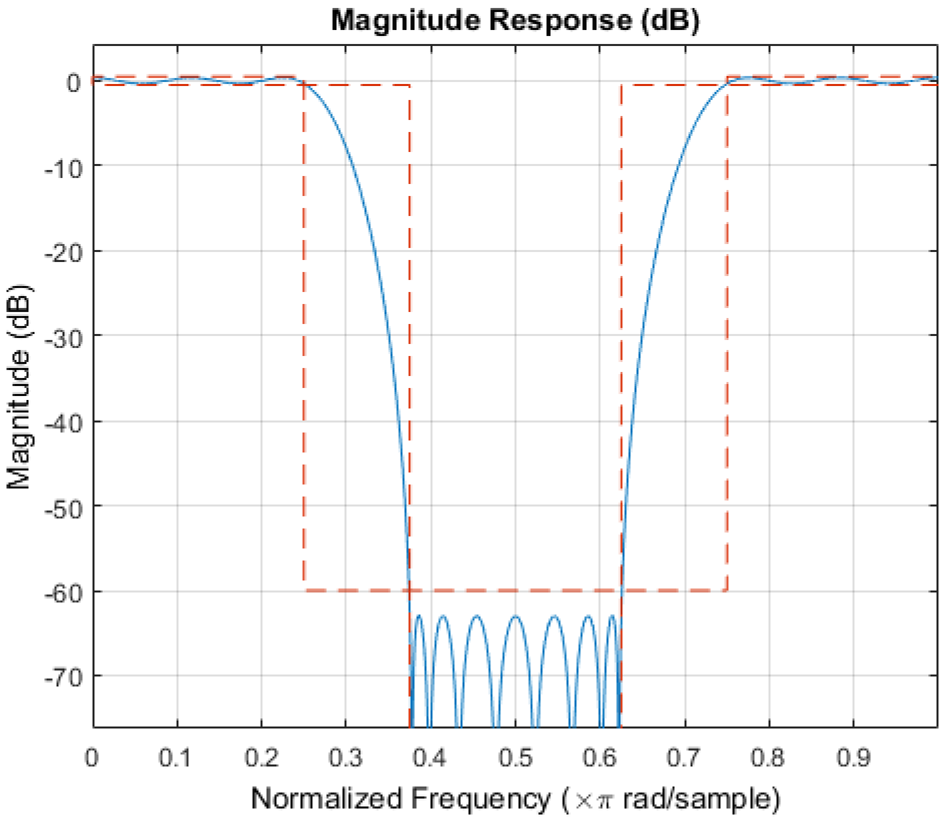
Examples

Bandstop Filtering of Sinusoids

Construct a bandstop filter to reject the discrete frequency band between 3π/8 and 5π/8 rad/sample. Apply the filter to a discrete-time signal consisting of the superposition of three discrete-time sinusoids.

Design an FIR equiripple filter and view its magnitude response.

```
d = fdesign.bandstop('Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2', ...
    2/8,3/8,5/8,6/8,1,60,1);
Hd = design(d,'equiripple');

fvtool(Hd)
```



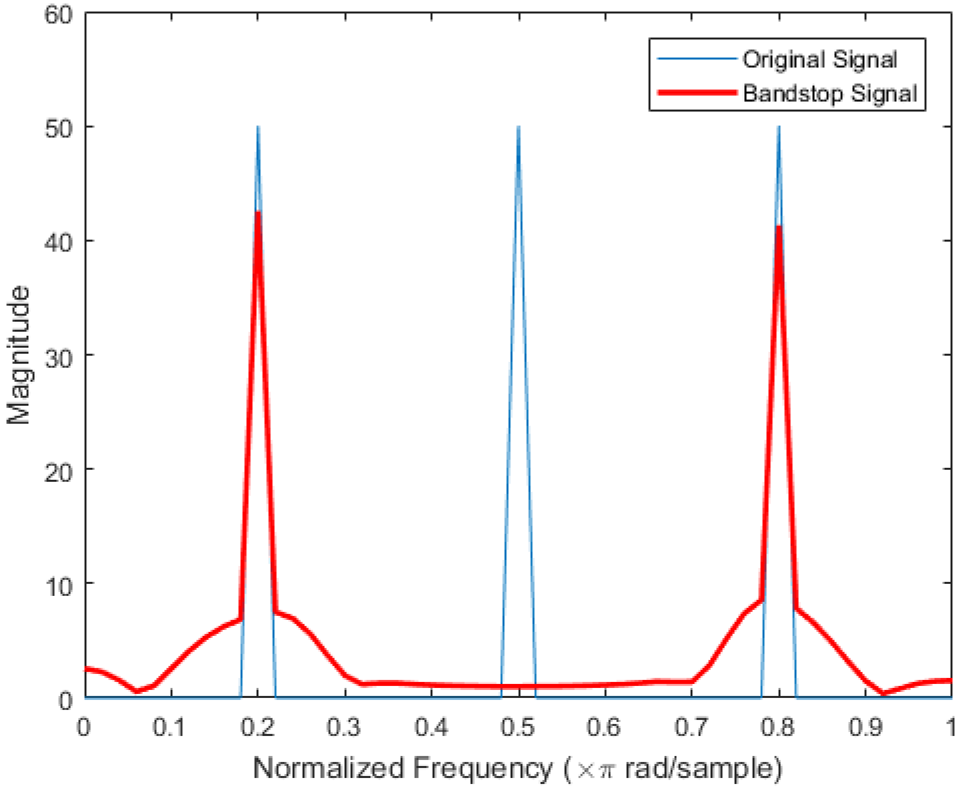
Construct the discrete-time signal and filter it.

```
n = 0:99;
x = cos(pi/5*n)+sin(pi/2*n)+cos(4*pi/5*n);
y = filter(Hd,x);
```

Plot the original and filtered signals in the frequency domain.

```
xdft = fft(x);
ydft = fft(y);
freq = 0:(2*pi)/length(x):pi;

plot(freq/pi,abs(xdft(1:length(x)/2+1)))
hold on
```

```
plot(freq/pi,abs(ydft(1:length(y)/2+1)),'r','linewidth',2)
hold off
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Magnitude')
legend('Original Signal','Bandstop Signal')
```

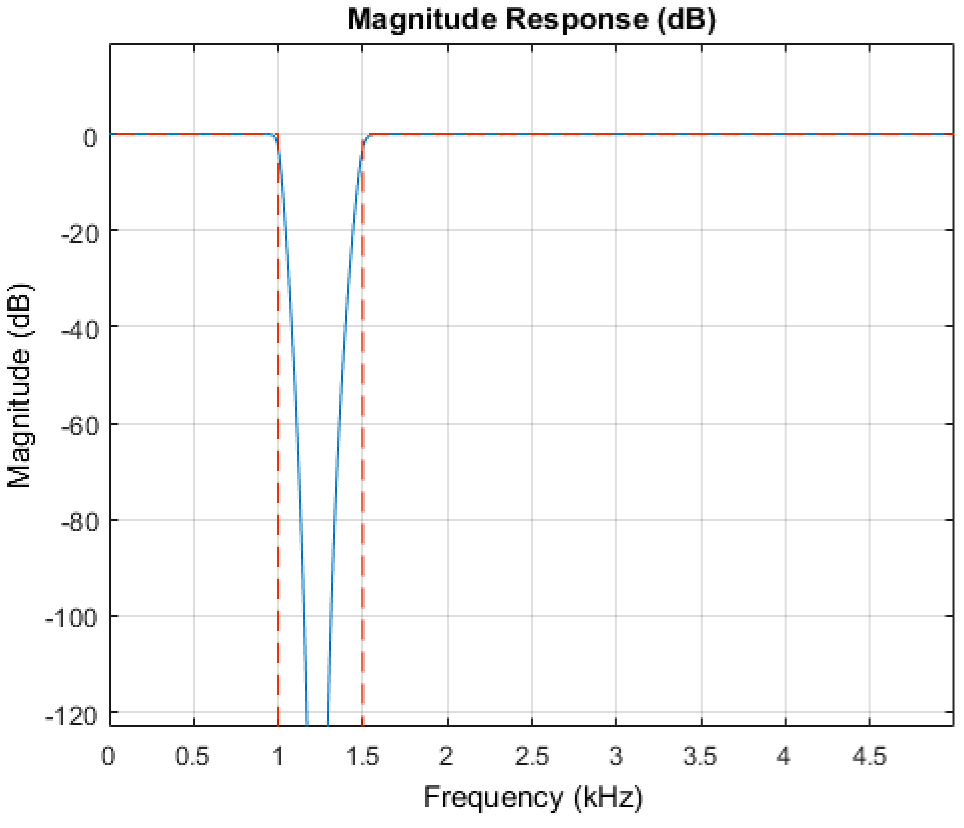


Butterworth Bandstop Filter

Create a Butterworth bandstop filter for data sampled at 10 kHz. The stopband is [1,1.5] kHz. The order of the filter is 20. Visualize the frequency response.

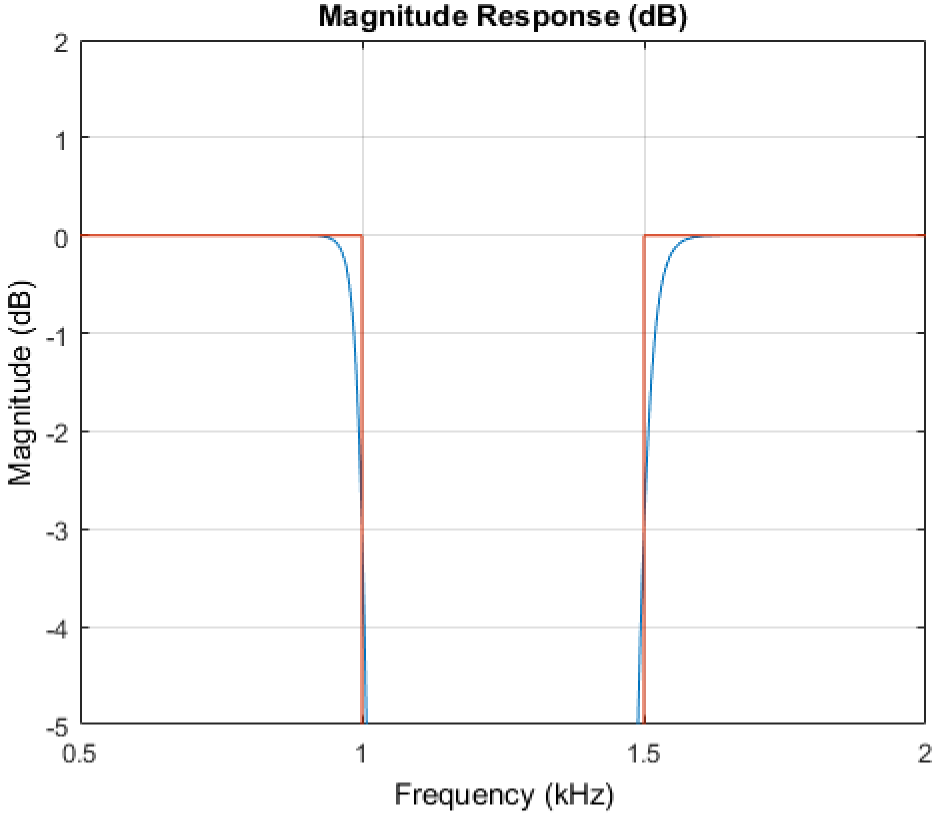
```
d = fdesign.bandstop('N,F3dB1,F3dB2',20,1e3,1.5e3,1e4);
Hd = design(d,'butter');
```

```
fvtool(Hd)
```



Zoom in on the magnitude response plot to verify that the 3-dB down points are located at 1 and 1.5 kHz.

```
axis([0.5 2 -5 2])
```



Equiripple FIR Bandstop Filter

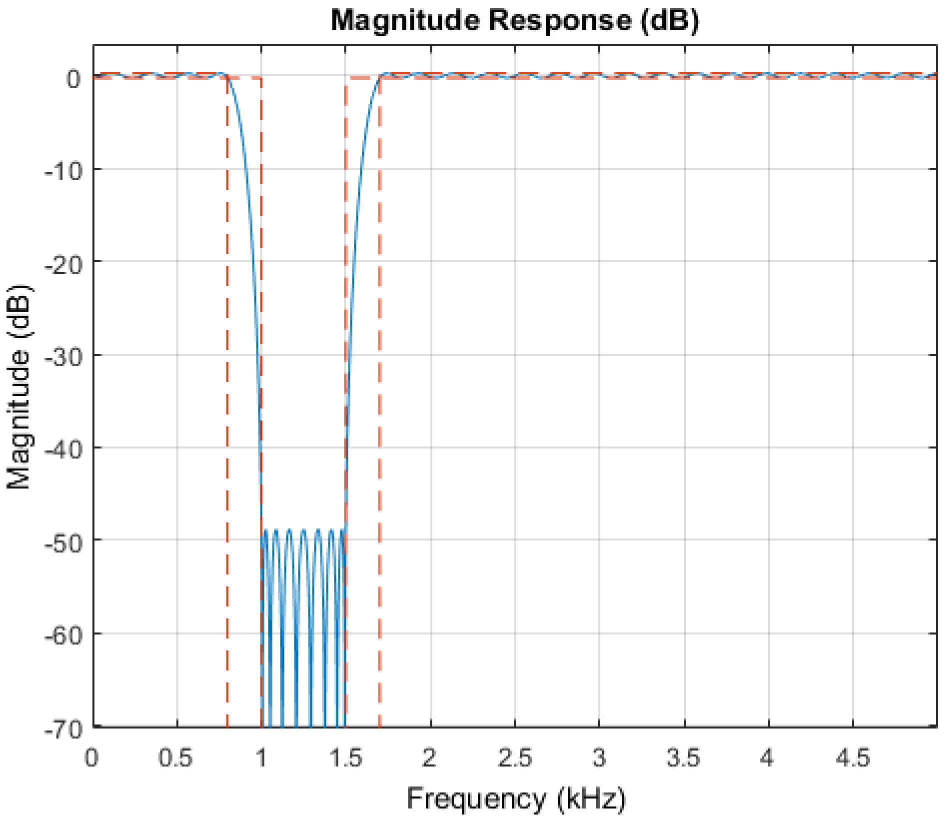
Design a constrained-band FIR equiripple filter of order 100 for data sampled at 10 kHz. You can specify constraints on at most two of the three bands: two passbands and one stopband. In this example, constrain the passband ripple to be 0.5 dB in each passband.

```
d = fdesign.bandstop('N,Fp1,Fst1,Fst2,Fp2,C',100,800,1e3,1.5e3,1.7e3,1e4);
d.Passband1Constrained = true; d.Apass1 = 0.5;
d.Passband2Constrained = true; d.Apass2 = 0.5;
Hd = design(d,'equiripple');
```

Visualize the magnitude response and measure the design.

```
fvtool(Hd)
```

```
measure(Hd)

ans =

Sample Rate             : 10 kHz
First Passband Edge     : 800 Hz
First 3-dB Point        : 852.4565 Hz
First 6-dB Point        : 881.7834 Hz
First Stopband Edge     : 1 kHz
Second Stopband Edge    : 1.5 kHz
Second 6-dB Point       : 1.6185 kHz
Second 3-dB Point       : 1.6478 kHz
Second Passband Edge    : 1.7 kHz
First Passband Ripple   : 0.49642 dB
Stopband Atten.         : 48.8466 dB
Second Passband Ripple  : 0.49634 dB
First Transition Width  : 200 Hz
Second Transition Width : 200 Hz
```



With this order filter and passband ripple constraints, you achieve approximately 50 dB of stopband attenuation.

See Also

See Also

`fdesign` | `fdesign.bandpass` | `fdesign.highpass` | `fdesign.lowpass`

Introduced in R2009a

fdesign.ciccomp

CIC compensator filter specification object

Syntax

```
d= fdesign.ciccomp
d= fdesign.ciccomp(d,nsections,rcic)
d= fdesign.ciccomp(...,spec)
h = fdesign.ciccomp(...,spec,specvalue1,specvalue2,...)
```

Description

`d= fdesign.ciccomp` constructs a CIC compensator specifications object `d`, applying default values for the properties Fpass, Fstop, Apass, and Astop. In this syntax, the filter has two sections and the differential delay is 1.

Using `fdesign.ciccomp` with `design` method creates a System object, if the `'SystemObject'` flag is set to `true`.

`d= fdesign.ciccomp(d,nsections,rcic)` constructs a CIC compensator specifications object with the filter differential delay set to `d`, the number of sections in the filter set to `nsections`, and the CIC rate change factor set to `rcic`. The default values of these parameters are: the differential delay equal to 1, the number of sections equal to 2, and the CIC rate change factor equal to 1.

If the CIC rate change factor is equal to 1, the filter passband response is an inverse sinc that is an approximation to the true inverse passband response of the CIC filter.

If you specify a CIC rate change factor not equal to 1, the filter passband response is an inverse Dirichlet sinc that matches exactly the true inverse passband response of the CIC filter.

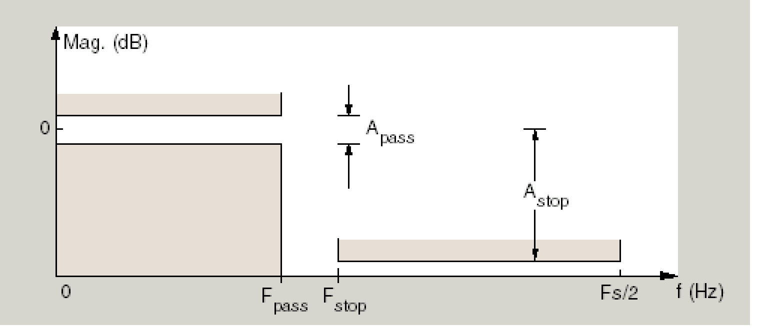
`d= fdesign.ciccomp(...,`*`spec`*`)` constructs a CIC Compensator specifications object and sets its `Specification` property to *`spec`*. Entries in the `spec` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for *`spec`* are shown in the list below. The entries are not case sensitive.

- `'fp,fst,ap,ast'` (default `spec`)
- `'n,fc,ap,ast'`
- `'n,fp,ap,ast'`
- `'n,fp,fst'`
- `'n,fst,ap,ast'`

The filter specifications are defined as follows:

- `ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.
- `ast` — attenuation in the stop band in decibels (the default units). Also called Astop.
- `fc` — cutoff frequency for the point 6 dB point below the passband value. Specified in normalized frequency units.
- `fp` — frequency at the end of the pass band. Specified in normalized frequency units. Also called Fpass.
- `fst` — frequency at the start of the stop band. Specified in normalized frequency units. Also called Fstop.
- `n` — filter order.

In graphic form, the filter specifications look like this:



Regions between specification values like `fp` and `fst` are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a CIC compensator specifications object change depending on the `Specification`. Use `designmethods` to determine which design method applies to an object and its specification.

h = fdesign.ciccomp(...,spec,specvalue1,specvalue2,...) constructs an object and sets the specifications in the order they are specified in the spec input when you construct the object.

Designing CIC Compensators

Typically, when they develop filters, designers want flat passbands and transition regions that are as narrow as possible. CIC filters present a (sinx/x) profile in the passband and relatively wide transitions.

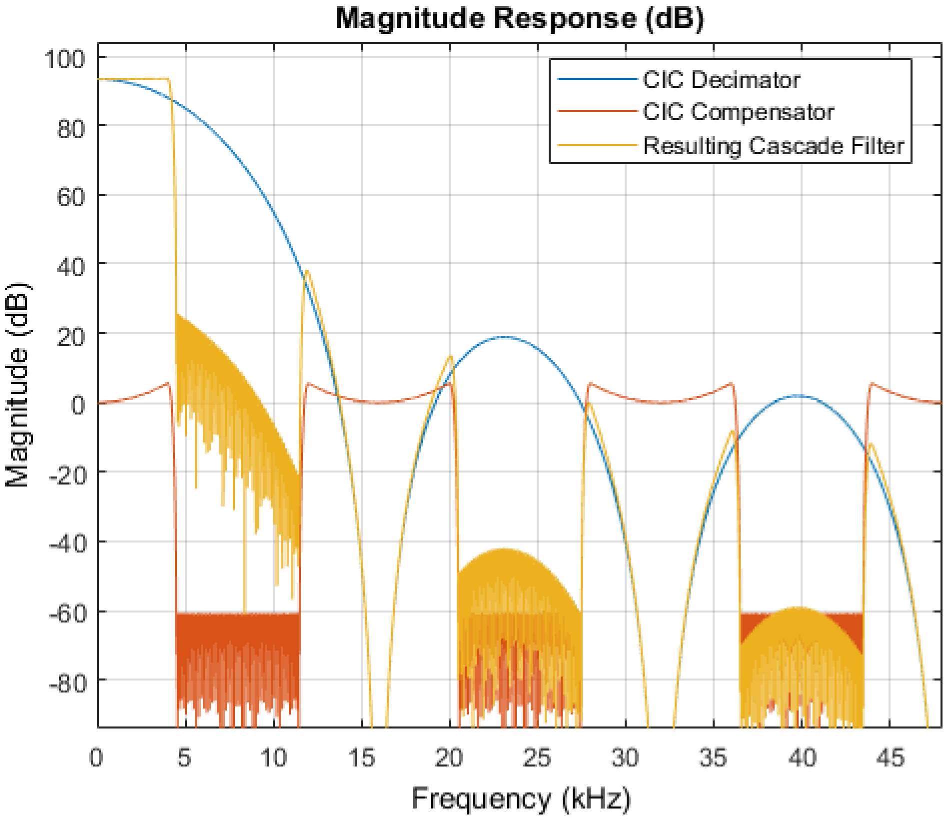
To compensate for this fall off in the passband, and to try to reduce the width of the transition region, you can use a CIC compensator filter that demonstrates an (x/sinx) profile in the passband. fdesign.ciccomp is specifically tailored to designing CIC compensators.

You can design a compensator for CIC filter using differential delay, d, number of sections, numberofsections, and the usable passband frequency, Fpass.

By taking the number of sections, passband, and differential delay from your CIC filter and using them in the definition of the CIC compensator, the resulting compensator filter effectively corrects for the passband droop of the CIC filter, and narrows the transition region.

As a demonstration of this concept, this example creates a CIC decimator and its compensator.

```
fs = 96e3;   % Input sampling frequency.
fpass = 4e3; % Frequency band of interest.
m = 6;  % Decimation factor.
hcic = design(fdesign.decimator(m,'cic',1,fpass,60,fs),'SystemObject',true);
hd = design(fdesign.ciccomp(hcic.DifferentialDelay, ...
            hcic.NumSections,fpass,4.5e3,.1,60,fs/m),'SystemObject',true);
fvtool(hcic,hd,...
cascade(hcic,hd),'ShowReference','off','Fs',[96e3 96e3/m 96e3])
legend('CIC Decimator','CIC Compensator','Resulting Cascade Filter');
```



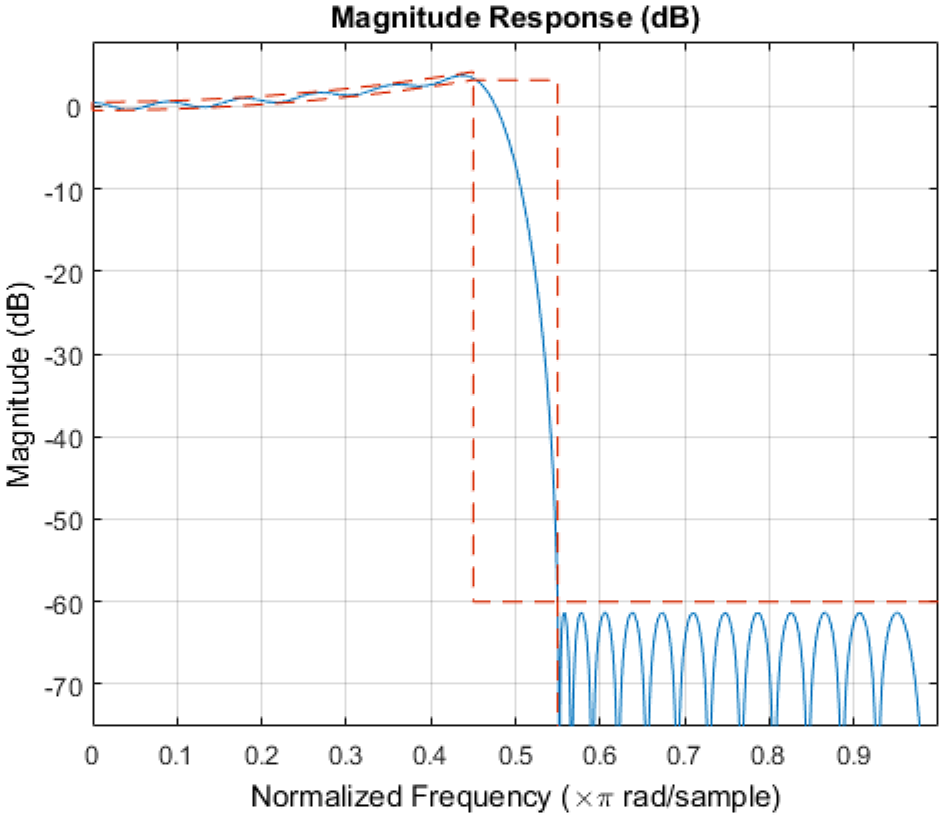
Here is a plot of a CIC filter and a compensator for that filter.

Examples

Design a CIC Compensator Using `fdesign` Object

Designed to compensate for the rolloff inherent in CIC filters, CIC compensators can improve the performance of your CIC design. This example designs a compensator d with five sections and a differential delay equal to one. The plot displayed after the code shows the increasing gain in the passband that is characteristic of CIC compensators, to overcome the droop in the CIC filter passband. Ideally, cascading the CIC compensator with the CIC filter results in a lowpass filter with flat passband response and narrow transition region.

```
h = fdesign.ciccomp;
set(h, 'NumberOfSections', 5, 'DifferentialDelay', 1);
cicComp = design(h,'equiripple','SystemObject',true);
fvtool(cicComp);
```



This compensator would work for a decimator or interpolator that had differential delay of 1 and 5 sections.

See Also

fdesign.decimator | fdesign.interpolator

Introduced in R2011a

fdesign.comb

IIR comb filter specification object

Syntax

```
d=fdesign.comb
d=fdesign.comb(combtype)
d=fdesign(combtype,specstring)
d=fdesign(combtype,specstring,specvalue1,specvalue2,...)
d=fdesign.comb(...,Fs)
```

Description

`fdesign.comb` specifies a peaking or notching comb filter. Comb filters amplify or attenuate a set of harmonically related frequencies.

`d=fdesign.comb` creates a notching comb filter specification object and applies default values for the filter order (`N=10`) and quality factor (`Q=16`).

`d=fdesign.comb(combtype)` creates a comb filter specification object of the specified type and applies default values for the filter order and quality factor. The valid entries for `combtype` are shown in the following table. The entries are not case-sensitive.

Argument	Description
`notch`	creates a comb filter that attenuates a set of harmonically related frequencies.
`peak`	creates a comb filter that amplifies a set of harmonically related frequencies.

`d=fdesign(combtype,`*`specstring`*`)` creates a comb filter specification object of type `combtype` and sets its `Specification` property to *`specstring`* with default values. The entries in *`specstring`* determine the number of peaks or notches in the comb filter as well as their bandwidth and slope. Valid entries for *`specstring`* are shown below. The entries are not case-sensitive.

- `'N,Q'` (default)

- `''N,BW'`
- `'L,BW,GWB,Nsh'`

The following table describes the arguments in *`specstring`*.

Argument	Description
`BW`	Bandwidth of the notch or peak. By default the bandwidth is calculated at the point –3 dB down from the center frequency of the peak or notch. For example, setting `BW=0.01` specifies that the –3 dB point will be +/- 0.005 (in normalized frequency) from the center of the notch or peak.
`GWB`	Gain at which the bandwidth is measured. This allows the user to specify the bandwidth of the notch or peak at a gain different from the –3 dB default.
`L`	Upsampling factor for a shelving filter of order `Nsh`. `L` determines the number of peaks or notches, which are equally spaced over the normalized frequency interval [-1,1].
`N`	Filter order. Specifies a filter with `N+1` numerator and denominator coefficients. The filter will have `N` peaks or notches equally spaced over the interval [–1,1] .
`Nsh`	Shelving filter order. `Nsh` represents a positive integer that determines the sharpness of the peaks or notches. The greater the value of the shelving filter order, the steeper the slope of the peak or notch. This results in a filter of order `L*Nsh`.
`Q`	Peak or notch quality factor. `Q` represents the ratio of the lowest center frequency peak or notch (not including DC) to the bandwidth calculated at the –3 dB point.

`d=fdesign(combtype,`*`specstring`*`,specvalue1,specvalue2,...)` creates an IIR comb filter specification object of type `combtype` and sets its `Specification` property to the values in `specvalue1,specvalue2,...`

`d=fdesign.comb(...,Fs)` creates an IIR comb filter specification object using the sampling frequency, `Fs`, of the signal to be filtered. The function assumes that `Fs` is in Hertz and must be specified as a scalar trailing all other provided values.

Examples

Construct a Peaking and Notching Combing Filter

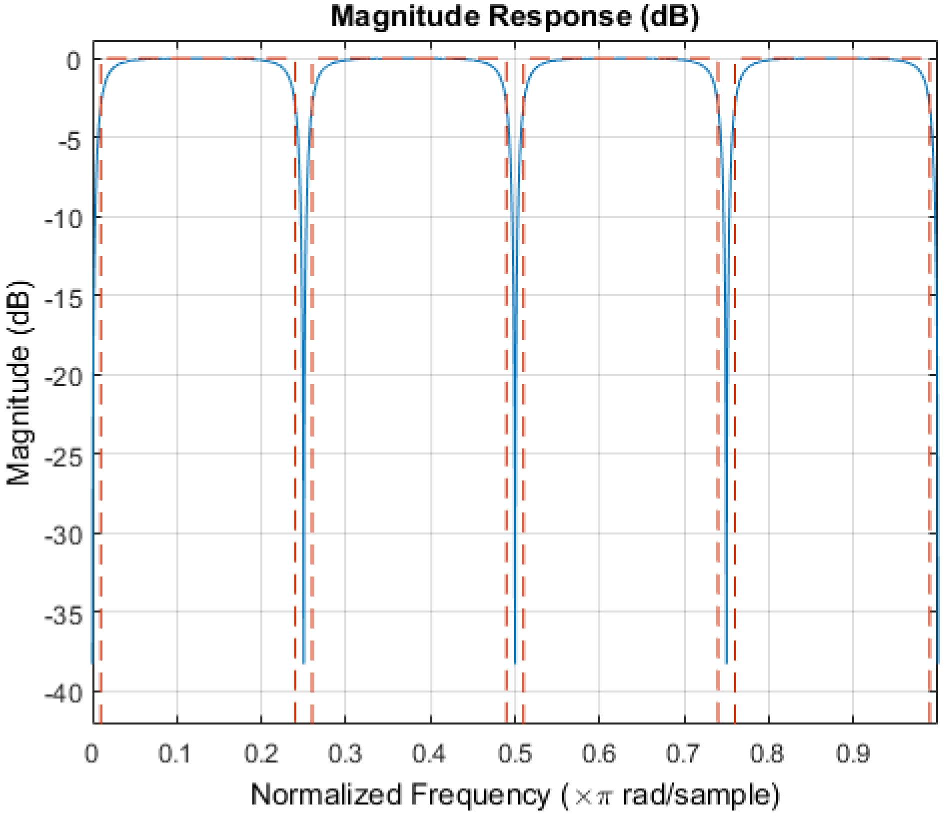
These examples demonstrate how to create IIR comb filter specification objects.

First, create a default specification object.

```
d = fdesign.comb;
```

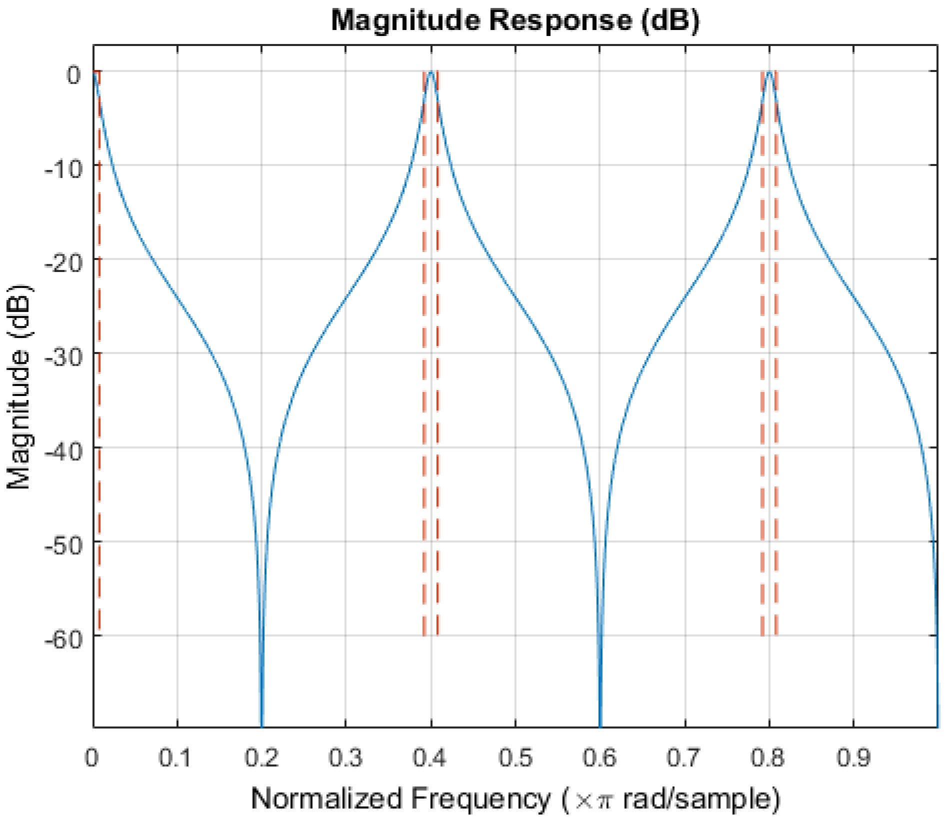
In the next example, create a notching filter of order 8 with a bandwidth of 0.02 (normalized frequency) referenced to the -3 dB point.

```
d  = fdesign.comb('notch','N,BW',8,0.02);
Hd = design(d,'SystemObject',true);
fvtool(Hd);
```



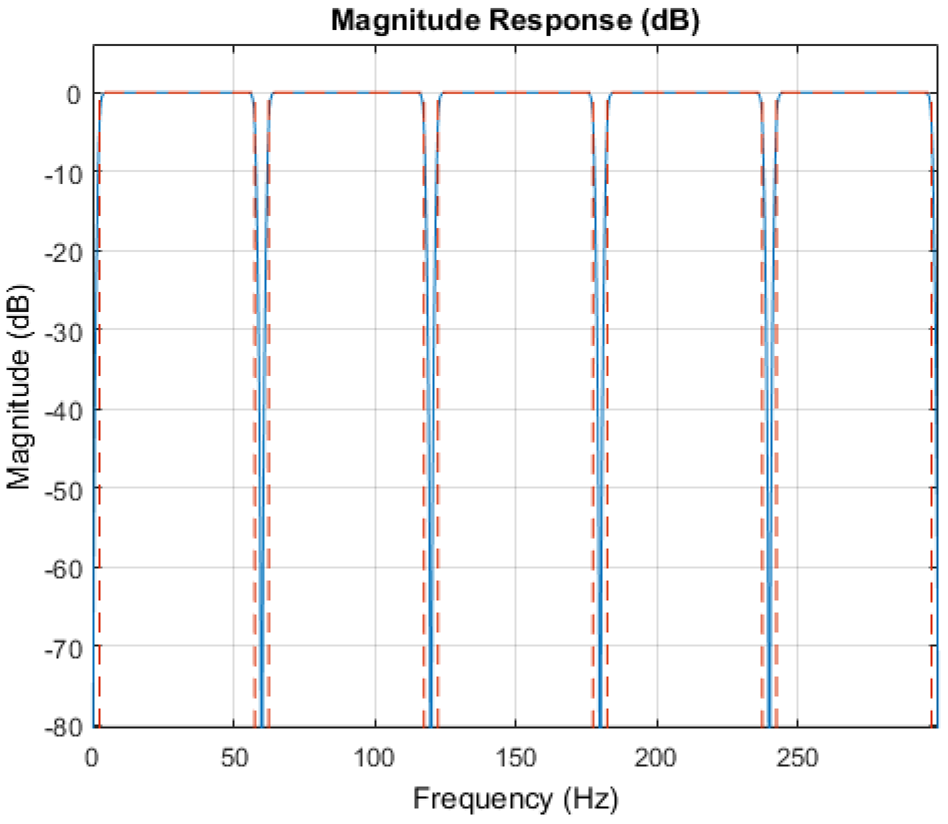
Next, create a peaking comb filter with 5 peaks and a quality factor of 25.

```
d = fdesign.comb('peak','N,Q',5,25);
Hd = design(d,'SystemObject',true);
fvtool(Hd);
```



In the next example, create a notching filter to remove interference at 60 Hz and its harmonics. The following code creates a filter with 10 notches and a notch bandwidth of 5 Hz referenced to the -4dB level. The filter has a shelving filter order of 4 and a sampling frequency of 600 Hz. Because the notches are equidistantly spaced in the interval [-300, 300] Hz, they occur at multiples of 60 Hz.

```
d = fdesign.comb('notch','L,BW,GBW,Nsh',10,5,-4,4,600);
Hd=design(d,'SystemObject',true);
fvtool(Hd);
```



Introduced in R2011a

fdesign.decimator

Decimator filter specification object

Syntax

```
D = fdesign.decimator(M)
D = fdesign.decimator(M, RESPONSE)
D = fdesign.decimator(M, CICRESPONSE, D)
D = fdesign.decimator(M, RESPONSE, SPEC)
D = fdesign.decimator(...,SPEC,specvalue1,specvalue2,...)
D = fdesign.decimator(...,Fs)
D = fdesign.decimator(...,MAGUNITS)
```

Description

`D = fdesign.decimator(M)` constructs a decimator filter specification object `D` with the `DecimationFactor` property equal to the positive integer `M` and the `Response` property set to `'Nyquist'`. The default values for the transition width and stopband attenuation in the Nyquist design are 0.1π radians/sample and 80 dB. If `M` is unspecified, `M` defaults to 2.

`D = fdesign.decimator(M, RESPONSE)` constructs a decimator specification object with the decimation factor `M` and the `'Response'` property.

`D = fdesign.decimator(M, CICRESPONSE, D)` constructs a CIC or CIC compensator decimator specification object with the decimation factor, `M`, `'Response'` property equal to `'CIC'` or `'CICCOMP'`, and `D` equal to the differential delay. The differential delay, `D`, must precede any specification option.

Because you are designing multirate filters, the specification options available are not the same as the specifications for designing single-rate filters. The decimation factor `M` is not included in the specification options. Different filter responses support different specifications. The following table lists the supported response types and specification options. The options are not case sensitive.

Design Method	Valid Specification Options
`'Arbitrary Magnitude'`	See `fdesign.arbmag` for a description of the specification entries. • `'N,F,A'` (default option) • `'N,B,F,A'`
`'Arbitrary Magnitude and Phase'`	See `fdesign.arbmagnphase` for a description of the specification entries. • `'N,F,H'` (default option) • `'N,B,F,H'`
`'Bandpass'`	See `fdesign.bandpass` for a description of the specification entries. • `'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'` (default option) • `'N,Fc1,Fc2'` • `'N,Fst1,Fp1,Fp2,Fst2'`
`'Bandstop'`	See `fdesign.bandstop` for a description of the specification entries. • `'N,Fc1,Fc2'` • `'N,Fp1,Fst1,Fst2,Fp2'` • `'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'` (default option)

`'CIC'`	`'Fp,Ast'` — Only valid specification. `Fp` is the passband frequency and `Ast` is the stopband attenuation in decibels.
	To specify a CIC decimator, include the differential delay after `'CIC'` and before the filter specification option: `'Fp,Ast'`. For example:
	`d =`
	`fdesign.decimator(2,'cic',4,'Fp,Ast',0.4,40);`

Design Method	Valid Specification Options
'CIC Compensator'	See `fdesign.ciccomp` for a description of the specification entries. • 'Fp,Fst,Ap,Ast' (default option) • 'N,Fc,Ap,Ast' • 'N,Fp,Ap,Ast' • 'N,Fp,Fst' • 'N,Fst,Ap,Ast' To specify a CIC compensator decimator, include the differential delay after 'CICCOMP' and before the filter specification. For example: `d = fdesign.decimator(2,'ciccomp',4);`
'Differentiator'	'N' — filter order
'Gaussian'	'Nsym,BT — Nsym is the filter order in symbols and BT is the bandwidth-symbol time product. The specification must be preceded by an integer-valued SamplesPerSymbol.
'Halfband	See `fdesign.halfband` for a description of the specification entries. • 'TW,Ast' (default option) • 'N,TW' • ‘N’ • 'N,Ast' If you use the quasi-linear IIR design method, `iirlinphase`, with a halfband specification, the interpolation factor must be 2.

Design Method	Valid Specification Options
'Highpass'	See fdesign.highpass for a description of the specification entries. • 'Fst,Fp,Ast,Ap' (default option) • 'N,F3db' • 'N,Fc' • 'N,Fc,Ast,Ap' • 'N,Fp,Ast,Ap' • 'N,Fst,Ast,Ap' • 'N,Fst,Fp' • 'N,Fst,Ast,Ap' • 'N,Fst,Fp,Ast'
'Hilbert'	See fdesign.hilbert for a description of the specification entries. • 'N,TW' (default option) • 'TW,Ap'
'Inverse-sinc Lowpass'	See fdesign.isinclp for a description of the specification entries. • 'Fp,Fst,Ap,Ast' (default option) • 'N,Fc,Ap,Ast' • 'N,Fp,Fst' • 'N,Fst,Ap,Ast'
'Inverse-sinc Highpass'	See fdesign.isinchp for a description of the specification entries. • 'Fst,Fp,Ast,Ap' (default option) • 'N,Fc,Ast,Ap' • 'N,Fst,Fp' • 'N,Fst,Ast,Ap'

Design Method	Valid Specification Options
'Lowpass'	See `fdesign.lowpass` for a description of the specification entries. • 'Fp,Fst,Ap,Ast' (default option) • 'N,F3dB' • 'N,Fc' • 'N,Fc,Ap,Ast' • 'N,Fp,Ap,Ast' • 'N,Fp,Fst' • 'N,Fp,Fst,Ap' • 'N,Fp,Fst,Ast' • 'N,Fst,Ap,Ast'
'Nyquist'	See `fdesign.nyquist` for a description of the specification entries. For all Nyquist specifications, you must specify the *L*th band. This typically corresponds to the `DecimationFactor`. • 'TW,Ast' (default option) • 'N' • 'N,Ast' • 'N,Ast'

`D = fdesign.decimator(M, RESPONSE, SPEC)` constructs object `D` and sets the `Specification` property to `SPEC` for the response type, `RESPONSE`. Entries in the `SPEC` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `SPEC` depend on the `RESPONSE` type.

Because you are designing multirate filters, the specification options available are not the same as the specifications for designing single-rate filters with such response types as `fdesign.lowpass`. The options are not case sensitive.

The decimation factor `M` is not in the specification options.

`D = fdesign.decimator(...,SPEC,specvalue1,specvalue2,...)` constructs an object `D` and sets its specifications at construction time.

`D = fdesign.decimator(...,Fs)` provides the sampling frequency of the signal to be filtered. `Fs` must be specified as a scalar trailing the other numerical values provided. `Fs` is assumed to be in Hz as are all other frequency values provided.

`D = fdesign.decimator(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units.
- `'dB'` — specify the magnitude in dB (decibels).
- `'squared'` — specify the magnitude in power units.

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

Construct a Decimator Using `fdesign` Object

These examples show how to construct decimating filter specification objects.

First, create a default specifications object without using input arguments except for the decimation factor m.

```
d = fdesign.decimator(2,'Nyquist',2,0.1,80) % Set tw=0.1, and ast=80.
```

```
d = 

  decimator with properties:

          MultirateType: 'Decimator'
               Response: 'Nyquist'
       DecimationFactor: 2
          Specification: 'TW,Ast'
            Description: {2×1 cell}
                   Band: 2
    NormalizedFrequency: 1
        TransitionWidth: 0.1000
                  Astop: 80
```

Now create an object by passing a specification type option 'fst1,fp1,fp2,fst2,ast1,ap,ast2' and a design - the resulting object uses default values for the filter specifications. You must provide the design input argument, bandpass in this example, when you include a specification.

```
d = fdesign.decimator(8,'bandpass',...
'fst1,fp1,fp2,fst2,ast1,ap,ast2');
```

Create another decimating filter specification object, passing the specification values to the object rather than accepting the default values for fp,fst,ap,ast.

```
d = fdesign.decimator(3,'lowpass',.45,0.55,.1,60);
```

Now pass the filter specifications that correspond to the specifications - n,fc,ap,ast.

```
d = fdesign.decimator(3,'ciccomp',1,2,'n,fc,ap,ast',...
20,0.45,.05,50);
```

Now design a decimator using the equiripple design method.

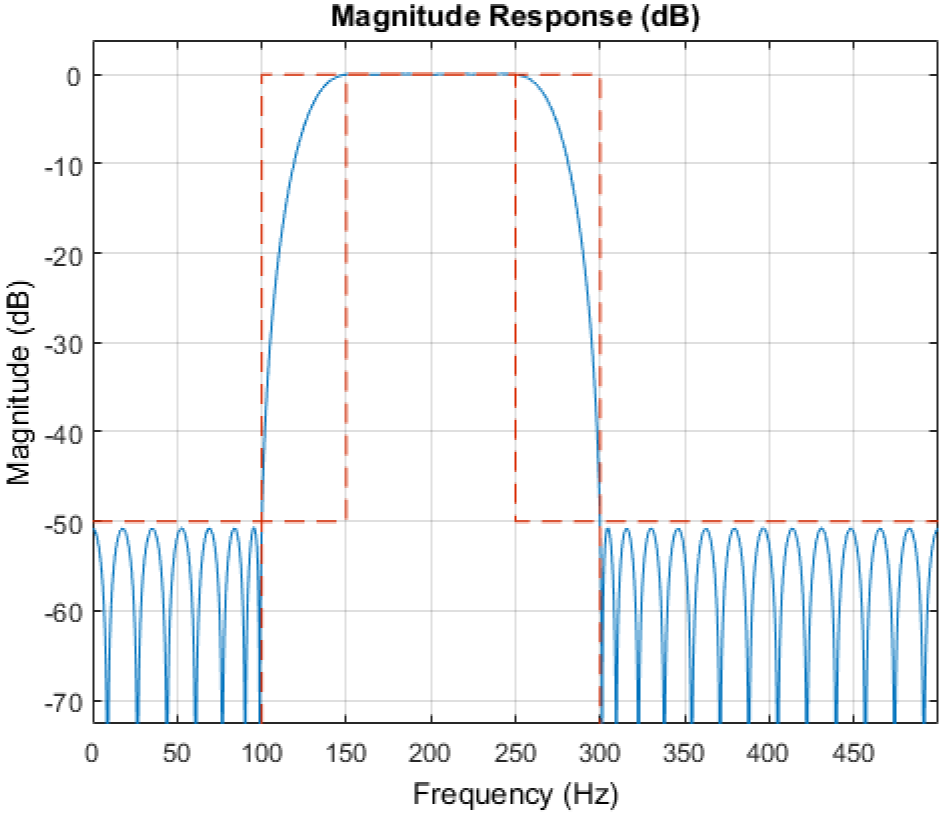
```
equiDecimator = design(d,'equiripple','SystemObject',true);
```

Pass a new specification type for the filter, specifying the filter order. Note that the inputs must include the differential delay dd with the CIC input argument to design a CIC specification object.

```
m = 5;
dd = 2;
d = fdesign.decimator(m,'cic',dd,'fp,ast',0.55,55);
```

In this example, you specify a sampling frequency as the last input argument. Here is it 1000 Hz. Design an equiripple filter and plot the magnitude response:

```
d = fdesign.decimator(8,'bandpass','fst1,fp1,fp2,fst2,ast1,ap,ast2',...
100,150,250,300,50,.05,50,1000);
fvtool(design(d,'equiripple','SystemObject',true))
```



See Also

`fdesign` | `fdesign.arbmag` | `fdesign.arbmagnphase` | `fdesign.interpolator` | `fdesign.rsrc`

Introduced in R2011a

fdesign.differentiator

Differentiator filter specification object

Syntax

```
D = fdesign.differentiator
D = fdesign.differentiator(SPEC)
D = fdesign.differentiator(SPEC,specvalue1,specvalue2, ...)
D = fdesign.differentiator(specvalue1)
D = fdesign.differentiator(...,Fs)
D = fdesign.differentiator(...,MAGUNITS)
```

Description

`D = fdesign.differentiator` constructs a default differentiator filter designer `D` with the filter order set to 31.

`D = fdesign.differentiator(SPEC)` initializes the filter designer `Specification` property to `SPEC`. You provide one of the following filter entries as input to replace `SPEC`. These entries are not case sensitive.

Note: Specifications marked with an asterisk require the DSP System Toolbox software.

- `'N'` — Full band differentiator (default)
- `'N,Fp,Fst'` — Partial band differentiator
- `'N,Fp,Fst,Ap'` — Partial band differentiator *
- `'N,Fp,Fst,Ast'` — Partial band differentiator *
- `'Ap'` — Minimum order full band differentiator *
- `'Fp,Fst,Ap,Ast'` — Minimum order partial band differentiator *

The filter specifications are defined as follows:

- `Ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.

- `Ast` — attenuation in the stop band in decibels (the default units). Also called Astop.
- `Fp` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass.
- `Fst` — frequency at the end of the stop band. Specified in normalized frequency units. Also called Fstop.
- `N` — filter order.

By default, `fdesign.differentiator` assumes that all frequency specifications are provided in normalized frequency units. Also, decibels is the default for all magnitude specifications.

Use `designopts` to determine the design options for a given design method. Enter `help(D,METHOD)` at the MATLAB command line to obtain detailed help on the design options for a given design method, `METHOD`.

`D = fdesign.differentiator(SPEC,specvalue1,specvalue2, ...)` initializes the filter designer specifications in `SPEC` with `specvalue1`, `specvalue2`, and so on. To get a description of the specifications `specvalue1`, `specvalue2`, and more, enter

```
get(d,'description')
```

at the Command prompt.

`D = fdesign.differentiator(specvalue1)` assumes the default specification `N`, setting the filter order to the value you provide.

`D = fdesign.differentiator(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`D = fdesign.differentiator(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units
- `'dB'` — specify the magnitude in dB (decibels)
- `'squared'` — specify the magnitude in power units

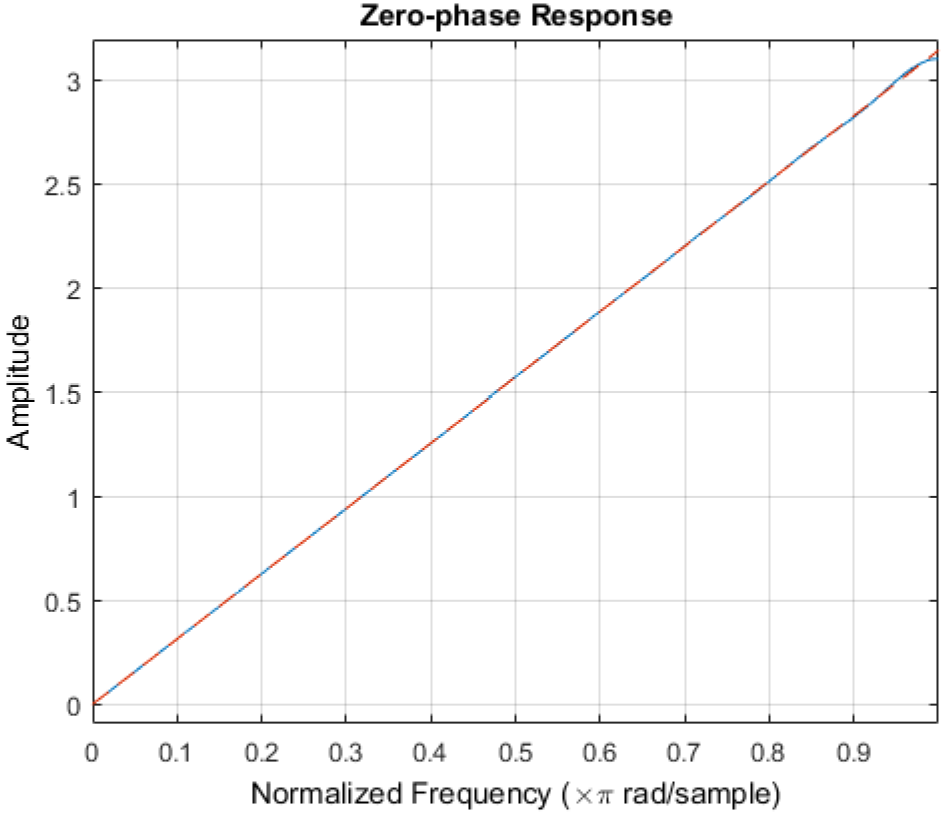
When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

FIR Differentiator Filters

Design a 33rd-order FIR differentiator using least squares. Plot the zero-phase response of the filter.

```
d = fdesign.differentiator(33);
hd = design(d,'firls','SystemObject',true);

zerophase(hd)
```

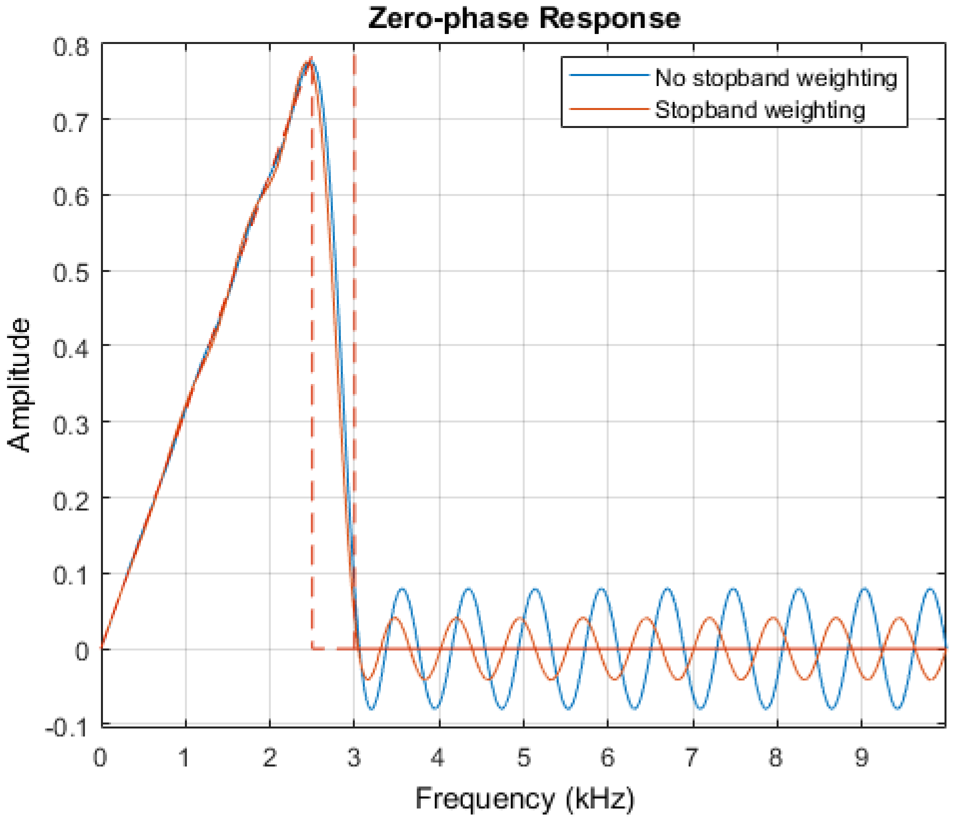


Design a 54th-order narrowband equiripple differentiator. Differentiate the lowest 25% of the frequencies in the Nyquist range and filter the higher frequencies. Specify a sample rate of 20 kHz, a passband frequency of 2.5 kHz, and a stopband frequency of 3 kHz.

```
Fs = 20000;

d = fdesign.differentiator('N,Fp,Fst',54,2500,3000,Fs);
Hd = design(d,'equiripple','SystemObject',true);
```

Redesign the filter, but this time weight the stopband to increase the attenuation.

```
Hd1 = design(d,'equiripple','Wstop',4,'SystemObject',true);

hfvt = fvtool(Hd,Hd1,'MagnitudeDisplay','zero-phase', ...
    'FrequencyRange','[0, Fs/2)');
legend(hfvt,'No stopband weighting','Stopband weighting');
```



See Also

See Also

`design` | `fdesign`

Introduced in R2009a

fdesign.fracdelay

Fractional delay filter specification object

Syntax

```
d = fdesign.fracdelay(delta)
d = fdesign.fracdelay(delta,'N')
d = fdesign.fracdelay(delta,'N',n)
d = fdesign.fracdelay(delta,n)
d = fdesign.fracdelay(...,fs)
```

Description

`d = fdesign.fracdelay(delta)` constructs a default fractional delay filter designer `d` with the filter order set to 3 and the delay value set to `delta`. The fractional delay `delta` must be between 0 and 1 samples.

`d = fdesign.fracdelay(`*`delta,`*`'N')` initializes the filter designer specification to `N`, where `N` specifies the fractional delay filter order and defaults to filter order of 3.

Use `designmethods(d)` to get a list of the design methods available for a specification.

`d = fdesign.fracdelay(delta,'N',n)` initializes the filter designer to `N` and sets the filter order to `n`.

`d = fdesign.fracdelay(delta,n)` assumes the default specification `N`, filter order, and sets the filter order to the value you provide in input `n`.

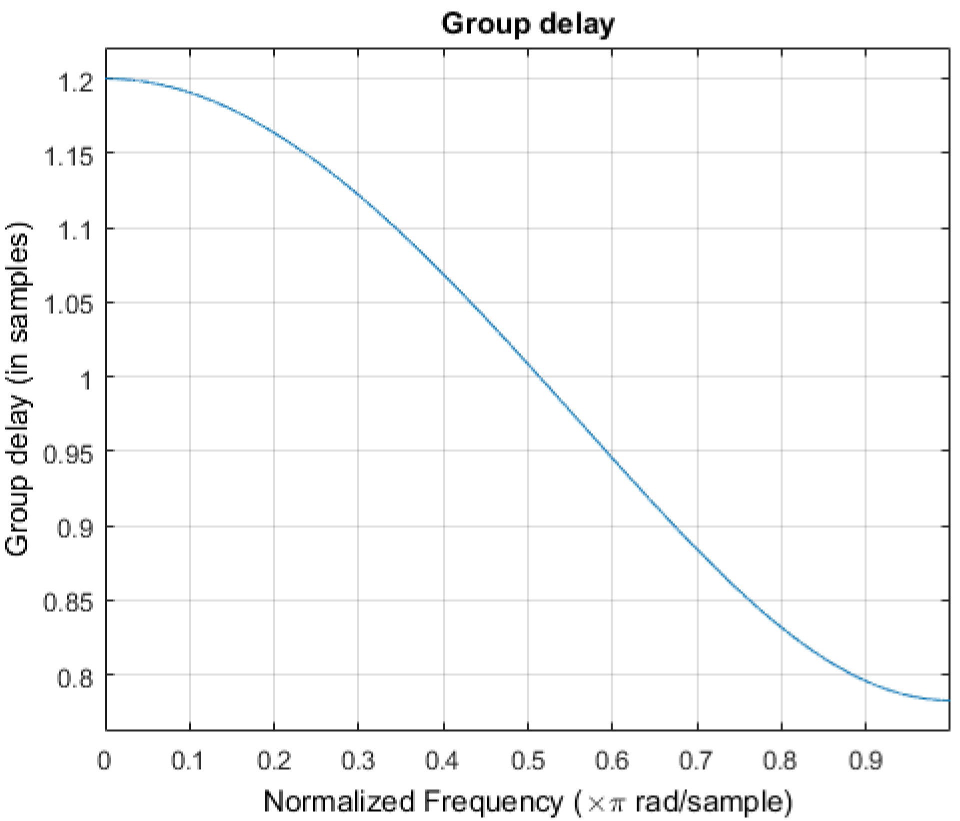
`d = fdesign.fracdelay(...,fs)` adds the argument `fs`, specified in units of Hertz (Hz) to define the sampling frequency. In this case, specify the fractional delay `delta` to be between 0 and 1/`fs`.

Examples

Create a Fractional Delay Filter Using `fdesign` Object

Design a second-order fractional delay filter of 0.2 samples using the Lagrange method. Implement the filter using a Farrow fractional delay (fd) structure.

```
d = fdesign.fracdelay(0.2,'N',2);
secondOrderFrac = design(d,'lagrange','filterstructure','farrowfd');
fvtool(secondOrderFrac,'analysis','grpdelay')
```



Design a cubic fractional delay filter with a sampling frequency of 8 kHz and fractional delay of 50 microseconds using the Lagrange method.

```
d = fdesign.fracdelay(50e-6,'N',3,8000);
cubicFrac = design(d, 'lagrange', 'FilterStructure', 'farrowfd');
```

See Also

design | designopts | fdesign | setspecs

Introduced in R2011a

fdesign.halfband

Halfband filter specification object

Syntax

```
d = fdesign.halfband
d = fdesign.halfband('type',type)
d = fdesign.halfband(spec)
d = fdesign.halfband(spec,specvalue1,specvalue2,...)
d = fdesign.halfband(specvalue1,specvalue2)
d = fdesign.halfband(...,fs)
d = fdesign.halfband(...,magunits)
```

Description

`d = fdesign.halfband` constructs a halfband filter specification object `d`, applying default values for the properties `tw` and `ast`.

Using `fdesign.halfband` along with `design` method generates a System object, if the `'SystemObject'` flag in the `design` method is set to `true`.

`d = fdesign.halfband('type',type)` initializes the filter designer 'Type' property with `type`. '`type`' must be either `lowpass` or `highpass` and is not case sensitive.

`d = fdesign.halfband(`*`spec`*`)` constructs object `d` and sets its '`Specification`' to *`spec`*. Entries in `spec` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for *`spec`* are shown below. These options are not case sensitive.

- `tw,ast` (default `spec`)
- `n,tw`
- `n`
- `n,ast`

where,

- `ast` — attenuation in the stop band in decibels (the default units).
- `n` — filter order.
- `tw` — width of the transition region between the pass and stop bands. Specified in normalized frequency units.

By default, all frequency specifications are assumed to be in normalized frequency units. Moreover, all magnitude specifications are assumed to be in dB. Different specification types may have different design methods available.

The filter design methods that apply to a halfband filter specification object change depending on the `Specification` choice. Use `designmethods` to determine which design method applies to an object and its specification choice. Different filter design methods also have options that you can specify. Use `designopts` with the design method to see the available options. For example:

```
f=fdesign.halfband('N,TW');
designmethods(f)
```

`d = fdesign.halfband(spec,specvalue1,specvalue2,...)` constructs an object `d` and sets its specifications at construction time.

`d = fdesign.halfband(specvalue1,specvalue2)` constructs an object `d` assuming the default `Specification` property `tw,ast`, using the values you provide for the input arguments `specvalue1` and `specvalue2` for `tw` and `ast`.

`d = fdesign.halfband(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.halfband(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of

- `linear` — specify the magnitude in linear units
- `dB` — specify the magnitude in dB (decibels)
- `squared` — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

Create Halfband Filters Using `fdesign` Object

Create a default halfband filter specifications object.

```
d=fdesign.halfband;
```

Create another halfband filter object, passing the specification values to the object rather than accepting the default values for n and ast.

```
d2 = fdesign.halfband('n,ast', 42, 80);
```

For another example, pass the filter values that correspond to the default Specification - n,ast.

```
d3 = fdesign.halfband(.01, 80);
```

This example designs an equiripple FIR filter, starting by passing a new specification type and specification values to fdesign.halfband.

```
hs = fdesign.halfband('n,ast',80,70);
hd =design(hs,'equiripple','SystemObject',true);
```

In this example, pass the specifications for the filter, and then design a least-squares FIR filter from the object, using firls as the design method.

```
hs = fdesign.halfband('n,tw', 42, .04);
hd2 = design(hs,'firls','SystemObject',true);
```

Create two equiripple halfband filters with and without a nonnegative zero phase response:

```
f=fdesign.halfband('N,TW',12,0.2);
```

Equiripple halfband filter with nonnegative zero phase response

```
Hd1 = design(f,'equiripple','ZeroPhase',true,'SystemObject',true);
```

Equiripple halfband filter with zero phase false 'zerophase',false is the default

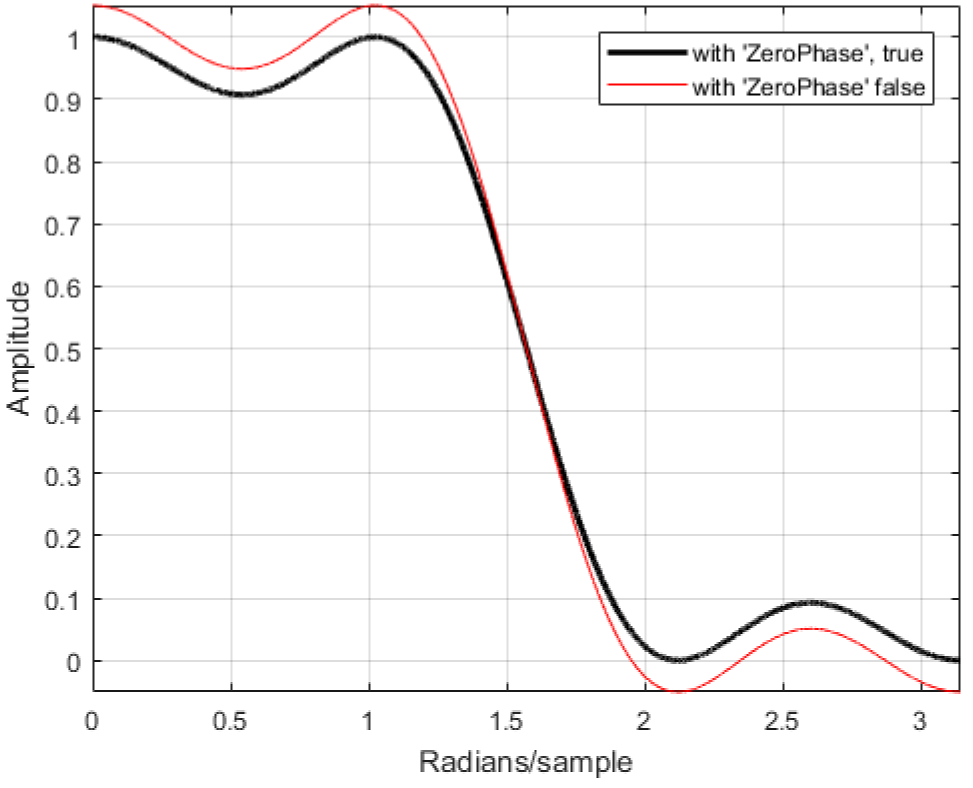
```
Hd2=design(f,'equiripple','ZeroPhase',false,'SystemObject',true);
```

Obtain real-valued amplitudes (not magnitudes)

```
[Hr_zerophase,W]=zerophase(Hd1);
[Hr,W]=zerophase(Hd2);
```

Plot and compare response

```
plot(W,Hr_zerophase,'k','linewidth',2);
xlabel('Radians/sample'); ylabel('Amplitude');
hold on;
plot(W,Hr,'r');
axis tight; grid on;
legend('with ''ZeroPhase'', true','with ''ZeroPhase'' false');
```



Note that the amplitude of the zero phase response (black line) is nonnegative for all frequencies. The 'ZeroPhase' option is valid only for equiripple halfband designs with the

'N,TW' specification. You cannot specify 'MinPhase' and 'ZeroPhase' to be simultaneously 'true'.

See Also

`fdesign` | `fdesign.decimator` | `design` | `fdesign.interpolator` | `fdesign.nyquist` | `setspecs` | `zerophase`

Introduced in R2011a

fdesign.highpass

Highpass filter specification object

Syntax

```
D = fdesign.highpass
D = fdesign.highpass(SPEC)
D = fdesign.highpass(SPEC,specvalue1,specvalue2,...)
D = fdesign.highpass(specvalue1,specvalue2,specvalue3,
specvalue4)
D = fdesign.highpass(...,Fs)
D = fdesign.highpass(...,MAGUNITS)
```

Description

`D = fdesign.highpass` constructs a highpass filter specification object `D`, applying default values for the specification, `'Fst,Fp,Ast,Ap'`.

`D = fdesign.highpass(SPEC)` constructs object `D` and sets the `Specification` property to `SPEC`. Entries in the `SPEC` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `SPEC` are shown below. These entries are not case sensitive.

Note: Specification entries marked with an asterisk require the DSP System Toolbox software.

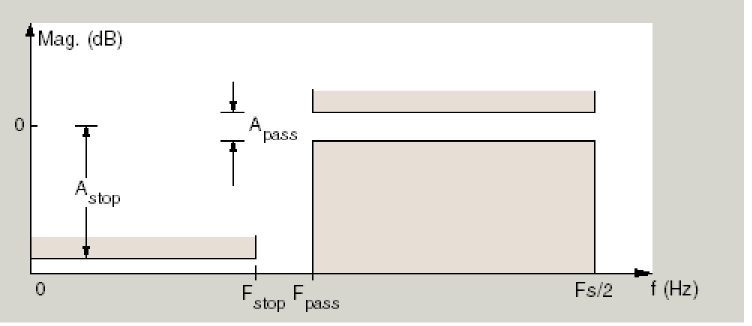
- `'Fst,Fp,Ast,Ap'` (default `spec`)
- `'N,F3db'`
- `'N,F3db,Ap'` *
- `'N,F3db,Ast'` *
- `'N,F3db,Ast,Ap'` *
- `'N,F3db,Fp` *

- 'N,Fc'
- 'N,Fc,Ast,Ap'
- 'N,Fp,Ap'
- 'N,Fp,Ast,Ap'
- 'N,Fst,Ast'
- 'N,Fst,Ast,Ap'
- 'N,Fst,F3db' *
- 'N,Fst,Fp'
- 'N,Fst,Fp,Ap' *
- 'N,Fst,Fp,Ast' *
- 'Nb,Na,Fst,Fp' *

The filter specifications are defined as follows:

- Ap — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.
- Ast — attenuation in the stop band in decibels (the default units). Also called Astop.
- F3db — cutoff frequency for the point 3 dB point below the passband value. Specified in normalized frequency units.
- Fc — cutoff frequency for the point 6 dB point below the passband value. Specified in normalized frequency units.
- Fp — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass.
- Fst — frequency at the end of the stop band. Specified in normalized frequency units. Also called Fstop.
- N — filter order.
- Na and Nb are the order of the denominator and numerator.

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like Fst and Fp are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a highpass filter specification object change depending on the Specification. Use designmethods to determine which design method applies to an object and its specification.

Use designopts to determine which design options are valid for a given design method. For detailed information on design options for a given design method, METHOD, enter help(D,METHOD) at the MATLAB command line.

D = fdesign.highpass(SPEC,specvalue1,specvalue2,...) constructs an object d and sets its specification values at construction time.

D = fdesign.highpass(specvalue1,specvalue2,specvalue3, specvalue4) constructs an object D with the default Specification property and the values you enter for specvalue1,specvalue2,....

D = fdesign.highpass(...,Fs) provides the sampling frequency for the filter specification object. Fs is in Hz and must be specified as a scalar trailing the other numerical values provided. If you specify a sampling frequency, all other frequency specifications are in Hz.

D = fdesign.highpass(...,MAGUNITS) specifies the units for any magnitude specification you provide in the input arguments. MAGUNITS can be one of

- 'linear' — specify the magnitude in linear units
- 'dB' — specify the magnitude in dB (decibels)
- 'squared' — specify the magnitude in power units

When you omit the MAGUNITS argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

Highpass Filtering of Sinusoids

Highpass filter a discrete-time signal consisting of two sine waves.

Create a highpass filter specification object. Specify the passband frequency to be 0.25π rad/sample and the stopband frequency to be 0.15π rad/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB.

```
d = fdesign.highpass('Fst,Fp,Ast,Ap',0.15,0.25,60,1);
```

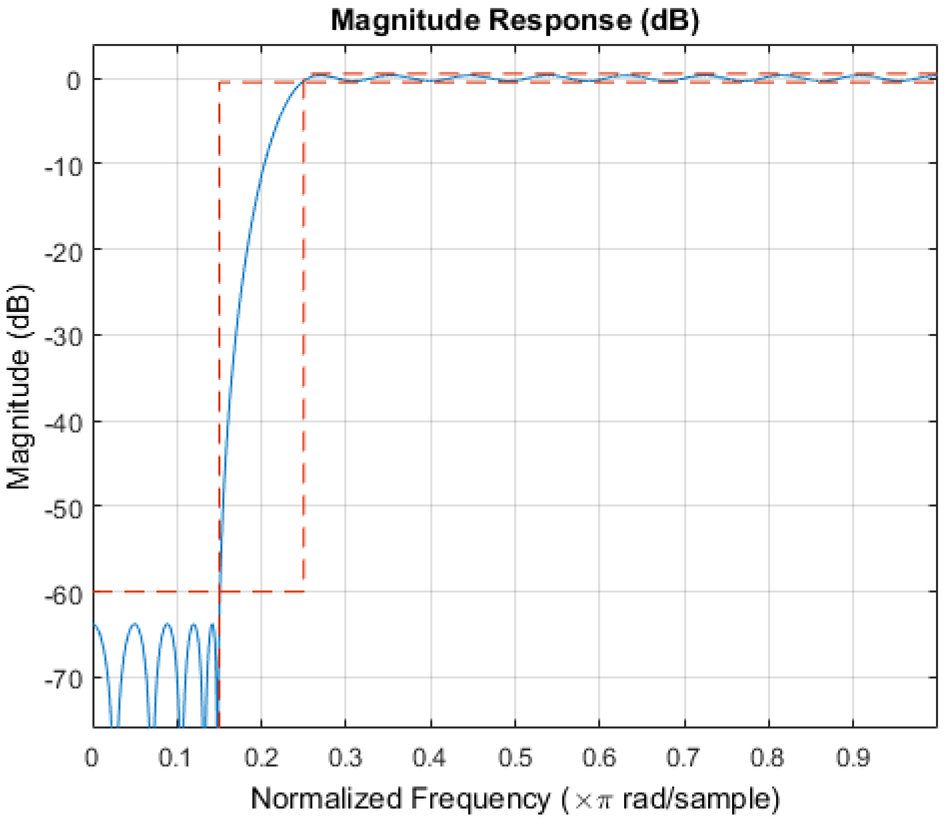
Query the valid design methods for your filter specification object.

```
designmethods(d)
```

```
Design Methods for class fdesign.highpass (Fst,Fp,Ast,Ap):


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
```

Create an FIR equiripple filter and view the filter magnitude response with FVTool.

```
Hd = design(d,'equiripple');
fvtool(Hd)
```



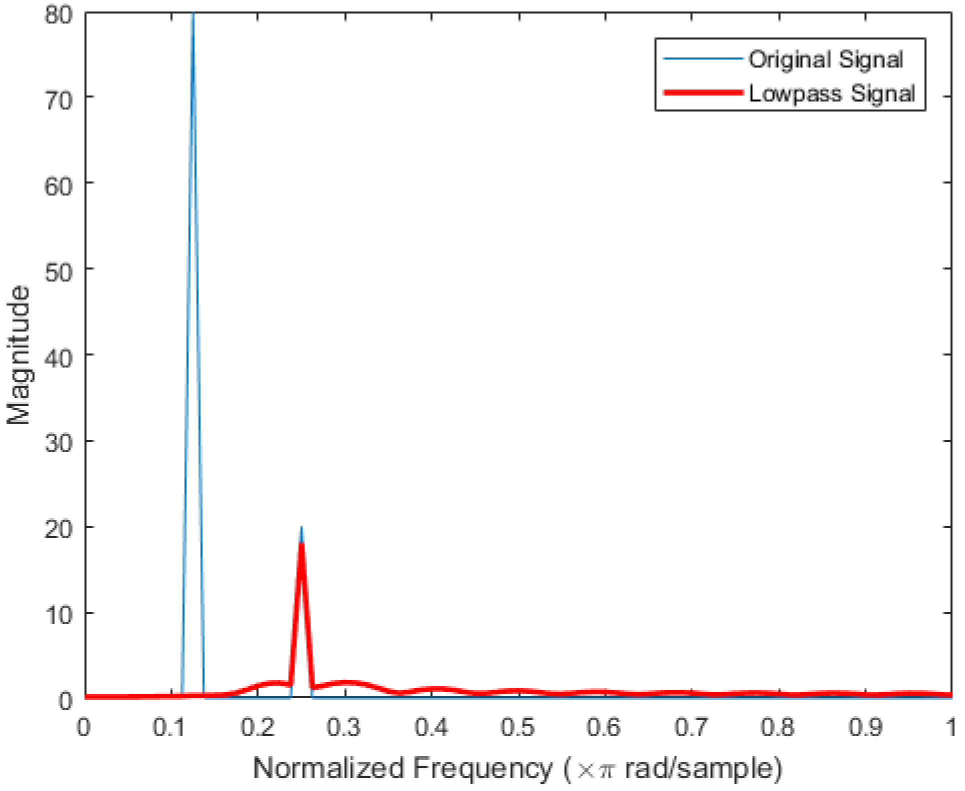
Create a signal consisting of the sum of two discrete-time sinusoids with frequencies of π/8 and π/4 rad/sample and amplitudes of 1 and 0.25 respectively. Filter the discrete-time signal with the FIR equiripple filter object.

```
n = 0:159;
x = cos(pi/8*n)+0.25*sin(pi/4*n);
y = filter(Hd,x);
```

Plot the original and filtered signals in the frequency domain.

```
freq = 0:(2*pi)/160:pi;
xdft = fft(x);
ydft = fft(y);
```

```
plot(freq/pi,abs(xdft(1:length(x)/2+1)))
hold on
plot(freq/pi,abs(ydft(1:length(y)/2+1)),'r','linewidth',2)
hold off
legend('Original Signal','Lowpass Signal','Location','NorthEast')
ylabel('Magnitude')
xlabel('Normalized Frequency (\times\pi rad/sample)')
```



Window Method Highpass Design

Create a filter of order 10 with a 6-dB frequency of 9.6 kHz and a sample rate of 48 kHz. Look at the available design methods.

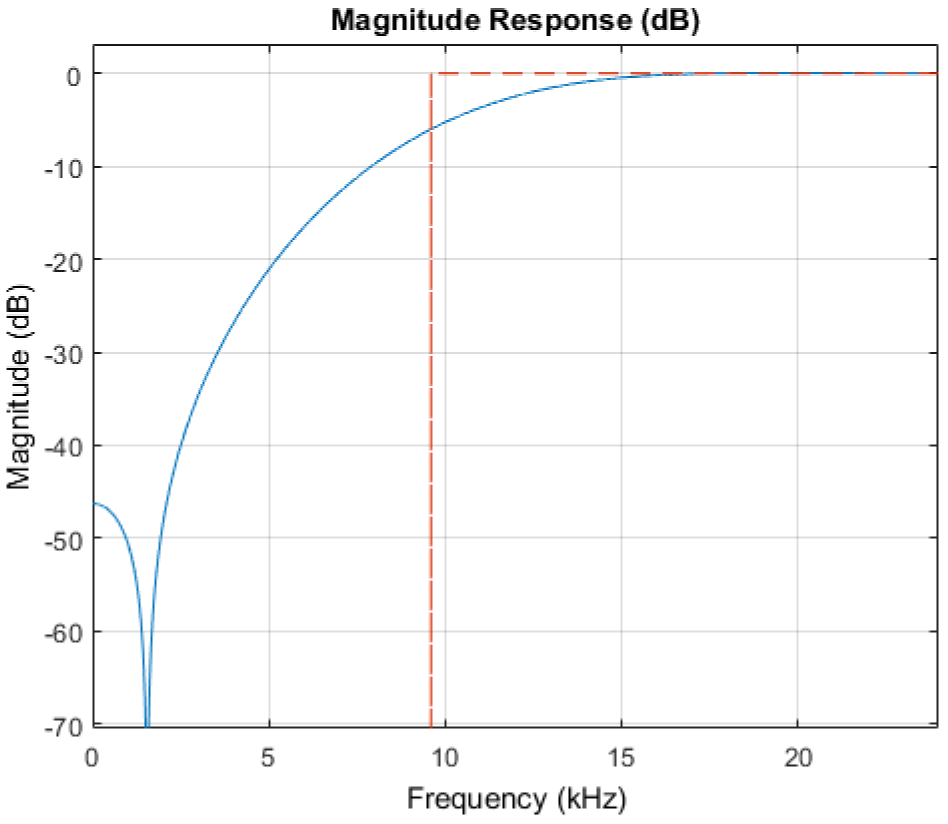
```
d=fdesign.highpass('N,Fc',10,9600,48000);
```

```
designmethods(d)
```

```
Design Methods for class fdesign.highpass (N,Fc):

window
```

The only available method is the FIR window method. Design the filter and display its magnitude response.

```
Hd = design(d);

fvtool(Hd)
```



Stopband Constraints

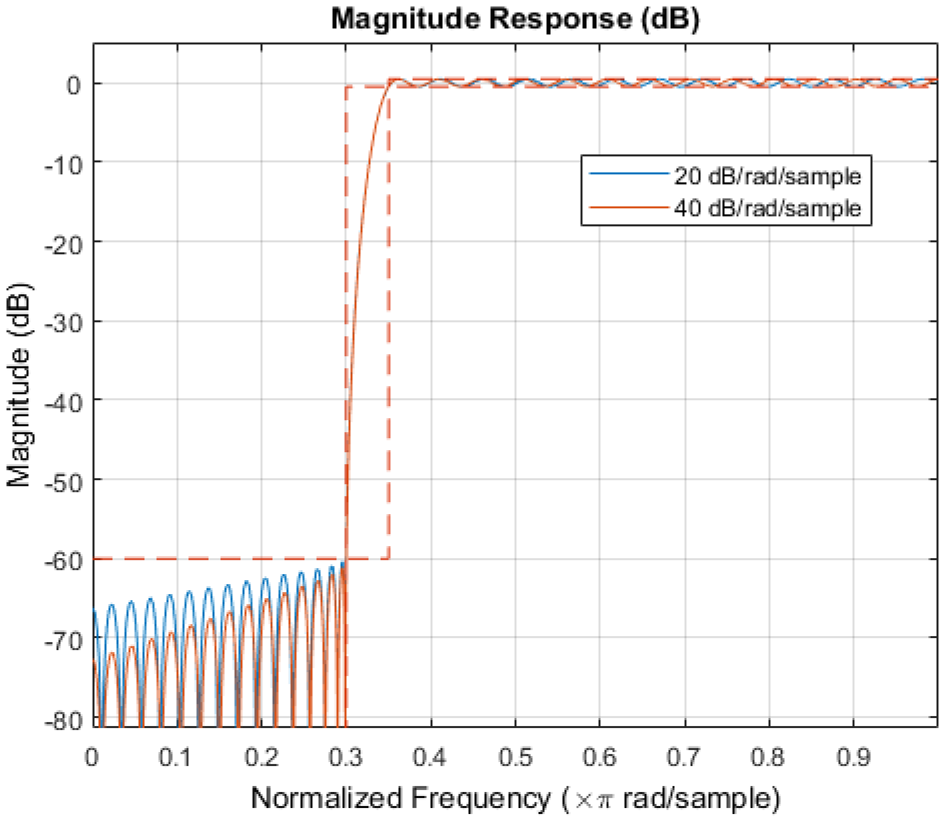
You can specify the shape of the stopband and the rate at which the stopband decays.

Create two FIR equiripple filters with different linear stopband slopes. Specify the passband frequency to be 0.3π rad/sample and the stopband frequency to be 0.35π rad/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB. Design one filter with a 20 dB/(rad/sample) stopband slope and another filter with a slope of 40 dB/(rad/sample).

```
D = fdesign.highpass('Fst,Fp,Ast,Ap',0.3,0.35,60,1);
Hd1 = design(D,'equiripple','StopBandShape','linear','StopBandDecay',20);
Hd2 = design(D,'equiripple','StopBandShape','linear','StopBandDecay',40);
```

Visualize the magnitude responses of the filters.

```
hfvt = fvtool([Hd1 Hd2]);
legend(hfvt,'20 dB/rad/sample','40 dB/rad/sample')
```



See Also

See Also

`design` | `designmethods` | `fdesign`

Introduced in R2009a

fdesign.hilbert

Hilbert filter specification object

Syntax

```
d = fdesign.hilbert
d = fdesign.hilbert(specvalue1,specvalue2)
d = fdesign.hilbert(spec)
d = fdesign.hilbert(spec,specvalue1,specvalue2)
d = fdesign.hilbert(...,Fs)
d = fdesign.hilbert(...,MAGUNITS)
```

Description

`d = fdesign.hilbert` constructs a default Hilbert filter designer `d` with `N`, the filter order, set to 30 and `TW`, the transition width set to 0.1π radians/sample.

`d = fdesign.hilbert(specvalue1,specvalue2)` constructs a Hilbert filter designer `d` assuming the default specification `'N,TW'`. You input `specvalue1` and `specvalue2` for `N` and `TW`.

`d = fdesign.hilbert(`*`spec`*`)` initializes the filter designer `Specification` property to `spec`. You provide one of the following as input to replace `spec`. The specification options are not case sensitive.

Note: Specifications marked with an asterisk require the DSP System Toolbox software.

- `'N,TW'` default specification option.
- `'TW,Ap'` *

The filter specifications are defined as follows:

- `Ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.

- `N` — filter order.
- `TW` — width of the transition region between the passband and the stopband.

By default, `fdesign.hilbert` assumes that all frequency specifications are provided in normalized frequency units. Also, decibels is the default for all magnitude specifications.

Different specifications may have different design methods available. Use `designmethods(d)` to get a list of the design methods available for a given specification.

`d = fdesign.hilbert(`*`spec`*`,specvalue1,specvalue2)` initializes the filter designer specifications in `spec` with `specvalue1`, `specvalue2`, and so on. To get a description of the specifications `specvalue1` and `specvalue2`, enter

```
get(d,'description')
```

at the Command prompt.

`d = fdesign.hilbert(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.hilbert(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units
- `'dB'` — specify the magnitude in dB (decibels)
- `'squared'` — specify the magnitude in power units

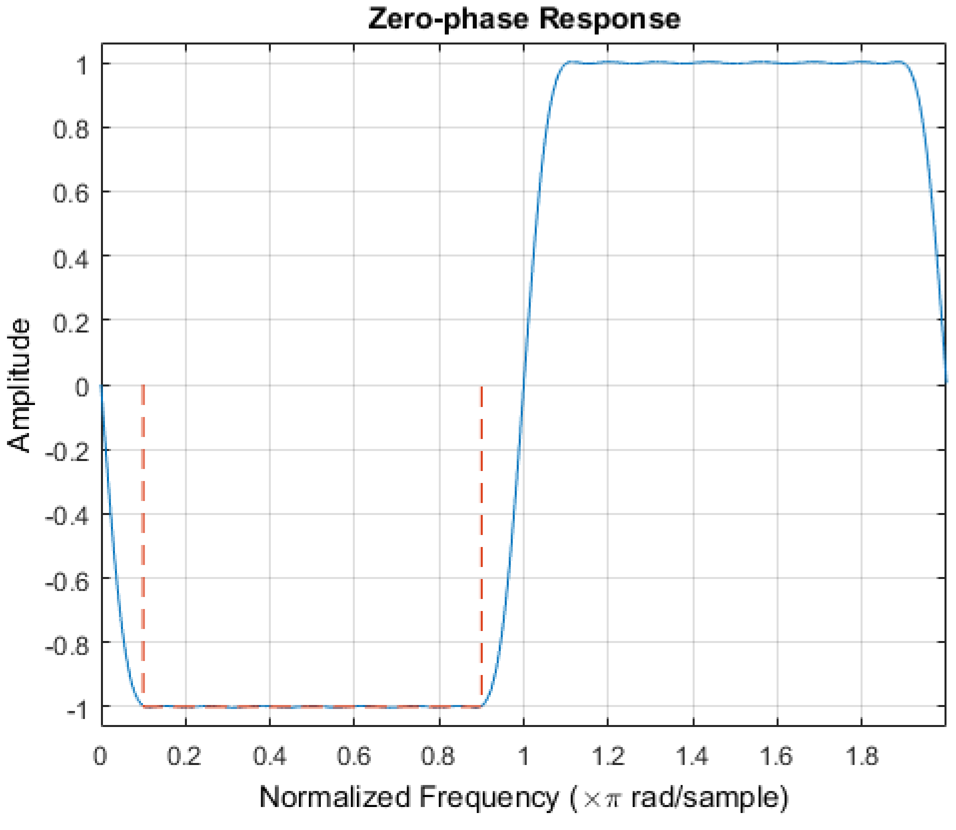
When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

Hilbert Transformers

Design a Hilbert transformer of order 30 with a transition width of 0.2π rad/sample. Use least-squares minimization to obtain an equiripple linear-phase FIR filter. Plot the zero-phase response in the interval [–π,π).

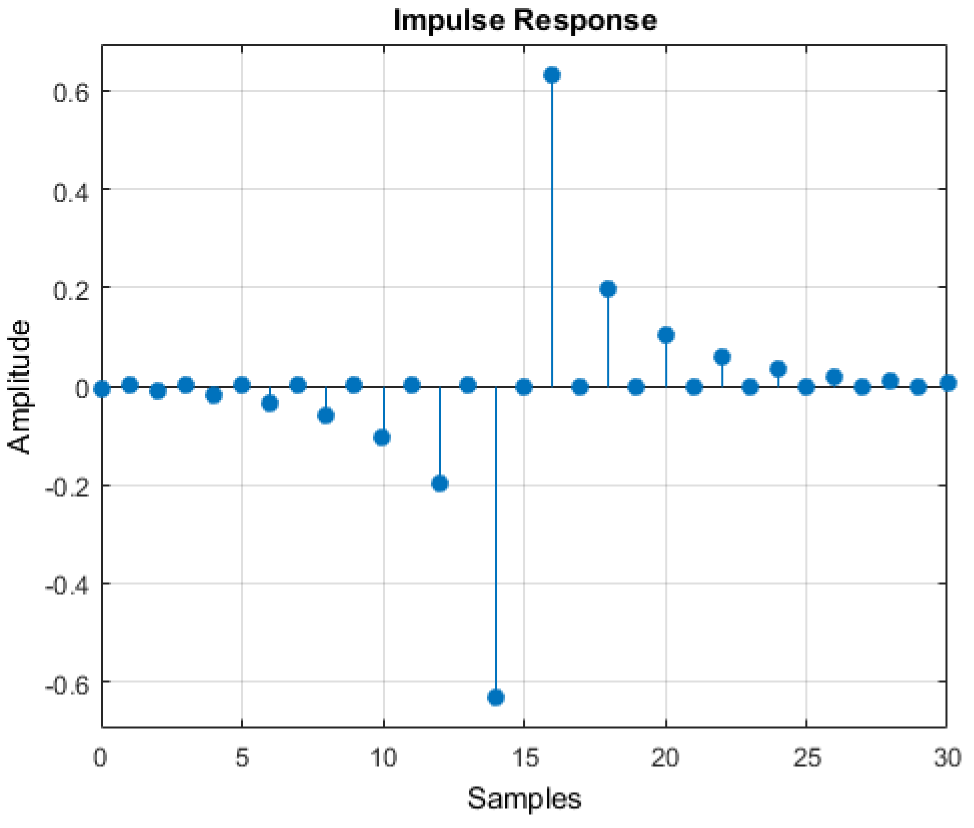
```
d = fdesign.hilbert('N,TW',30,0.2);
Hd = design(d,'equiripple','SystemObject',true);
zerophase(Hd,'whole')
```



The impulse response of this even-order type-3 filter is antisymmetric.

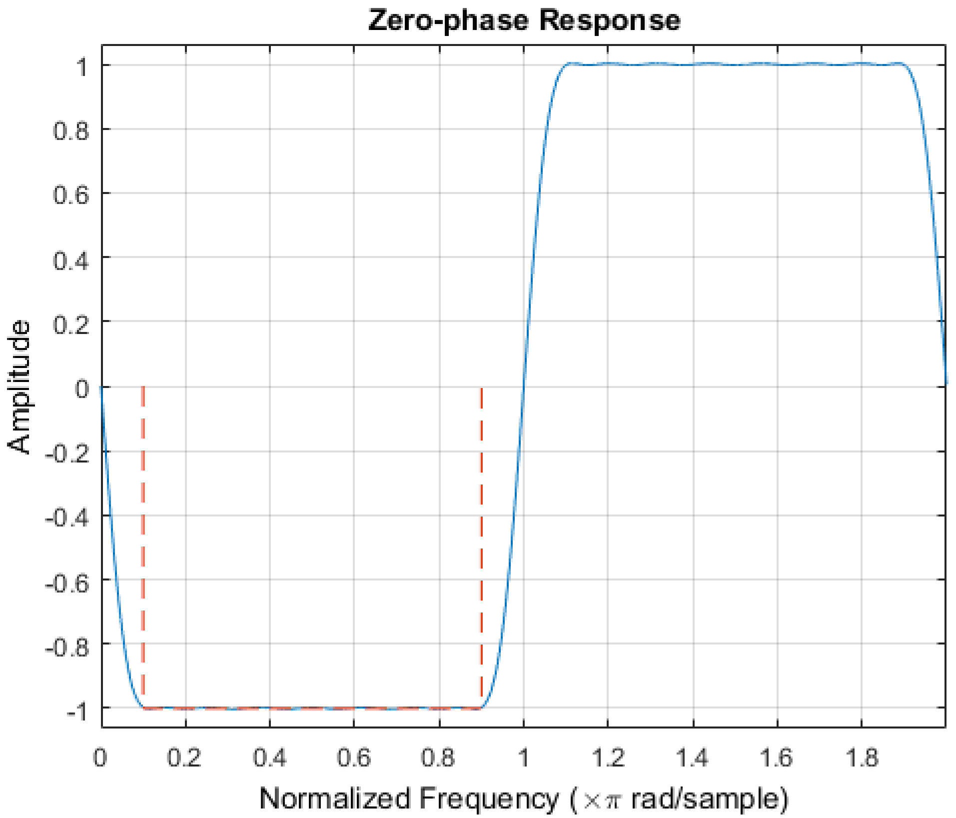
```
impz(Hd)
ftype = firtype(Hd)
```

```
ftype =

     3
```



Design a minimum-order Hilbert transformer that has a sample rate of 1 kHz. Specify the passband ripple to be 1 dB. Display the zero-phase response of the filter.

```
d = fdesign.hilbert('TW,Ap',1,0.1,1e3);
hd = design(d,'equiripple','SystemObject',true);
zerophase(Hd,'whole')
```



See Also

See Also

`design` | `fdesign` | `setspecs`

Introduced in R2009a

fdesign.interpolator

Interpolator filter specification

Syntax

```
D = fdesign.interpolator(L)
D = fdesign.interpolator(L,RESPONSE)
D = fdesign.interpolator(L,CICRESPONSE,D)
D = fdesign.interpolator(L,RESPONSE,spec)
D = fdesign.interpolator(...,spec,specvalue1,specvalue2,...)
D = fdesign.interpolator(...,Fs)
d = fdesign.interpolator(...,MAGUNITS)
```

Description

`D = fdesign.interpolator(L)` constructs an interpolator filter specification object `D` with the `InterpolationFactor` property equal to the positive integer `L` and the `Response` property set to `'Nyquist'`. The default values for the transition width and stopband attenuation in the Nyquist design are 0.1π radians/sample and 80 dB. If `L` is unspecified, `L` defaults to 2.

`D = fdesign.interpolator(L,RESPONSE)` constructs a interpolator specification object with the interpolation factor `L` and the `'Response'` property set to one of the supported types.

`D = fdesign.interpolator(L,CICRESPONSE,D)` constructs a CIC or CIC compensator interpolator specification object with the interpolation factor, `L`, and `'Response'` property equal to `'CIC'` or `'CICCOMP'`. `D` is the differential delay. The differential delay, `D`, must precede any specification option.

`D = fdesign.interpolator(L,RESPONSE,spec)` constructs object `D` and sets its `Specification` property to `spec`. Entries in the `spec` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `spec` depend on the design type of the specifications object.

When you add the `spec` input argument, you must also add the `RESPONSE` input argument.

Because you are designing multirate filters, the specification options available are not the same as the specifications for designing single-rate filters with design methods such as `fdesign.lowpass`. The options are not case sensitive.

The interpolation factor `L` is not in the specification options. The different filter responses support different specifications. The following table lists the supported response types and specification options.

Design Method	Valid Specification Options
`'Arbitrary Magnitude'`	See `fdesign.arbmag` for a description of the specification entries. • `'N,F,A'` (default option) • `'N,B,F,A'`
`'Arbitrary Magnitude and Phase'`	See `fdesign.arbmagnphase` for a description of the specification entries. • `'N,F,H'` (default option) • `'N,B,F,H'`
`'Bandpass'`	See `fdesign.bandpass` for a description of the specification entries. • `'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'` (default option) • `'N,Fc1,Fc2'` • `'N,Fst1,Fp1,Fp2,Fst2'`
`'Bandstop'`	See `fdesign.bandstop` for a description of the specification entries. • `'N,Fc1,Fc2'` • `'N,Fp1,Fst1,Fst2,Fp2'` • `'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'` (default option)
`'CIC'`	`'Fp,Ast'` — Only valid specification. `Fp` is the passband frequency and `Ast` is the stopband attenuation in decibels. To specify a CIC interpolator, include the differential delay after `'CIC'` and before the filter specification: `'Fp,Ast'`. For example:

Design Method	Valid Specification Options
	d = fdesign.interpolator(2,'cic',4,'Fp,Ast',0.4,40);
'CIC Compensator'	See fdesign.ciccomp for a description of the specification entries. • 'Fp,Fst,Ap,Ast' (default option) • 'N,Fc,Ap,Ast' • 'N,Fp,Ap,Ast' • 'N,Fp,Fst' • 'N,Fst,Ap,Ast' To specify a CIC compensator interpolator, include the differential delay after 'CICCOMP' and before the filter specification. For example: d = fdesign.interpolator(2,'ciccomp',4);
'Differentiator'	'N' — filter order
'Gaussian'	'Nsym,BT — Nsym is the filter order in symbols and BT is the bandwidth-symbol time product. The specification must be preceded by an integer-valued SamplesPerSymbol.
'Halfband	See fdesign.halfband for a description of the specification entries. • 'TW,Ast' (default option) • 'N,TW' • ‘N’ • 'N,Ast' If you use the quasi-linear IIR design method, iirlinphase, with a halfband specification, the interpolation factor must be 2.

Design Method	Valid Specification Options
'Highpass'	See fdesign.highpass for a description of the specification entries. • 'Fst,Fp,Ast,Ap' (default option) • 'N,F3db' • 'N,Fc' • 'N,Fc,Ast,Ap' • 'N,Fp,Ast,Ap' • 'N,Fst,Ast,Ap' • 'N,Fst,Fp' • 'N,Fst,Ast,Ap' • 'N,Fst,Fp,Ast'
'Hilbert'	See fdesign.hilbert for a description of the specification entries. • 'N,TW' (default option) • 'TW,Ap'
'Inverse-sinc Lowpass'	See fdesign.isinclp for a description of the specification entries. • 'Fp,Fst,Ap,Ast' (default option) • 'N,Fc,Ap,Ast' • 'N,Fp,Fst' • 'N,Fst,Ap,Ast'
'Inverse-sinc Highpass'	See fdesign.isinchp for a description of the specification entries. • 'Fst,Fp,Ast,Ap' (default option) • 'N,Fc,Ast,Ap' • 'N,Fst,Fp' • 'N,Fst,Ast,Ap'

Design Method	Valid Specification Options
'Lowpass'	See `fdesign.lowpass` for a description of the specification entries. • 'Fp,Fst,Ap,Ast' (default option) • 'N,F3dB' • 'N,Fc' • 'N,Fc,Ap,Ast' • 'N,Fp,Ap,Ast' • 'N,Fp,Fst' • 'N,Fp,Fst,Ap' • 'N,Fp,Fst,Ast' • 'N,Fst,Ap,Ast'
'Nyquist'	See `fdesign.nyquist` for a description of the specification entries. For all Nyquist specifications, you must specify the *L*th band. This typically corresponds to the interpolation factor so that the nonzero samples of the upsampler output are preserved. • 'TW,Ast' (default option) • 'N' • 'N,Ast' • 'N,Ast'

`D = fdesign.interpolator(...,spec,specvalue1,specvalue2,...)` constructs an object `D` and sets its specifications at construction time.

`D = fdesign.interpolator(...,Fs)` adds the argument `Fs`, specified in Hz, to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.interpolator(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units.
- `'dB'` — specify the magnitude in dB (decibels).

- `'squared'` — specify the magnitude in power units.

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

Create an Interpolator Using `fdesign` Object

These examples show how to construct interpolating filter specification objects.

First, create a default specifications object without using input arguments except for the interpolation factor l.

```
l = 2;
d = fdesign.interpolator(2);
```

Now create an object by passing a specification option 'fst1,fp1,fp2,fst2,ast1,ap,ast2' and a design - the resulting object uses default values for all of the filter specifications. You must provide the design input argument when you include a specification.

```
d = fdesign.interpolator(8,'bandpass','fst1,fp1,fp2,fst2,ast1,ap,ast2');
```

Create another interpolating filter object, passing the specification values to the object rather than accepting the default values for, in this case, fp,fst,ap,ast.

```
d = fdesign.interpolator(3,'lowpass',.45,0.55,.1,60);
```

Now pass the filter specifications that correspond to the specifications - n,fc,ap,ast.

```
d = fdesign.interpolator(3,'ciccomp',1,2,'n,fc,ap,ast',...
20,0.45,.05,50);
```

With the specifications object in your workspace, design an interpolator using the equiripple design method.

```
hm = design(d,'equiripple','SystemObject',true);
```

Pass a new specification type for the filter, specifying the filter order.

```
d = fdesign.interpolator(5,'CIC',1,'fp,ast',0.05,55);
```

With the specifications object in your workspace, design an interpolator using the multisection design method.

```
hm = design(d,'multisection','SystemObject',true);
```

In this example, you specify a sampling frequency as the right most input argument. Here, it is set to 1000 Hz.

```
d = fdesign.interpolator(8,'bandpass','fst1,fp1,fp2,fst2,ast1,ap,ast2',...
    0.25,0.35,.55,.65,50,.05,1e3);
```

In this, the last example, use the linear option for the filter specification object and specify the stopband ripple attenuation in linear form.

```
d = fdesign.interpolator(4,'lowpass','n,fst,ap,ast',15,0.55,.05,0.001,...
    'linear');
```

Now design a CIC interpolator for a signal sampled at 19200 Hz. Specify the differential delay of 2 and set the attenuation of information beyond 50 Hz to be at least 80 dB.

```
% The filter object sampling frequency is (l x fs) where fs is the sampling frequency o

dd  = 2;     % Differential delay.
fp  = 50;    % Passband of interest.
ast = 80;    % Minimum attenuation of alias components in passband.
fs  = 600;   % Sampling frequency for input signal.
l   = 32;    % Interpolation factor.
d   = fdesign.interpolator(l,'cic',dd,'fp,ast',fp,ast,l*fs);
hm = design(d,'SystemObject',true);  %Use the default design method.
```

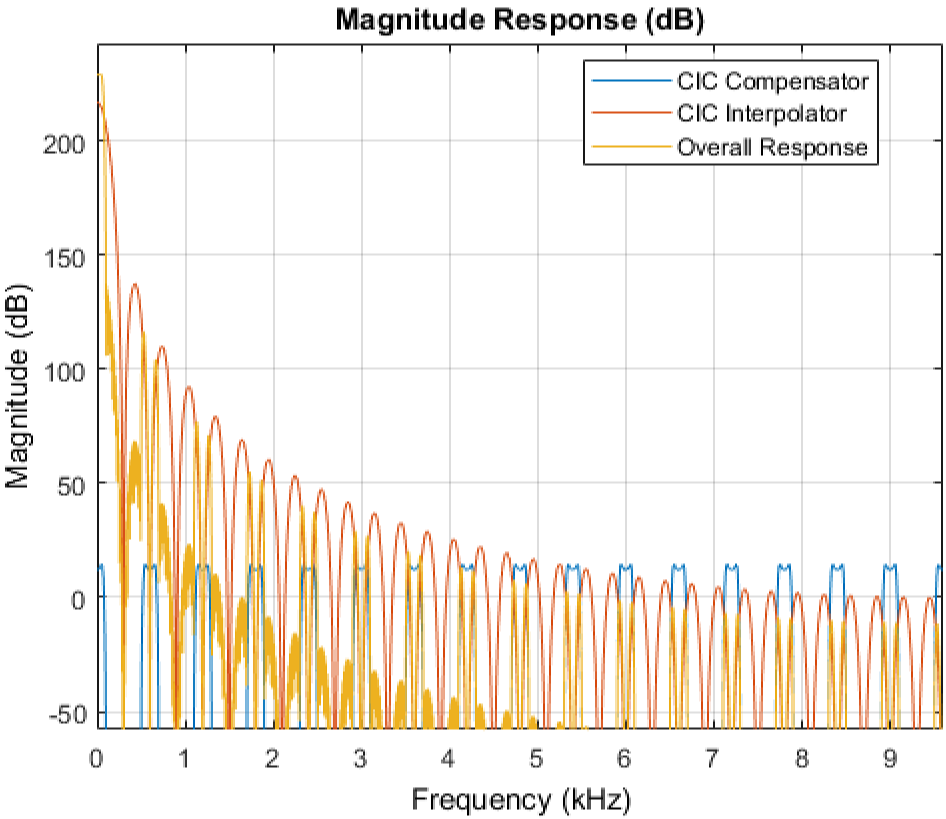
This next example results in a minimum-order CIC compensator that interpolates by 4 and compensates for the droop in the passband for the CIC filter hm from the previous example.

```
nsecs = hm.NumSections;
d = fdesign.interpolator(4,'ciccomp',dd,nsecs,...
50,100,0.1,80,fs);
hmc = design(d,'equiripple','SystemObject',true);
```

hmc is designed to compensate for hm. To see the effect of the compensating CIC filter, use `fvtool` to analyze both filters individually and include the compound filter response by cascading hm and hmc.

```
hfvt = fvtool(hmc,hm,cascade(hmc,hm),'fs',[fs,l*fs,l*fs],'ShowReference','off');
```

```
legend(hfvt,'CIC Compensator','CIC Interpolator',...
'Overall Response');
```

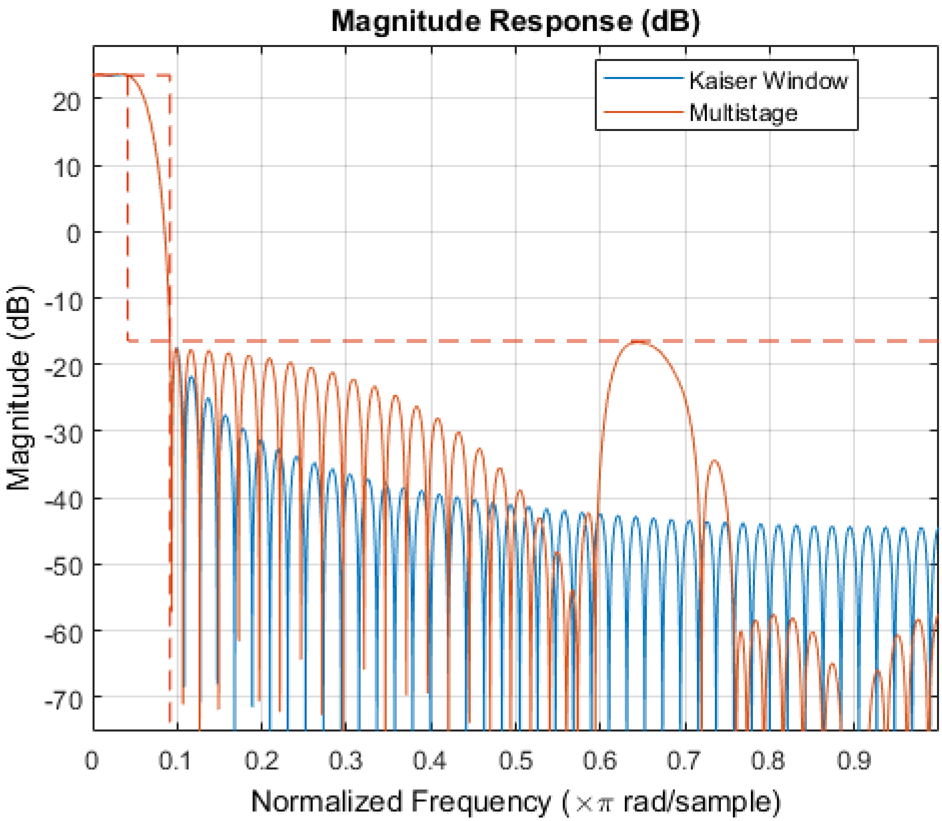


`fvtool` returns with this plot.

For the third example, use fdesign.interpolator to design a minimum-order Nyquist interpolator that uses a Kaiser window. For comparison, design a multistage interpolator as well and compare the responses.

```
l   = 15;   % Set the interpolation factor and the Nyquist band.
tw  = 0.05; % Specify the normalized transition width.
ast = 40;   % Set the minimum stopband attenuation in dB.
d   = fdesign.interpolator(l,'nyquist',l,tw,ast);
hm = design(d,'kaiserwin','SystemObject',true);
```

```
hm2 = design(d,'multistage','SystemObject',true); % Design the multistage interpolator
hfvt = fvtool(hm,hm2);
legend(hfvt,'Kaiser Window','Multistage')
```

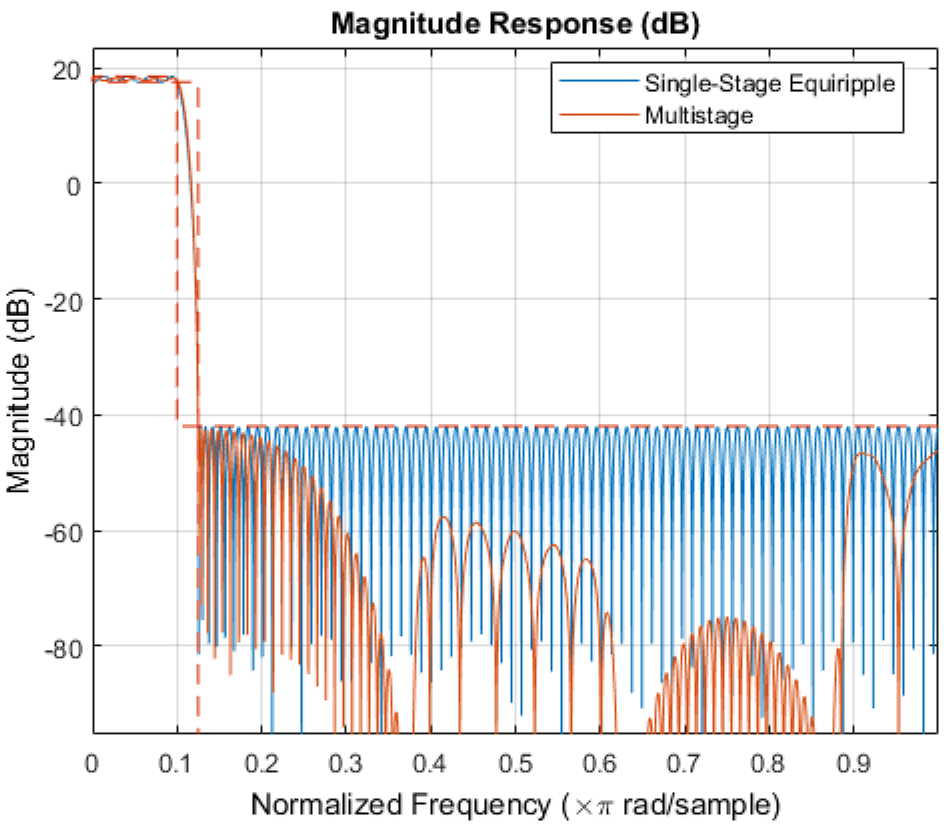


`fvtool` shows both responses.

Design a lowpass interpolator for an interpolation factor of 8. Compare the single-stage equiripple design to a multistage design with the same interpolation factor.

```
l = 8; % Interpolation factor.
d = fdesign.interpolator(l,'lowpass');
hm1 = design(d,'equiripple','SystemObject',true);
% Use halfband filters whenever possible.
hm2 = design(d,'multistage','usehalfbands',true,'SystemObject',true);
```

```
hfvt = fvtool(hm1,hm2);
legend(hfvt,'Single-Stage Equiripple','Multistage')
```



See Also

fdesign | fdesign.arbmag | fdesign.arbmagnphase | fdesign.interpolator | fdesign.rsrc | setspecs

Introduced in R2011a

fdesign.isinchp

Inverse sinc highpass filter specification

Syntax

```
D = fdesign.isinchp
D = fdesign.isinchp(SPEC)
D = fdesign.isinchp(SPEC,specvalue1,specvalue2,...)
D = fdesign.isinchp(specvalue1,specvalue2,specvalue3,specvalue4)
D = fdesign.isinchp(...,Fs)
D = fdesign.isinchp(...,MAGUNITS)
```

Description

`D = fdesign.isinchp` constructs an inverse sinc highpass filter specification object `D`, applying default values for the default specification `'Fst,Fp,Ast,Ap'`.

`D = fdesign.isinchp(SPEC)` constructs object `D` and sets the `Specification` property to `SPEC`. Entries in the `SPEC` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `SPEC` are shown below. The entries are not case sensitive.

- `'Fst,Fp,Ast,Ap'` (default `spec`)
- `'N,Fc,Ast,Ap'`
- `'N,Fst,Fp'`
- `'N,Fp,Ast,Ap'`
- `'N,Fst,Ast,Ap'`

The filter specifications are defined as follows:

- `Ast` — attenuation in the stopband in decibels (the default units). Also called Astop.
- `Ap` — amount of ripple allowed in the passband in decibels (the default units). Also called Apass.
- `Fp` — frequency at the start of the passband. Specified in normalized frequency units. Also called Fpass.

- `Fst` — frequency at the end of the stopband. Specified in normalized frequency units. Also called Fstop.
- `N` — filter order.

The filter design methods that apply to an inverse sinc highpass filter specification object change depending on the value of the `Specification` property. Use `designmethods` to determine which design method applies to a specific `Specification`.

Use `designopts` to see the available design options for a specific design method. Enter `help(D,METHOD)` at the MATLAB command line to obtain detailed information on the design options for a given design method, `METHOD`.

`D = fdesign.isinchp(SPEC,specvalue1,specvalue2,...)` constructs an object `D` and sets the specifications at construction time.

`D = fdesign.isinchp(specvalue1,specvalue2,specvalue3,specvalue4)` constructs an object `D` assuming the default `Specification` property `'Fst,Fp,Ast,Ap'`, using the values you provide in `specvalue1,specvalue2, specvalue3, and specvalue4`.

`D = fdesign.isinchp(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`D = fdesign.isinchp(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units
- `'dB'` — specify the magnitude in dB (decibels)
- `'squared'` — specify the magnitude in power units

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

The design method of `fdesign.isinchp` implements a filter with a passband magnitude response equal to:

$$H(\omega) = sinc(C(1-\omega))^{-P}$$

You can control the values of the sinc frequency factor, C, and the sinc power, P, using the 'SincFrequencyFactor' and 'SincPower' options in the design method. 'SincFrequencyFactor' and 'SincPower' default to 0.5 and 1 respectively.

Examples

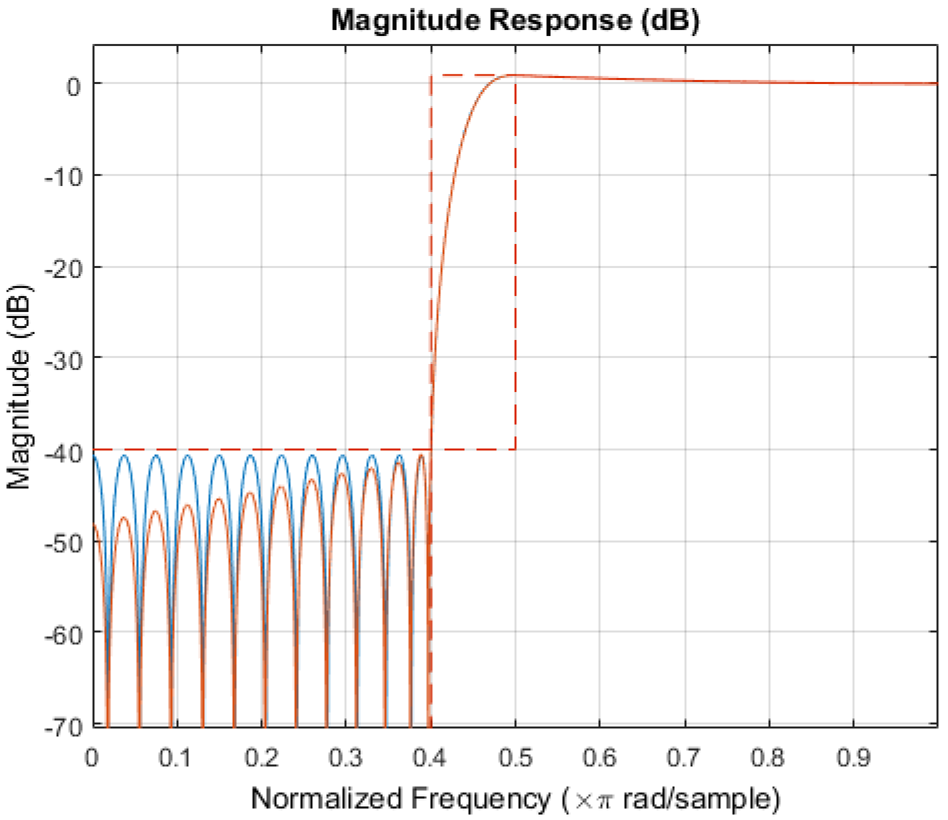
Construct an Inverse Sinc Highpass Filter Using fdesign Object

Design a minimum order inverse sinc highpass filter and shape the stopband to have a slope of 20 dB/radian/sample.

```
d = fdesign.isinchp('Fst,Fp,Ast,Ap',.4,.5,40,0.01);
Hd = design(d,'SystemObject',true);
```

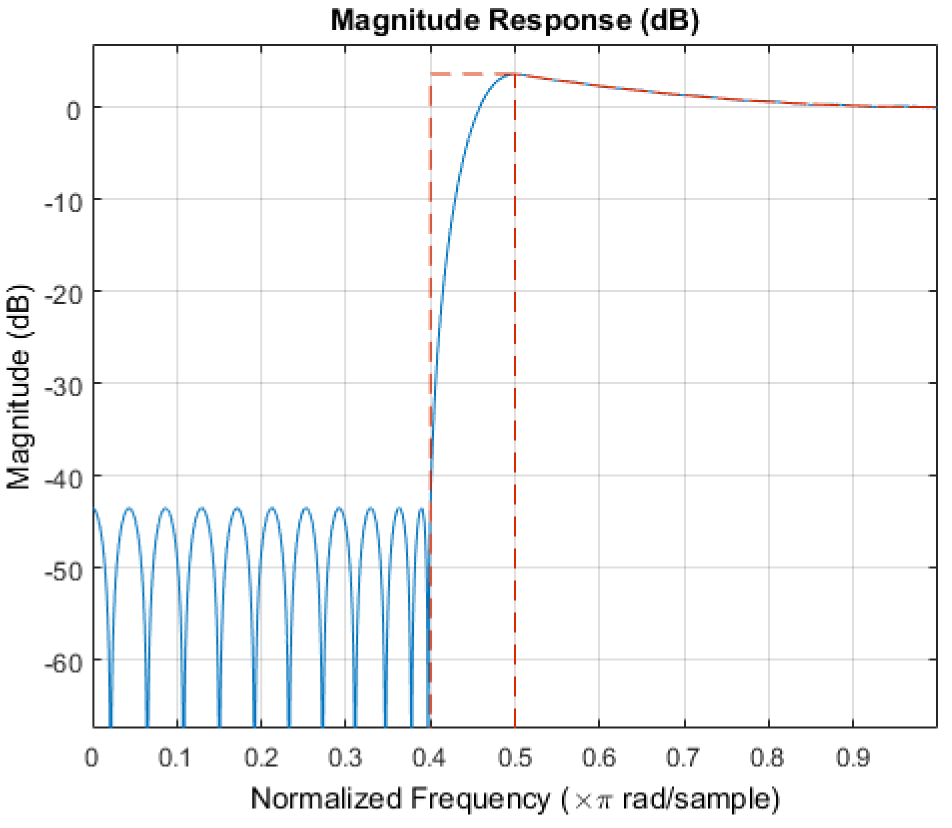
Shape the stopband to have a linear slope of 20 dB/rad/sample

```
Hd1 = design(d,'StopbandShape','linear','StopbandDecay',20,'SystemObject',...
    true);
fvtool(Hd,Hd1);
```



Design a 50th order inverse sinc highpass filter. Set the sinc frequency factor to 0.25, and the sinc power to 16 to achieve a magnitude response in the passband of the form H(?) = sinc(0.25*(1-?))^(-16) .

```
d = fdesign.isinchp('N,Fst,Fp',50,.4,.5);
Hd = design(d,'SincFrequencyFactor',0.25,'SincPower',16,...
    'SystemObject',true);
fvtool(Hd);
```



See Also

`design` | `designmethods` | `fdesign` | `fdesign.ciccomp` | `fdesign.highpass` | `fdesign.isinclp` | `fdesign.nyquist`

Introduced in R2011b

fdesign.isinclp

Inverse sinc lowpass filter specification

Syntax

```
d = fdesign.isinclp
d = fdesign.isinclp(spec)
d = fdesign.isinclp(spec,specvalue1,specvalue2,...)
d = fdesign.isinclp(specvalue1,specvalue2,specvalue3,specvalue4)
d = fdesign.isinclp(...,Fs)
d = fdesign.isinclp(...,MAGUNITS)
```

Description

`d = fdesign.isinclp` constructs an inverse sinc lowpass filter specification object `d`, applying default values for the default specification, `'Fp,Fst,Ap,Ast'`.

`d = fdesign.isinclp(`*`spec`*`)` constructs object `d` and sets its '`Specification`' to *`spec`*. Entries in the `spec` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for *`spec`* are shown below. The options are not case sensitive.

- `'Fp,Fst,Ap,Ast'` (default option)
- `'N,Fc,Ap,Ast'`
- `'N,Fp,Ap,Ast'`
- `'N,Fp,Fst'`
- `'N,Fst,Ap,Ast'`

The filter specifications are defined as follows:

- `Ast` — attenuation in the stopband in decibels (the default units). Also called Astop.
- `Ap` — amount of ripple allowed in the passband in decibels (the default units). Also called Apass.
- `Fp` — frequency at the start of the passband. Specified in normalized frequency units. Also called Fpass.

- `Fst` — frequency at the end of the stopband. Specified in normalized frequency units. Also called Fstop.
- `N` — filter order.

The filter design methods that apply to an inverse sinc lowpass filter specification object change depending on the `Specification`. Use `designmethods` to determine which design method applies to an object and its specification.

`d = fdesign.isinclp(spec,specvalue1,specvalue2,...)` constructs an object `d` and sets its specifications at construction time.

`d = fdesign.isinclp(specvalue1,specvalue2,specvalue3,specvalue4)` constructs an object `d` assuming the default `Specification` property `'Fp,Fst,Ap,Ast'`, using the values you provide in `specvalue1,specvalue2, specvalue3, and specvalue4`.

`d = fdesign.isinclp(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.isinclp(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units
- `'dB'` — specify the magnitude in dB (decibels)
- `'squared'` — specify the magnitude in power units

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

The design method of `fdesign.isinclp` implements a filter with a passband magnitude response equal to:

$$H(\omega) = sinc(C\omega)^{-P}$$

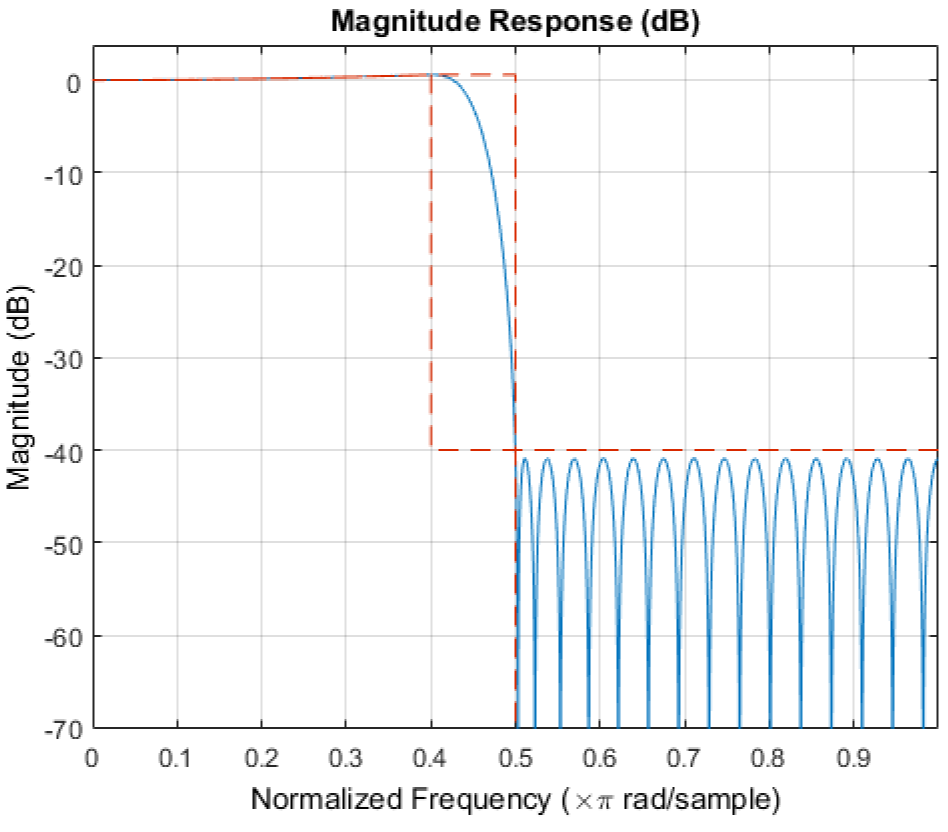
You can control the values of the sinc frequency factor, C, and the sinc power, P, using the `'SincFrequencyFactor'` and `'SincPower'` options in the `design` method. `'SincFrequencyFactor'` and `'SincPower'` default to 0.5 and 1 respectively.

Examples

Construct an Inverse Sinc Lowpass Filter Using `fdesign` Object

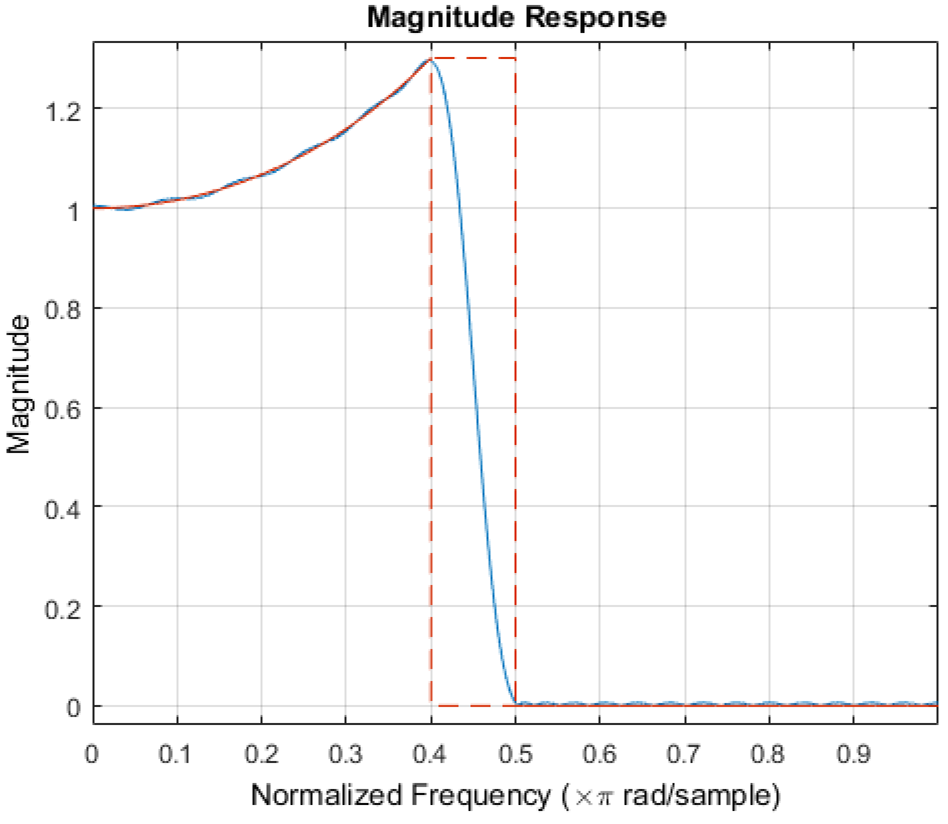
Pass the specifications for the default specification - 'Fp,Fst,Ap,Ast', as input arguments to the specifications object.

```
d = fdesign.isinclp(.4,.5,.01,40);
hd = design(d,'equiripple','SystemObject',true);
fvtool(hd);
```



Design a 50th order inverse sinc lowpass filter. Set the sinc frequency factor to 0.25 and the sinc power to 16 to achieve a magnitude response in the passband of the form H(w) = sinc(0.25*w)^(-16) .

```
d = fdesign.isinclp('N,Fp,Fst',50,.4,.5);
Hd = design(d,'SincFrequencyFactor',0.25,'SincPower',16,'SystemObject',...
    true);
fvtool(Hd, 'MagnitudeDisplay', 'Magnitude');
```



See Also

`fdesign` | `fdesign.bandpass` | `fdesign.bandstop` | `fdesign.halfband` | `fdesign.highpass` | `fdesign.lowpass` | `fdesign.nyquist`

Introduced in R2011a

fdesign.lowpass

Lowpass filter specification

Syntax

```
D = fdesign.lowpass
D = fdesign.lowpass(SPEC)
D = fdesign.lowpass(SPEC,specvalue1,specvalue2,...)
D = fdesign.lowpass(specvalue1,specvalue2,specvalue3,specvalue4)
D = fdesign.lowpass(...,Fs)
D = fdesign.lowpass(...,MAGUNITS)
```

Description

`D = fdesign.lowpass` constructs a lowpass filter specification object `D`, applying default values for the default specification option `'Fp,Fst,Ap,Ast'`.

`D = fdesign.lowpass(SPEC)` constructs object `D` and sets the `Specification` property to the entry in `SPEC`. Entries in `SPEC` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for `SPEC` are shown below. The options are not case sensitive.

Note: Specifications options marked with an asterisk require the DSP System Toolbox software.

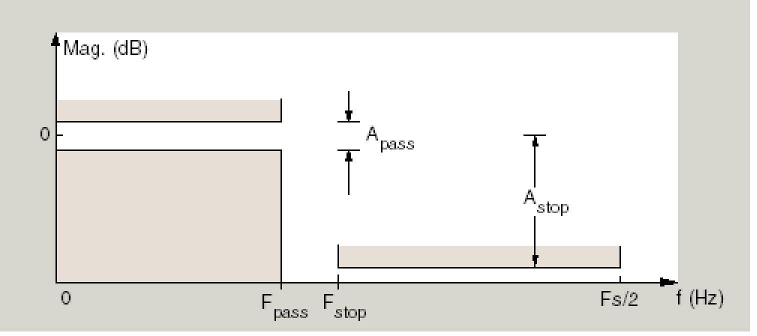
- `'Fp,Fst,Ap,Ast'` (default option)
- `'N,F3db'`
- `'N,F3db,Ap'` *
- `'N,F3db,Ap,Ast'` *
- `'N,F3db,Ast'` *
- `'N,F3db,Fst'` *
- `'N,Fc'`

- `'N,Fc,Ap,Ast'`
- `'N,Fp,Ap'`
- `'N,Fp,Ap,Ast'`
- `'N,Fp,Fst,Ap'` *
- `'N,Fp,F3db'` *
- `'N,Fp,Fst'`
- `'N,Fp,Fst,Ast'` *
- `'N,Fst,Ap,Ast'` *
- `'N,Fst,Ast'`
- `'Nb,Na,Fp,Fst'` *

The filter specifications are defined as follows:

- `Ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.
- `Ast` — attenuation in the stop band in decibels (the default units). Also called Astop.
- `F3db` — cutoff frequency for the point 3 dB point below the passband value. Specified in normalized frequency units.
- `Fc` — cutoff frequency for the point 6 dB point below the passband value. Specified in normalized frequency units.
- `Fp` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass.
- `Fst` — frequency at the end of the stop band. Specified in normalized frequency units. Also called Fstop.
- `N` — filter order.
- `Na` and `Nb` are the order of the denominator and numerator.

Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values like Fp and Fst are transition regions where the filter response is not explicitly defined.

D = fdesign.lowpass(SPEC,specvalue1,specvalue2,...) constructs an object D and sets the specification values at construction time using specvalue1, specvalue2, and so on for all of the specification variables in SPEC.

D = fdesign.lowpass(specvalue1,specvalue2,specvalue3,specvalue4) constructs an object D with values for the default Specification property 'Fp,Fst,Ap,Ast' using the specifications you provide as input arguments specvalue1,specvalue2,specvalue3,specvalue4.

D = fdesign.lowpass(...,Fs) adds the argument Fs, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

D = fdesign.lowpass(...,MAGUNITS) specifies the units for any magnitude specification you provide in the input arguments. MAGUNITS can be one of

- 'linear' — specify the magnitude in linear units
- 'dB' — specify the magnitude in dB (decibels)
- 'squared' — specify the magnitude in power units

When you omit the MAGNUNITS argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

Lowpass Filtering of Sinusoids

Lowpass filter a discrete-time signal consisting of two sine waves.

Create a lowpass filter specification object. Specify the passband frequency to be 0.15π rad/sample and the stopband frequency to be 0.25π rad/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB.

```
d = fdesign.lowpass('Fp,Fst,Ap,Ast',0.15,0.25,1,60);
```

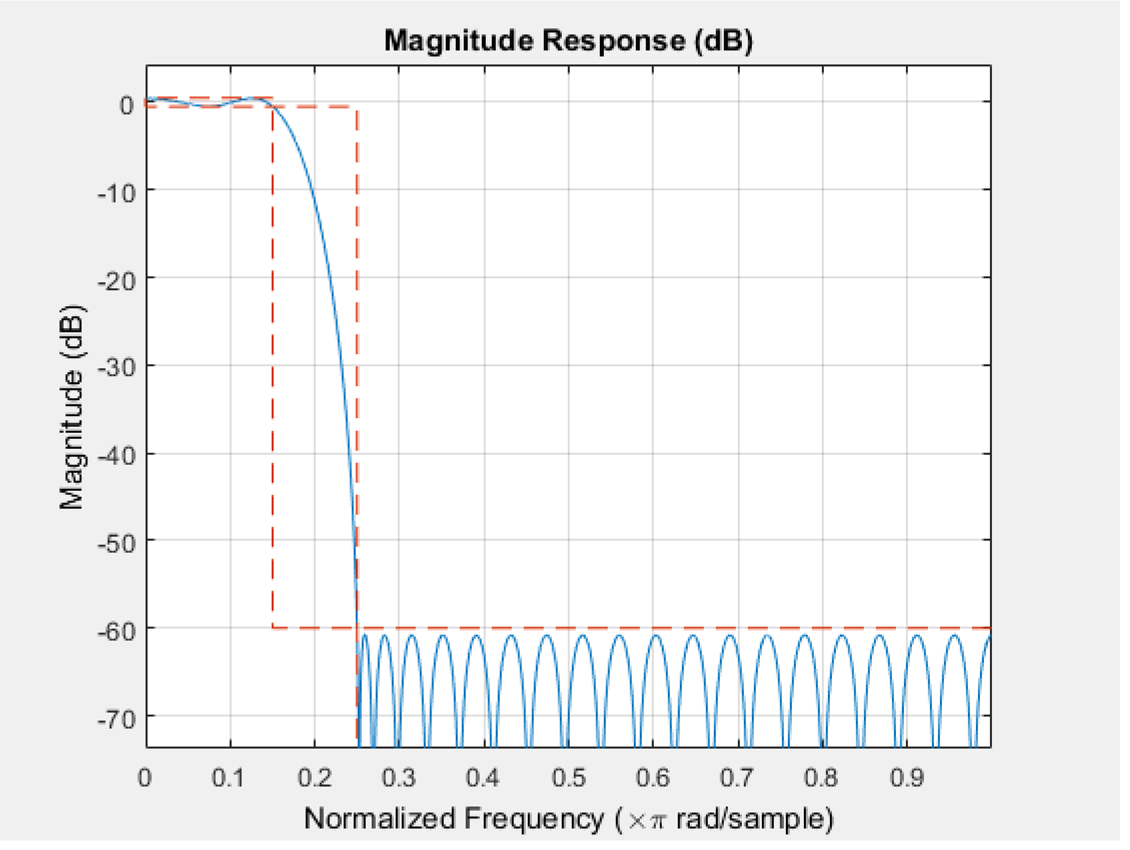
Query the valid design methods for your filter specification object, `d`.

```
designmethods(d)
```

```
Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

Create an FIR equiripple filter and view the filter magnitude response with `fvtool`.

```
Hd = design(d,'equiripple');
fvtool(Hd)
```



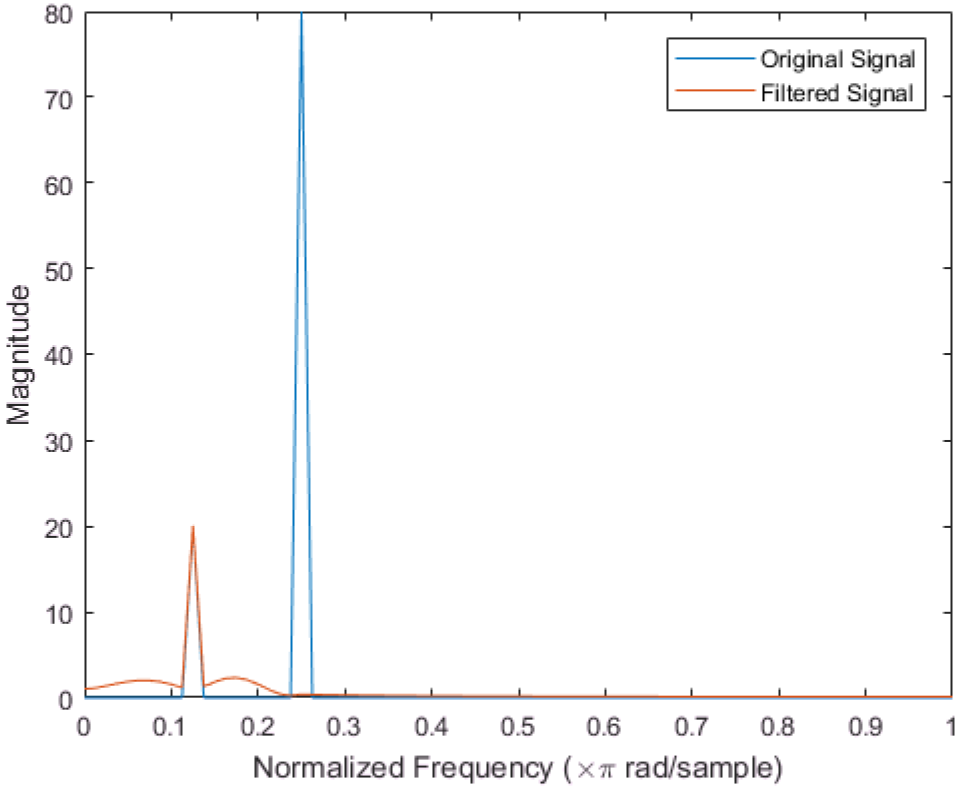
Create a signal consisting of the sum of two discrete-time sinusoids with frequencies of $\pi/8$ and $\pi/4$ rad/sample and amplitudes of 1 and 0.25, respectively. Filter the discrete-time signal with the FIR equiripple filter object, Hd.

```
n = 0:159;
x = (0.25*cos((pi/8)*n)+sin((pi/4)*n));
y = filter(Hd,x);
```

Compute the Fourier transform of the original signal and the filtered signal. Verify that the high-frequency component has been filtered out.

```
freq = 0:(2*pi)/160:pi;
xdft = fft(x);
```

```
ydft = fft(y);

figure
plot(freq/pi,abs(xdft(1:length(x)/2+1)))
hold on
plot(freq/pi,abs(ydft(1:length(y)/2+1)))
hold off

legend('Original Signal','Filtered Signal')
ylabel('Magnitude')
xlabel('Normalized Frequency (\times\pi rad/sample)')
```



Window Method Lowpass Design

Create a filter of order 10 with a 6-dB frequency of 9.6 kHz and a sampling frequency of 48 kHz. Look at the available design methods.

```
d = fdesign.lowpass('N,Fc',10,9600,48000);
designmethods(d)
```

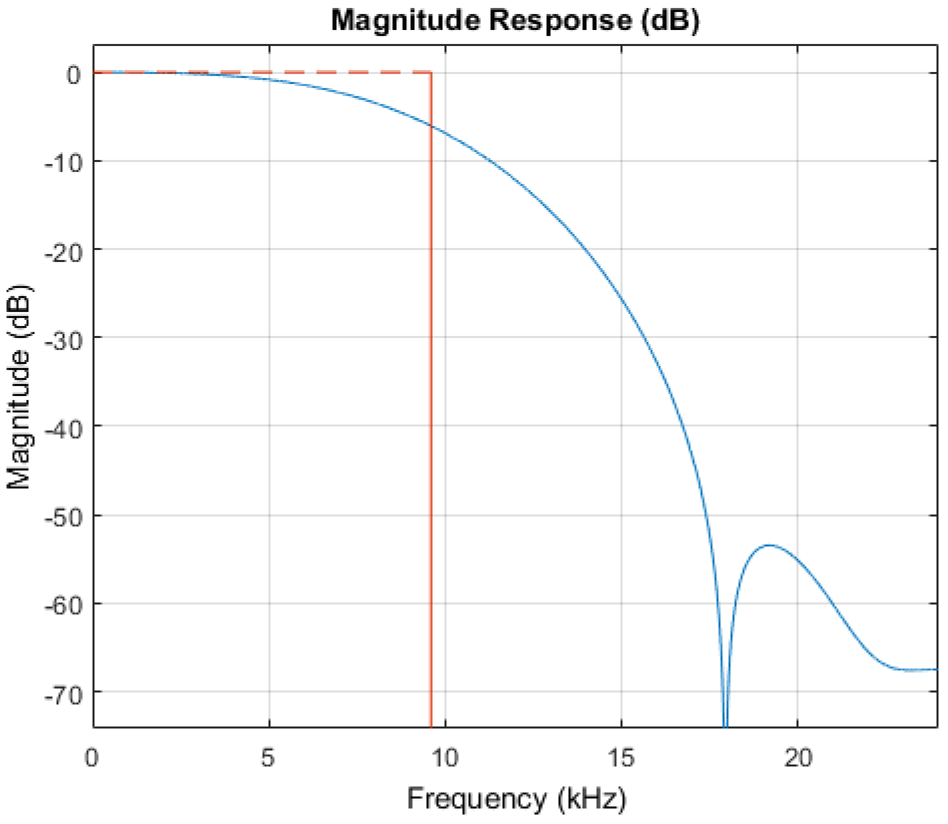
```
Design Methods for class fdesign.lowpass (N,Fc):


window
```

The only valid design method is the FIR window method. Design the filter.

```
Hd = design(d);
```

Display the filter magnitude response. The -6 dB point is at 9.6 kHz, as expected.

```
fvtool(Hd)
```



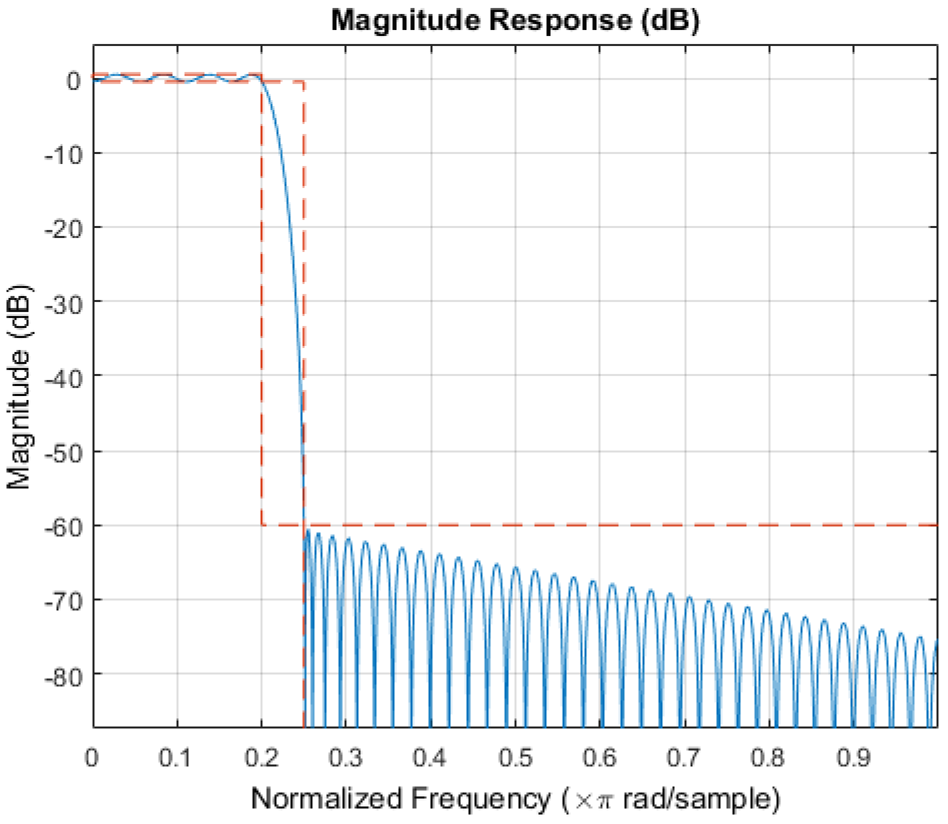
Stopband Shape and Decay

Create an FIR equiripple filter with a passband frequency of 0.2π rad/sample, a stopband frequency of 0.25π rad/sample, a passband ripple of 1 dB, and a stopband attenuation of 60 dB. Design the filter with a 20 dB/rad/sample linear stopband.

```
D = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.25,1,60);
Hd = design(D,'equiripple','StopbandShape','linear','StopbandDecay',20);
```

Visualize the frequency response of the filter.

```
fvtool(Hd)
```



See Also

See Also

design | designmethods | fdesign

Introduced in R2009a

fdesign.notch

Notch filter specification

Syntax

```
d = fdesign.notch(specstring, value1, value2, ...)
d = fdesign.notch(n,f0,q)
d = fdesign.notch(...,Fs)
d = fdesign.notch(...,MAGUNITS)
```

Description

`d = fdesign.notch(specstring, value1, value2, ...)` constructs a notch filter specification object `d`, with the specification set to `specstring` and values provided for all members of the `specstring`. The possible specification entries, which are not case sensitive, are listed as follows:

- 'N,F0,Q' (default)
- 'N,F0,Q,Ap'
- 'N,F0,Q,Ast'
- 'N,F0,Q,Ap,Ast'
- 'N,F0,BW'
- 'N,F0,BW,Ap'
- 'N,F0,BW,Ast'
- 'N,F0,BW,Ap,Ast'

where the variables are defined as follows:

- N - Filter Order (must be even)
- F0 - Center Frequency
- Q - Quality Factor
- BW - 3-dB Bandwidth

- Ap - Passband Ripple (decibels)
- Ast - Stopband Attenuation (decibels)

Different specification entries, resulting in different specification objects, may have different design methods available. Use the function `designmethods` to get a list of design methods available for a given specification. For example:

```
d  = fdesign.notch('N,F0,Q,Ap',6,0.5,10,1);
designmethods(d);
```

`d = fdesign.notch(n,f0,q)` constructs a notch filter specification object using the default `specstring ('N,F0,Q')` and setting the corresponding values to `n`, `f0`, and `q`.

By default, all frequency specifications are assumed to be in normalized frequency units. All magnitude specifications are assumed to be in decibels.

`d = fdesign.notch(...,Fs)` constructs a notch filter specification object while providing the sampling frequency of the signal to be filtered. `Fs` must be specified as a scalar trailing the other values provided. If you specify an `Fs`, it is assumed to be in Hz, as are all the other frequency values provided.

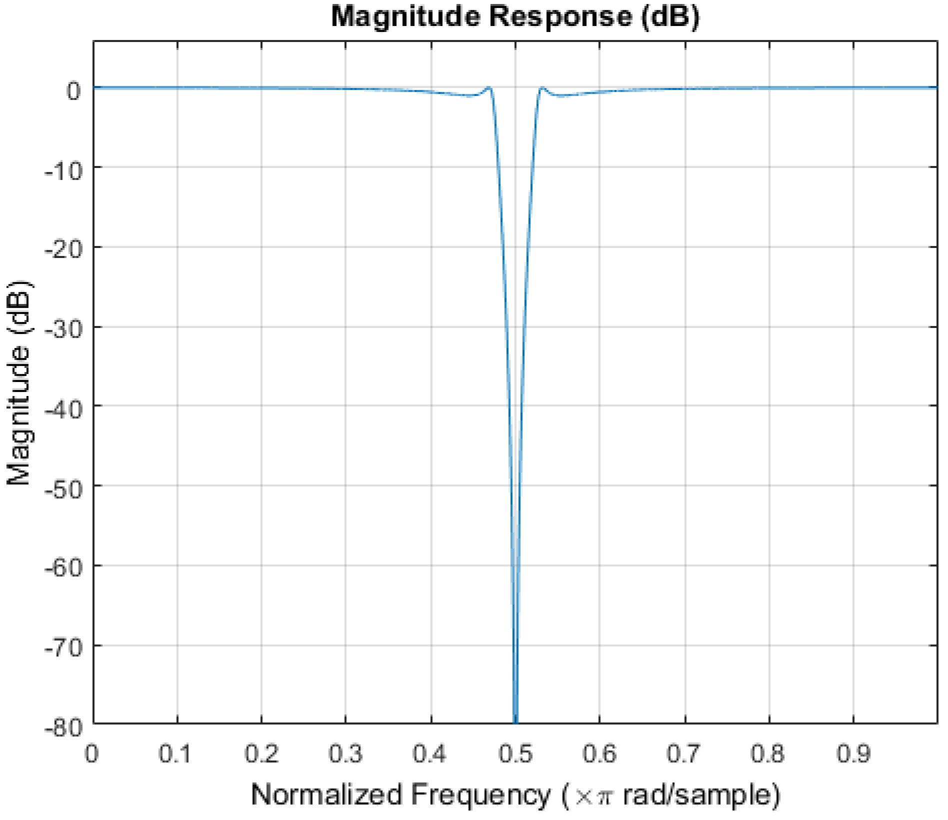
`d = fdesign.notch(...,MAGUNITS)` constructs a notch filter specification while providing the units for any magnitude specification given. `MAGUNITS` can be one of the following: `'linear'`, `'dB'`, or `'squared'`. If this argument is omitted, `'dB'` is assumed. The magnitude specifications are always converted and stored in decibels regardless of how they were specified. If `Fs` is provided, `MAGUNITS` must follow `Fs` in the input argument list.

Examples

Design a Notch Filter

Design a notching filter with a passband ripple of 1 dB.

```
d  = fdesign.notch('N,F0,Q,Ap',6,0.5,10,1);
Hd = design(d,'SystemObject',true);
fvtool(Hd)
```



See Also

`fdesign` | `fdesign.peak`

Introduced in R2011a

fdesign.nyquist

Nyquist filter specification

Syntax

```
d = fdesign.nyquist
d = fdesign.nyquist(l, spec)
d = fdesign.nyquist(l,spec,specvalue1,specvalue2,...)
d = fdesign.nyquist(l,specvalue1,specvalue2)
d = fdesign.nyquist(...,fs)
d = fdesign.nyquist(...,magunits)
```

Description

`d = fdesign.nyquist` constructs a Nyquist or L-band filter specification object `d`, applying default values for the properties `tw` and `ast`. By default, the filter object designs a minimum-order half-band (L=2) Nyquist filter.

Using `fdesign.nyquist` along with `design` method generates a System object, if the `'SystemObject'` flag in the `design` method is set to `true`.

`d = fdesign.nyquist(l, spec)` constructs object `d` and sets its `Specification` property to *`spec`*. Use `l` to specify the desired value for L. L = 2 designs a half-band FIR filter, L = 3 a third-band FIR filter, and so on. When you use a Nyquist filter as an interpolator, l or L is the interpolation factor. The first input argument must be l when you are not using the default syntax `d = fdesign.nyquist`.

Entries in the `spec` represent various filter response features, such as the filter order, that govern the filter design. Valid entries for *`spec`* are shown below. The entries are not case sensitive.

- `tw`,`ast` (default option)
- `n`,`tw`
- `n`
- `n`,`ast`

where,

- `ast` — attenuation in the stop band in decibels (the default units).
- `n` — filter order.
- `tw` — width of the transition region between the pass and stop bands. Specified in normalized frequency units.

The filter design methods that apply to a Nyquist filter specification object change depending on the `Specification` option. Use `designmethods` to determine which design method applies to an object and its specification option. Different filter design methods also have options that you can specify. Use `designopts` with the design method to see the available options. For example:

```
f=fdesign.nyquist(4,'N,TW');
designmethods(f)
```

`d = fdesign.nyquist(l,spec,specvalue1,specvalue2,...)` constructs an object `d` and sets its specification to *`spec`*, and the specification values to `specvalue1`, `specvalue2`, and so on at construction time.

`d = fdesign.nyquist(l,specvalue1,specvalue2)` constructs an object `d` with the values you provide in `l`, `specvalue1,specvalue2` as the values for `l`, `tw` and `ast`.

`d = fdesign.nyquist(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.nyquist(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of

- `linear` — specify the magnitude in linear units
- `dB` — specify the magnitude in dB (decibels)
- `squared` — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Limitations of the Nyquist fdesign Object

Using Nyquist filter specification objects with the `equiripple` design method imposes a few limitations on the resulting filter, caused by the `equiripple` design algorithm.

- When you request a minimum-order design from `equiripple` with your Nyquist object, the design algorithm might not converge and can fail with a filter convergence error.
- When you specify the order of your desired filter, and use the `equiripple` design method, the design might not converge.
- Generally, the following specifications, alone or in combination with one another, can cause filter convergence problems with Nyquist objects and the `equiripple` design method.
  - very high order
  - small transition width
  - very large stopband attenuation

Note that halfband filters (filters where band = 2) do not exhibit convergence problems.

When convergence issues arise, either in the cases mentioned or in others, you might be able to design your filter with the `kaiserwin` method.

In addition, if you use Nyquist objects to design decimators or interpolators (where the interpolation or decimation factor is not a prime number), using multistage filter designs might be your best approach.

Examples

Construct a Nyquist Filter Using `fdesign` Object

These examples show how to construct a Nyquist filter specification object.

First, create a default specifications object without using input arguments.

```
d = fdesign.nyquist;
```

Now create an object by passing a specification type 'n,ast' - the resulting object uses default values for n and ast.

```
d = fdesign.nyquist(2,'n,ast');
```

Create another Nyquist filter object, passing the specification values to the object rather than accepting the default values for n and ast.

```
d = fdesign.nyquist(3,'n,ast',42,80)
```

```
d =

  nyquist with properties:

               Response: 'Nyquist'
          Specification: 'N,Ast'
            Description: {2×1 cell}
    NormalizedFrequency: 1
            FilterOrder: 42
                  Astop: 80
                   Band: 3
```

Finally, pass the filter specifications that correspond to the default Specification - tw,ast. When you pass only the values, fdesign.nyquist assumes the default Specification option.

```
d = fdesign.nyquist(4,.01,80)
```

```
d =

  nyquist with properties:

               Response: 'Nyquist'
          Specification: 'TW,Ast'
            Description: {2×1 cell}
    NormalizedFrequency: 1
        TransitionWidth: 0.0100
                  Astop: 80
                   Band: 4
```

Now design a Nyquist filter using the kaiserwin design method.

```
hd = design(d,'kaiserwin','SystemObject',true);
```

Create two equiripple Nyquist 4th-band filters with and without a nonnegative zero phase response:

```
f = fdesign.nyquist(4,'N,TW',12,0.2);
```

Equiripple Nyquist 4th-band filter with nonnegative zero phase response

```
Hd1 = design(f,'equiripple','zerophase',true,'SystemObject',true);
```

Equiripple Nyquist 4th-band filter with 'ZeroPhase' set to false 'zerophase',false is the default

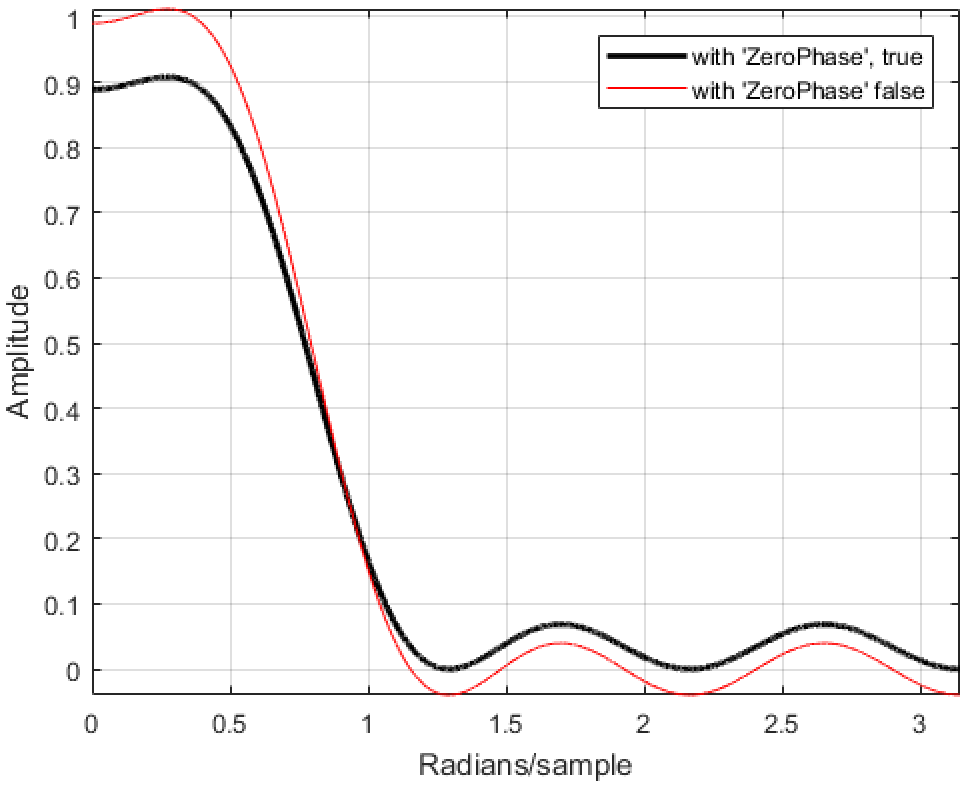
```
Hd2 = design(f,'equiripple','zerophase',false,'SystemObject',true);
```

Obtain real-valued amplitudes (not magnitudes)

```
[Hr_zerophase,W] = zerophase(Hd1);
[Hr,W] = zerophase(Hd2);
```

Plot and compare response

```
plot(W,Hr_zerophase,'k','linewidth',2);
xlabel('Radians/sample'); ylabel('Amplitude');
hold on;
plot(W,Hr,'r');
axis tight; grid on;
legend('with ''ZeroPhase'', true','with ''ZeroPhase'' false');
```



Note that the amplitude of the zero phase response (black line) is nonnegative for all frequencies.

The 'ZeroPhase' option is valid only for equiripple Nyquist designs with the 'N,TW' specification. You cannot specify 'MinPhase' and 'ZeroPhase' to be simultaneously 'true'.

See Also

`fdesign` | `fdesign.interpolator` | `fdesign.halfband` | `fdesign.interpolator` | `fdesign.rsrc` | `zerophase`

Introduced in R2011a

fdesign.octave

Octave filter specification

Syntax

```
d = fdesign.octave(l)
d = fdesign.octave(l, MASK)
d = fdesign.octave(l, MASK, spec)
d = fdesign.octave(..., Fs)
```

Description

`d = fdesign.octave(l)` constructs an octave filter specification object `d`, with `l` bands per octave. The default value for `l` is one.

Note: The filters created by `fdesign.octave` comply with the ANSI® S1.11-2004 and IEC 61260:1995 standards.

`d = fdesign.octave(l, MASK)` constructs an octave filter specification object `d` with `l` bands per octave and `MASK` specification for the FVTool. The available values for mask are:

- `'class 0'`
- `'class 1'`
- `'class 2'`

`d = fdesign.octave(l, MASK, spec)` constructs an octave filter specification object `d` with `l` bands per octave, `MASK` specification for the FVTool, and the `spec` specification string. The specification strings available are:

- `'N, F0'`

(not case sensitive), where:

- N is the filter order
- F0 is the center frequency. The center frequency is specified in normalized frequency units assuming a sampling frequency of 48 kHz, unless a sampling frequency in Hz is included in the specification: `d = fdesign.octave(..., Fs)`. If you specify an invalid center frequency, a warning is issued and the center frequency is rounded to the nearest valid value. You can determine the valid center frequencies for your design by using `validfrequencies` with your octave filter specification object. For example:

```
d = fdesign.octave(1,'Class 1','N,F0',6,1000,44.1e3);
validcenterfreq = validfrequencies(d);
```

Valid center frequencies:

- Must be greater than 20 Hz and less than 20 kHz if you specify a sampling frequency. The range 20 Hz to 20 kHz is the standard range of human hearing.
- Are calculated according to the following algorithm if the number of bands per octave, L, is even:

```
G = 10^(3/10);
x = -1000:1350;
validcenterfreq = 1000*(G.^((2*x-59)/(2*L)));
validcenterfreq = validcenterfreq(validcenterfreq>20 & validcenterfreq<2e4);
```

- Are calculated according to the following algorithm if the number of bands per octave, L, is odd:

```
G = 10^(3/10);
x = -1000:1350;
validcenterfreq = 1000*(G.^((x-30)/L));
validcenterfreq = validcenterfreq(validcenterfreq>20 & validcenterfreq<2e4);
```

Only center frequencies greater than 20 and less than 20,000 are retained. The center frequencies and the corresponding upper band frequencies must be less than the Nyquist frequency, which is half the sampling rate (`samplingfreq`). The vector of upper band frequencies (`upperbandfreq`) corresponding to the center frequencies (`validcenterfreq`) is computed using the following algorithm:

```
upperbandfreq = validcenterfreq.*(G^(1/(2*L)));
```

The algorithm removes the center frequencies whose corresponding upper band frequencies do not obey the Nyquist rule.

```
validcenterfreq = validcenterfreq(upperbandfreq < samplingfreq/2);
```

If you do not specify a sampling frequency, `fdesign.octave` assumes a `samplingfreq` of 48 kHz. To obtain valid normalized center frequencies, the remaining center frequencies are divided by 24,000.

```
validcenterfreq = validcenterfreq/24000;
```

Examples

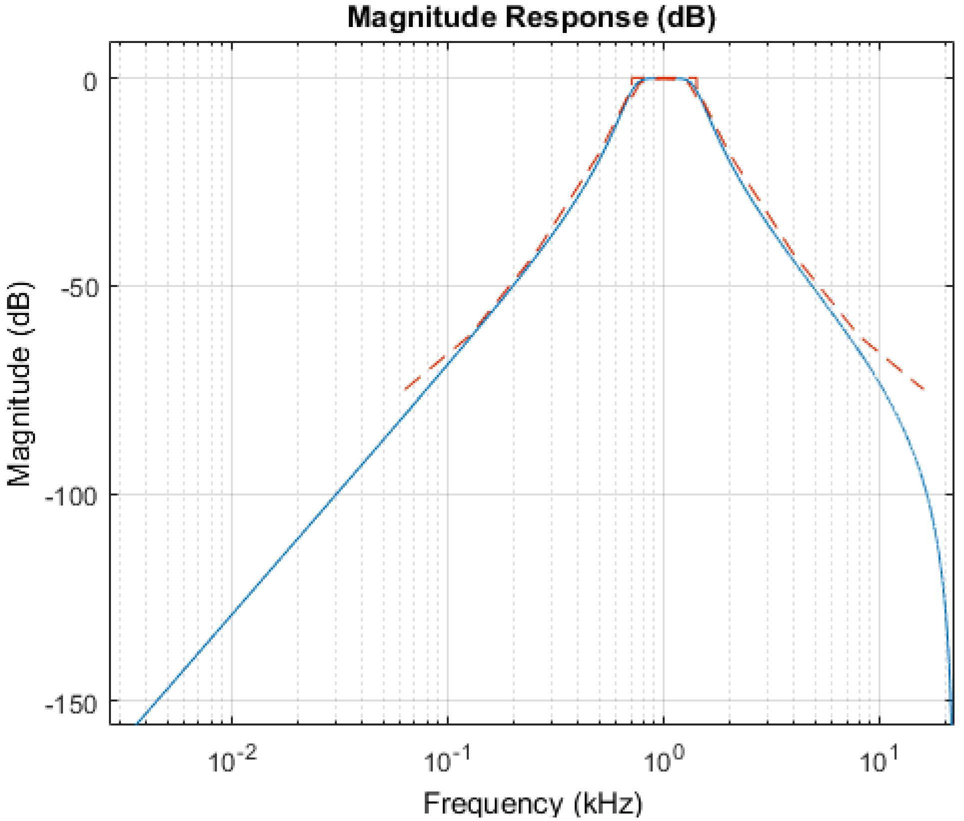
Design an Octave Band Filter

Design a sixth order, octave-band class 0 filter with a center frequency of 1000 Hz and, a sampling frequency of 44.1 kHz.

```
d = fdesign.octave(1,'Class 0','N,F0',6,1000,44100)
Hd = design(d,'SystemObject',true)
fvtool(Hd)
```

```
d =

  octave with properties:

               Response: 'Octave and Fractional Octave'
         BandsPerOctave: 1
                   Mask: 'Class 0'
          Specification: 'N,F0'
            Description: {2×1 cell}
    NormalizedFrequency: 0
                     Fs: 44100
            FilterOrder: 6
                     F0: 1000


Hd =

  dsp.BiquadFilter with properties:

                  Structure: 'Direct form II'
            SOSMatrixSource: 'Property'
                  SOSMatrix: [3×6 double]
                ScaleValues: [4×1 double]
          InitialConditions: 0
    OptimizeUnityScaleValues: true
```

```
Use get to show all properties
```



The figure shows the magnitude response plot of the filter. The logarithmic scale for frequency is automatically set by FVTool for the octave filters.

See Also

`fdesign`

Introduced in R2011a

fdesign.parameq

Parametric equalizer filter specification

Note: The `fdesign.parameq` function will be removed from DSP System Toolbox in a future release. Existing instances of the function continue to run. For new code, use the `fdesign.parameq` function from Audio System Toolbox instead.

Syntax

```
d = fdesign.parameq(spec, specvalue1, specvalue2, ...)
d = fdesign.parameq(... fs)
```

Description

`d = fdesign.parameq(spec, specvalue1, specvalue2, ...)` constructs a parametric equalizer filter design object, where `spec` is a non-case sensitive specification string. The choices for `spec` are as follows:

- `'F0, BW, BWp, Gref, G0, GBW, Gp'` (minimum order default)
- `'F0, BW, BWst, Gref, G0, GBW, Gst'`
- `'F0, BW, BWp, Gref, G0, GBW, Gp, Gst'`
- `'N, F0, BW, Gref, G0, GBW'`
- `'N, F0, BW, Gref, G0, GBW, Gp'`
- `'N, F0, Fc, Qa, G0'`
- `'N, F0, Fc, S, G0'`
- `'N, F0 ,BW, Gref, G0, GBW, Gst'`
- `'N, F0, BW, Gref, G0, GBW, Gp, Gst'`
- `'N, Flow, Fhigh, Gref, G0, GBW'`
- `'N, Flow, Fhigh, Gref, G0, GBW, Gp'`
- `'N, Flow, Fhigh, Gref, G0, GBW, Gst'`

- `'N, Flow, Fhigh, Gref, G0, GBW, Gp, Gst'`

where the parameters are defined as follows:

- BW — Bandwidth
- BWp — Passband Bandwidth
- BWst — Stopband Bandwidth
- Gref — Reference Gain (decibels)
- G0 — Center Frequency Gain (decibels)
- GBW — Gain at which Bandwidth (BW) is measured (decibels)
- Gp — Passband Gain (decibels)
- Gst — Stopband Gain (decibels)
- N — Filter Order
- F0 — Center Frequency
- Fc— Cutoff frequency
- Fhigh - Higher Frequency at Gain GBW
- Flow - Lower Frequency at Gain GBW
- Qa-Quality Factor
- S-Slope Parameter for Shelving Filters

Regardless of the specification string chosen, there are some conditions that apply to the specification parameters. These are as follows:

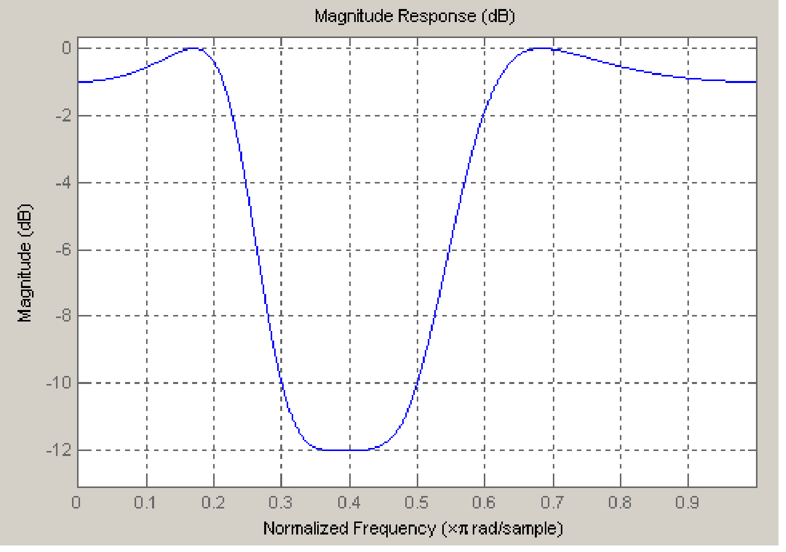
- Specifications for parametric equalizers must be given in decibels
- To boost the input signal, set `G0 > Gref`; to cut, set `Gref > G0`
- For boost: `G0 > Gp > GBW > Gst > Gref`; For cut: `G0 < Gp < GBW < Gst < Gref`
- Bandwidth must satisfy: `BWst > BW > BWp`

`d = fdesign.parameq(... fs)` adds the input sampling frequency. Fs must be specified as a scalar trailing the other numerical values provided, and is assumed to be in Hz.

Examples

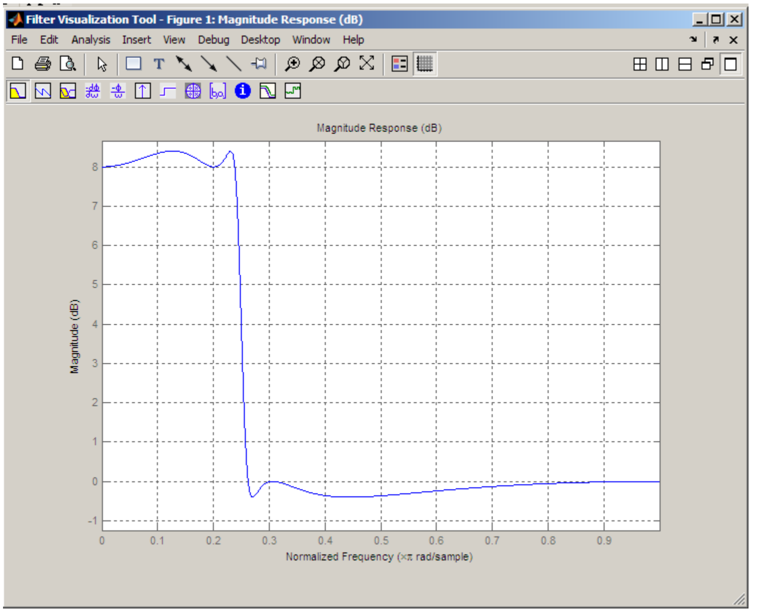
Design a Chebyshev Type II parametric equalizer filter that cuts by 12 dB:

```
d = fdesign.parameq('N,Flow,Fhigh,Gref,G0,GBW,Gst',...
      4,.3,.5,0,-12,-10,-1);
Hd = design(d,'cheby2');
fvtool(Hd)
```



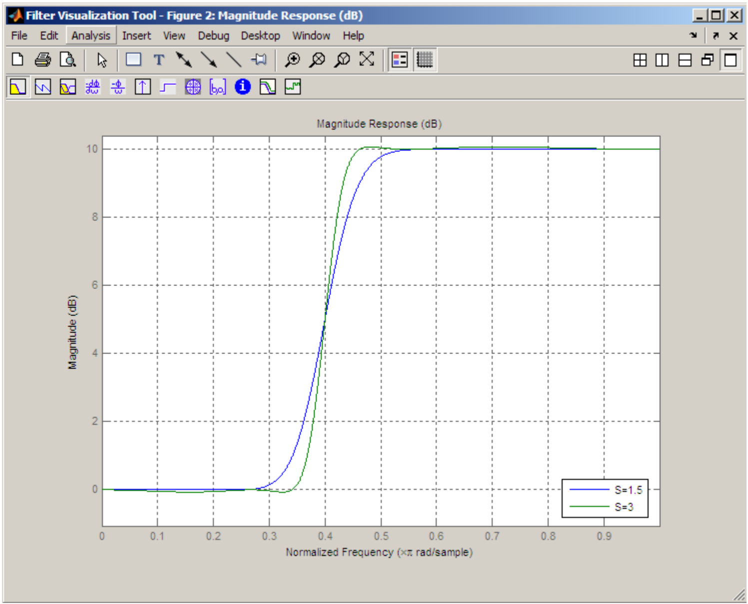
Design a 4th order audio lowpass (F0 = 0) shelving filter with cutoff frequency of Fc = 0.25, quality factor Qa =10, and boost gain of G0 = 8 dB:

```
d = fdesign.parameq('N,F0,Fc,Qa,G0',4,0,0.25,10,8);
Hd = design(d);
fvtool(Hd)
```



Design 4th-order highpass shelving filters with S=1.5 and S=3:

```
N=4;
F0 = 1;
Fc = .4; % Cutoff Frequency
G0 = 10;
S = 1.5;
S2=3;
f = fdesign.parameq('N,F0,Fc,S,G0',N,F0,Fc,S,G0);
h1 = design(f);
f.S=3;
h2=design(f);
hfvt=fvtool([h1 h2]);
set(hfvt,'Filters',[h1 h2]);
legend(hfvt,'S=1.5','S=3');
```



See Also

`fdesign`

Introduced in R2011a

fdesign.peak

Peak filter specification

Syntax

```
d = fdesign.peak(specstring, value1, value2, ...)
d = fdesign.peak(n,f0,q)
d = fdesign.peak(...,Fs)
d = fdesign.peak(...,MAGUNITS)
```

Description

`d = fdesign.peak(specstring, value1, value2, ...)` constructs a peaking filter specification object `d`, with specification set to `specstring` and values provided for all members of the `specstring`. The possible specification options, which are not case sensitive, are listed as follows:

- 'N,F0,Q' (default)
- 'N,F0,Q,Ap'
- 'N,F0,Q,Ast'
- 'N,F0,Q,Ap,Ast'
- 'N,F0,BW'
- 'N,F0,BW,Ap'
- 'N,F0,BW,Ast'
- 'N,F0,BW,Ap,Ast'

where the variables are defined as follows:

- N - Filter Order (must be even)
- F0 - Center Frequency
- Q - Quality Factor
- BW - 3-dB Bandwidth
- Ap - Passband Ripple (decibels)

- Ast - Stopband Attenuation (decibels)

Different specification options, resulting in different specification objects, may have different design methods available. Use the function `designmethods` to get a list of design methods available for a given specification. For example:

```
>> d  = fdesign.peak('N,F0,Q,Ap',6,0.5,10,1);
>> designmethods(d)


Design Methods for class fdesign.peak (N,F0,Q,Ap):


cheby1
```

`d = fdesign.peak(n,f0,q)` constructs a peaking filter specification object using the default `specstring ('N,F0,Q')` and setting the corresponding values to `n`, `f0`, and `q`.

By default, all frequency specifications are assumed to be in normalized frequency units. All magnitude specifications are assumed to be in decibels.

`d = fdesign.peak(...,Fs)` constructs a peak filter specification object while providing the sampling frequency of the signal to be filtered. `Fs` must be specified as a scalar trailing the other values provided. If you specify an `Fs`, it is assumed to be in Hz, as all the other frequency values provided.

`d = fdesign.peak(...,MAGUNITS)` constructs a notch filter specification while providing the units for any magnitude specification given. `MAGUNITS` can be one of the following: `'linear'`, `'dB'`, or `'squared'`. If this argument is omitted, `'dB'` is assumed. The magnitude specifications are always converted and stored in decibels regardless of how they were specified. If `Fs` is provided, `MAGUNITS` must follow `Fs` in the input argument list.

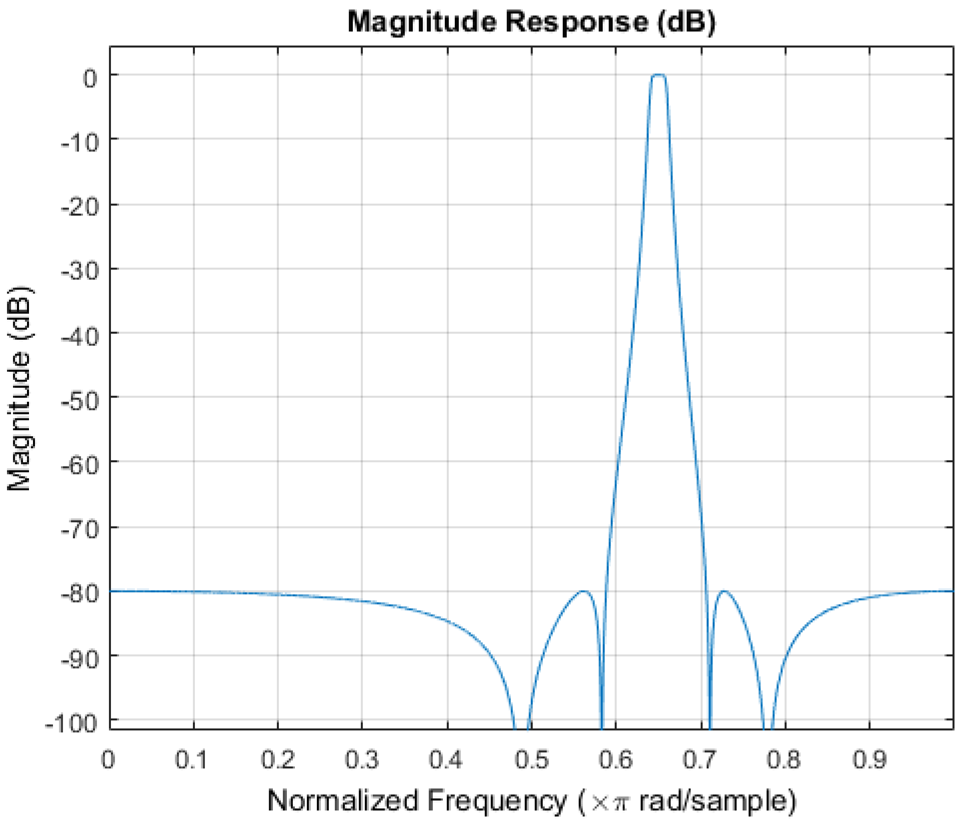
Examples

Design a Chebyshev Type II Peaking Filter

Design a Chebyshev Type II peaking filter with a stopband attenuation of 80 dB.

```
d = fdesign.peak('N,F0,BW,Ast',8,.65,.02,80);
Hd = design(d,'cheby2','SystemObject',true);
```

```
fvtool(Hd)
```



See Also

`fdesign` | `fdesign.notch` | `fdesign.parameq`

Introduced in R2011a

fdesign.polysrc

Construct polynomial sample-rate converter (POLYSRC) filter designer

Syntax

```
d = fdesign.polysrc(l,m)
d = fdesign.polysrc(l,m,'Fractional Delay','Np',Np)
d = fdesign.polysrc(...,Fs)
```

Description

`d = fdesign.polysrc(l,m)` constructs a polynomial sample-rate converter filter designer D with an interpolation factor L and a decimation factor M. L defaults to 3. M defaults to 2. L and M can be arbitrary positive numbers.

`d = fdesign.polysrc(l,m,'Fractional Delay','Np',Np)` initializes the filter designer specification with Np and sets the polynomial order to the value Np. If omitted Np defaults to 3.

`d = fdesign.polysrc(...,Fs)` specifies the sampling frequency (in Hz).

Examples

Design a Sample-Rate Converter

This example shows how to design sample-rate converter that uses a 3rd order Lagrange interpolation filter to convert from 44.1kHz to 48kHz.

```
[L,M] = rat(48/44.1);
f = fdesign.polysrc(L,M,'Fractional Delay','Np',3);
Hm = design(f,'lagrange');
```

Original sampling frequency

```
Fs = 44.1e3;
```

9408 samples, 0.213 seconds long

```
n = 0:9407;
```

Original signal, sinusoid at 1kHz

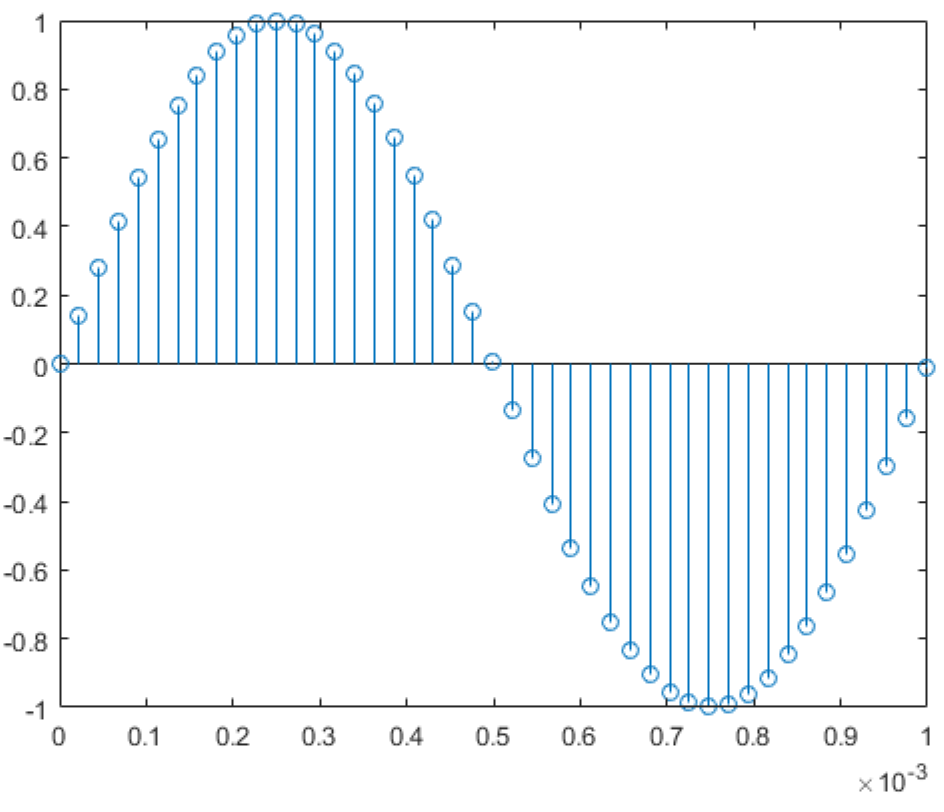
```
x  = sin(2*pi*1e3/Fs*n);
```

10241 samples, still 0.213 seconds

```
y = filter(Hm,x);
```

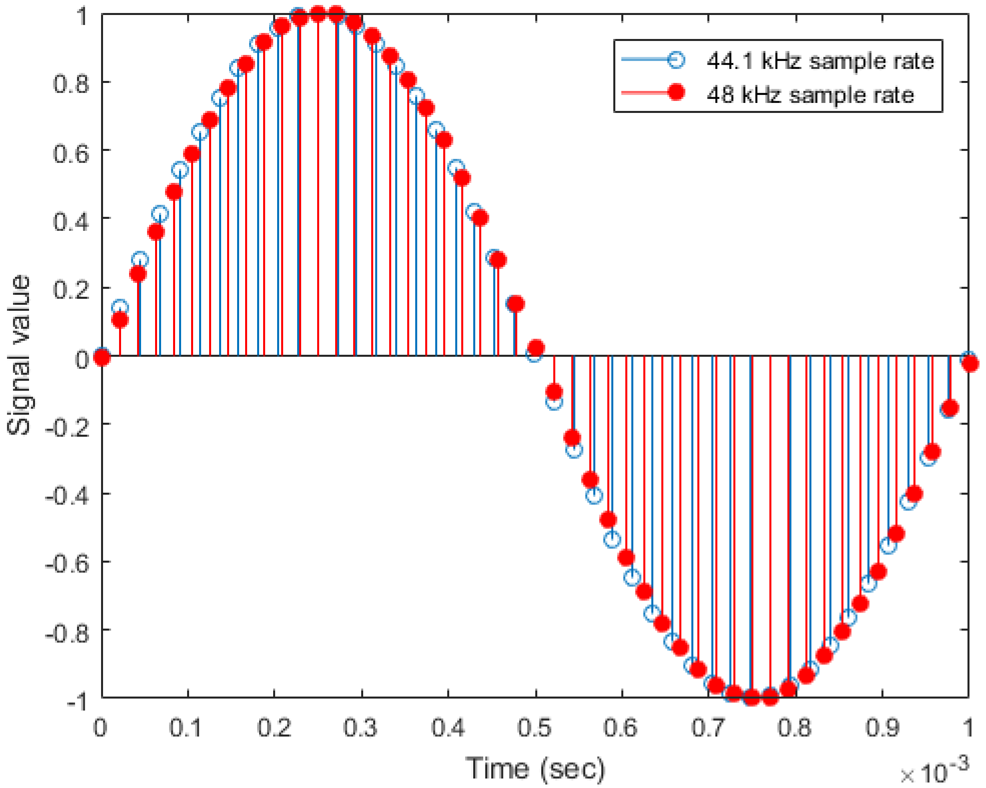
Plot original sampled at 44.1kHz

```
stem(n(1:45)/Fs,x(1:45))
hold on
```



Plot fractionally interpolated signal (48kHz) in red

```
stem((n(3:51)-2)/(Fs*L/M),y(3:51),'r','filled')
xlabel('Time (sec)');ylabel('Signal value')
legend('44.1 kHz sample rate','48 kHz sample rate')
```



For more information about Farrow SRCs, see the "Efficient Sample Rate Conversion between Arbitrary Factors" example, `efficientsrcdemo`.

See Also

`fdesign`

Introduced in R2011a

fdesign.pulseshaping

Pulse-shaping filter specification object

Syntax

```
D = fdesign.pulseshaping
D = fdesign.pulseshaping(sps)
D = fdesign.pulseshaping(sps,shape)
d = fdesign.pulseshaping(sps,shape,spec,value1,value2,...)
d = fdesign.pulseshaping(...,fs)
d = fdesign.pulseshaping(...,magunits)
```

Description

Note: The use of `fdesign.pulseshaping` is not recommended. Use `rcosdesign` or `gaussdesign` instead.

`D = fdesign.pulseshaping` constructs a specification object `D`, which can be used to design a minimum-order raised cosine filter object with a default stop band attenuation of 60dB and a rolloff factor of 0.25.

`D = fdesign.pulseshaping(sps)` constructs a minimum-order raised cosine filter specification object `d` with a positive integer-valued oversampling factor, `SamplesPerSymbol` .

`D = fdesign.pulseshaping(sps,shape)` constructs `d` where `shape` specifies the `PulseShape` property. Valid entries for `shape` are:

- `'Raised Cosine'`
- `'Square Root Raised Cosine'`
- `'Gaussian'`

`d = fdesign.pulseshaping(sps,shape,spec,value1,value2,...)` constructs `d` where `spec` defines the `Specification` properties. The entries for `spec` specify various properties of the filter, including the order and frequency response. Valid entries for `spec` depend upon the `shape` property. For `'Raised Cosine'` and `'Square Root Raised Cosine'` filters, the valid entries for `spec` are:

- `'Ast,Beta'` (minimum order; default)
- `'Nsym,Beta'`
- `'N,Beta'`

The filter specifications are defined as follows:

- `Ast` —stopband attenuation (in dB). The default stopband attenuation for a raised cosine filter is 60 dB. The default stopband attenuation for a square root raised cosine filter is 30 dB. If `Ast` is specified, the minimum-order filter is returned.
- `Beta` —rolloff factor expressed as a real-valued scalar ranging from 0 to 1. Smaller rolloff factors result in steeper transitions between the passband and stopband of the filter.
- `Nsym` —filter order in symbols. The length of the impulse response is given by `Nsym*SamplesPerSymbol+1`. The product `Nsym*SamplesPerSymbol` must be even.
- `N` —filter order (must be even). The length of the impulse response is `N+1`.

If the `shape` property is specified as `'Gaussian'`, the valid entries for `spec` are:

- `'Nsym,BT'` (default)

The filter specifications are defined as follows:

- `Nsym`—filter order in symbols. `Nsym` defaults to 6. The length of the filter impulse response is `Nsym*SamplesPerSymbol+1`. The product `Nsym*SamplesPerSymbol` must be even.
- `BT` —the 3–dB bandwidth-symbol time product. `BT` is a positive real-valued scalar, which defaults to 0.3. Larger values of `BT` produce a narrower pulse width in time with poorer concentration of energy in the frequency domain.

`d = fdesign.pulseshaping(...,fs)` specifies the sampling frequency of the signal to be filtered. `fs` must be specified as a scalar trailing the other numerical values provided. For this case, `fs` is assumed to be in Hz and is used for analysis and visualization.

`d = fdesign.pulseshaping(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. Valid entries for `magunits` are:

- `linear` — specify the magnitude in linear units
- `dB` — specify the magnitude in dB (decibels)
- `squared` — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

After creating the specification object `d`, you can use the `design` function to create a filter object such as `h` in the following example:

```
d = fdesign.pulseshaping(8,'Raised Cosine','Nsym,Beta',6,0.25);
h = design(d);
```

Normally, the `Specification` property of the specification object also determines which design methods you can use when you create the filter object. Currently, regardless of the `Specification` property, the `design` function uses the `window` design method with all `fdesign.pulseshaping` specification objects. The `window` method creates an FIR filter with a windowed impulse response.

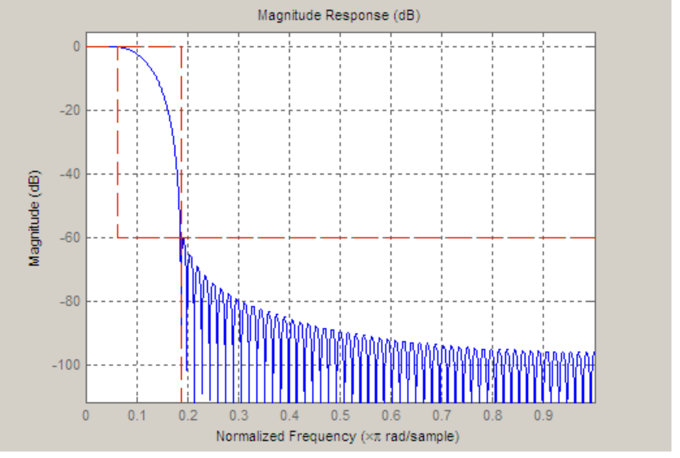
Examples

Pulse-shaping can be used to change the waveform of transmitted pulses so the signal bandwidth matches that of the communication channel. This helps to reduce distortion and intersymbol interference (ISI).

This example shows how to design a minimum-order raised cosine filter that provides a stop band attenuation of 60 dB, rolloff factor of 0.50, and 8 samples per symbol.

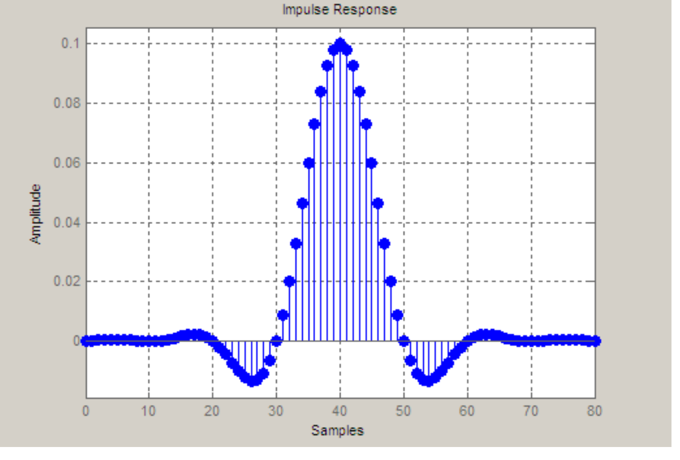
```
h  = fdesign.pulseshaping(8,'Raised Cosine','Ast,Beta',60,0.50);
    Hd = design(h);
    fvtool(Hd)
```

This code generates the following figure.



This example shows how to design a raised cosine filter that spans 8 symbol durations (i.e., of order 8 symbols), has a rolloff factor of 0.50, and oversampling factor of 10.

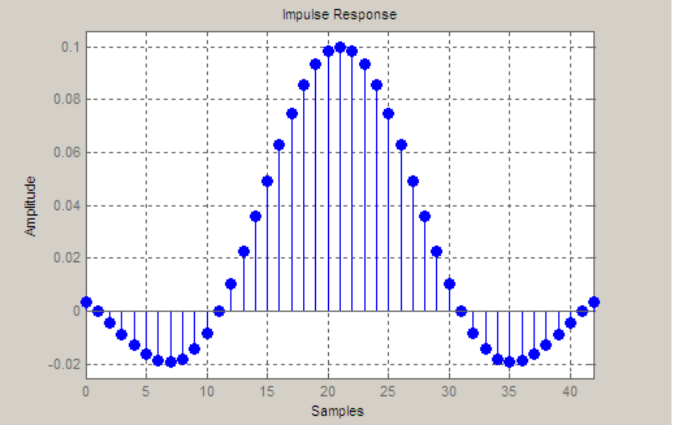
```
h  = fdesign.pulseshaping(10,'Raised Cosine','Nsym,Beta',8,0.50);
    Hd = design(h);
    fvtool(Hd, 'impulse')
```



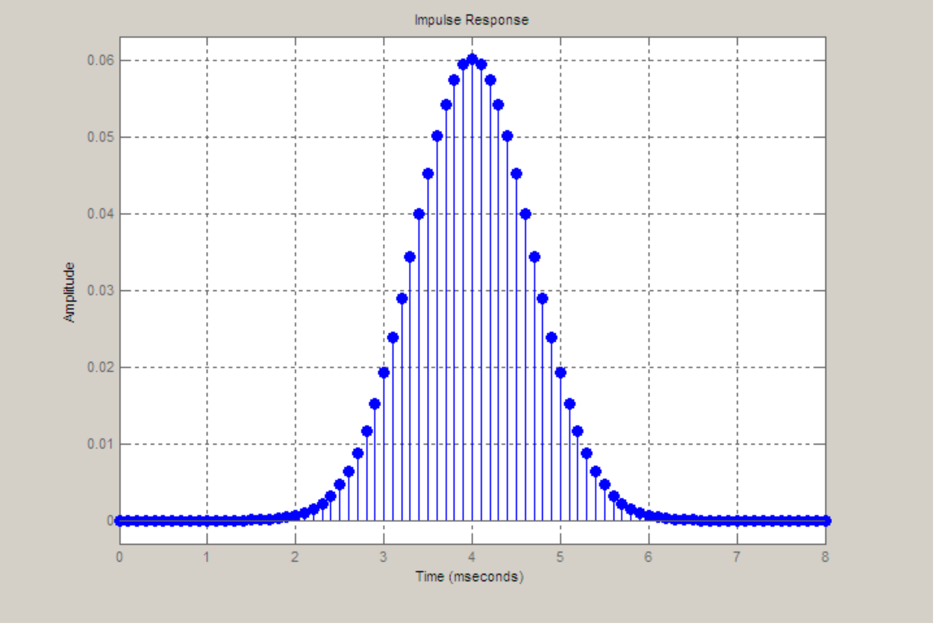
This example shows how to design a square root raised cosine filter of order 42, rolloff factor of 0.25, and 10 samples per symbol.

```
h  = fdesign.pulseshaping(10,'Square Root Raised Cosine','N,Beta',42);
```

```
Hd = design(h);
fvtool(Hd, 'impulse')
```



The following example demonstrates how to create a Gaussian pulse-shaping filter with an oversampling factor (sps) of 10, a bandwidth-time symbol product of 0.2, and 8 symbol periods. The sampling frequency is specified as 10 kHz.



Introduced in R2011a

fdesign.rsrc

Rational-factor sample-rate converter specification

Syntax

```
D = fdesign.rsrc(L,M)
D = fdesign.rsrc(L,M,RESPONSE)
D = fdesign.rsrc(L,M,CICRESPONSE,D)
D = fdesign.rsrc(L,M,RESPONSE,SPEC)
D = fdesign.rsrc(L,M,SPEC,specvalue1,specvalue2,...)
D = fdesign.rsrc(...,Fs)
D = fdesign.rsrc(...,MAGUNITS)
```

Description

`D = fdesign.rsrc(L,M)` constructs a rational-factor sample-rate filter specification object `D` with the `InterpolationFactor` property equal to the positive integer `L`, the `DecimationFactor` property equal to the positive integer `M` and the `Response` property set to `'Nyquist'`. The default values for the transition width and stopband attenuation in the Nyquist design are 0.1π radians/sample and 80 dB. If `L` is unspecified, `L` defaults to 3. If `M` is unspecified, `M` defaults to 2.

`D = fdesign.rsrc(L,M,RESPONSE)` constructs an rational-factor sample-rate converter with the interpolation factor `L`, decimation factor `M`, and the response you specify in `RESPONSE`.

`D = fdesign.rsrc(L,M,CICRESPONSE,D)` constructs a CIC or CIC compensator rational-factor sample-rate convertor filter specification object with the `'RESPONSE'` property equal to `'CIC'` or `'CICCOMP'`. `D` is the differential delay. The differential delay, `D`, must precede the filter specification.

Because you are designing multirate filters, the specification options available are not the same as the specification options for designing single-rate filters. The interpolation and decimation factors are not included in the specification. Different filter responses support different specifications. The following table lists the supported response types and specification options. These options are not case sensitive.

Design Method	Valid Specification Options
`'Arbitrary Magnitude'`	See `fdesign.arbmag` for a description of the specification entries. • `'N,F,A'` (default option) • `'N,B,F,A'`
`'Arbitrary Magnitude and Phase'`	See `fdesign.arbmagnphase` for a description of the specification entries. • `'N,F,H'` (default option) • `'N,B,F,H'`
`'Bandpass'`	See `fdesign.bandpass` for a description of the specification entries. • `'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'` (default option) • `'N,Fc1,Fc2'` • `'N,Fst1,Fp1,Fp2,Fst2'`
`'Bandstop'`	See `fdesign.bandstop` for a description of the specification entries. • `'N,Fc1,Fc2'` • `'N,Fp1,Fst1,Fst2,Fp2'` • `'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'` (default option)
`'CIC'`	`'Fp,Fst,Ap,Ast'` — Only valid specification. `Fp` is the passband frequency, `Fst` is the stopband frequency, `Ap` is the passband ripple, and `Ast` is the stopband attenuation in decibels. To specify a CIC rational-factor sample-rate convertor, include the differential delay after `'CIC'` and before the filter specification: `'Fp,Ast'`. For example: `d = fdesign.rsrc(2,2,'cic',4);`

<table>
<tr><th>Design Method</th><th>Valid Specification Options</th></tr>
<tr><td><code>'CIC Compensator'</code></td><td>See <code>fdesign.ciccomp</code> for a description of the specification entries.

• <code>'Fp,Fst,Ap,Ast'</code> (default option)
• <code>'N,Fc,Ap,Ast'</code>
• <code>'N,Fp,Ap,Ast'</code>
• <code>'N,Fp,Fst'</code>
• <code>'N,Fst,Ap,Ast'</code>

To specify a CIC compensator rational-factor sample-rate convertor, include the differential delay after <code>'CICCOMP'</code> and before the filter specification. For example:
<code>d = fdesign.rsrc(2,2,'ciccomp',4);</code></td></tr>
<tr><td><code>'Differentiator'</code></td><td><code>'N'</code> — filter order</td></tr>
<tr><td><code>'Gaussian'</code></td><td><code>'Nsym,BT</code> — <code>Nsym</code> is the filter order in symbols and <code>BT</code> is the bandwidth-symbol time product.

The filter specification must be preceded by an integer-valued <code>SamplesPerSymbol</code>.</td></tr>
<tr><td><code>'Halfband</code></td><td>See <code>fdesign.halfband</code> for a description of the specification entries.

• <code>'TW,Ast'</code> (default option)
• <code>'N,TW'</code>
• <code>‘N’</code>
• <code>'N,Ast'</code>

If you use the quasi-linear IIR design method, <code>iirlinphase</code>, with a halfband specification, the interpolation factor must be 2.</td></tr>
</table>

Design Method	Valid Specification Options
'Highpass'	See fdesign.highpass for a description of the specification entries. • 'Fst,Fp,Ast,Ap' (default option) • 'N,F3db' • 'N,Fc' • 'N,Fc,Ast,Ap' • 'N,Fp,Ast,Ap' • 'N,Fst,Ast,Ap' • 'N,Fst,Fp' • 'N,Fst,Ast,Ap' • 'N,Fst,Fp,Ast'
'Hilbert'	See fdesign.hilbert for a description of the specification entries. • 'N,TW' (default option) • 'TW,Ap'
'Inverse-sinc Lowpass'	See fdesign.isinclp for a description of the specification entries. • 'Fp,Fst,Ap,Ast' (default option) • 'N,Fc,Ap,Ast' • 'N,Fp,Fst' • 'N,Fst,Ap,Ast'
'Inverse-sinc Highpass'	See fdesign.isinchp for a description of the specification entries. • 'Fst,Fp,Ast,Ap' (default option) • 'N,Fc,Ast,Ap' • 'N,Fst,Fp' • 'N,Fst,Ast,Ap'

Design Method	Valid Specification Options
'Lowpass'	See fdesign.lowpass for a description of the specification entries. • 'Fp,Fst,Ap,Ast' (default option) • 'N,F3dB' • 'N,Fc' • 'N,Fc,Ap,Ast' • 'N,Fp,Ap,Ast' • 'N,Fp,Fst' • 'N,Fp,Fst,Ap' • 'N,Fp,Fst,Ast' • 'N,Fst,Ap,Ast'
'Nyquist'	See fdesign.nyquist for a description of the specification entries. For all Nyquist specifications, you must specify the *L*th band. This typically corresponds to the interpolation factor so that the nonzero samples of the upsampler output are preserved. • 'TW,Ast' (default option) • 'N' • 'N,Ast' • 'N,Ast'

D = fdesign.rsrc(L,M,RESPONSE,SPEC) constructs object D and sets its Specification property to SPEC. Entries in the SPEC represent various filter response features, such as the filter order, that govern the filter design. Valid entries for SPEC depend on the design type of the specifications object.

When you add the SPEC input argument, you must also add the RESPONSE input argument.

D = fdesign.rsrc(L,M,SPEC,specvalue1,specvalue2,...) constructs an object D and sets its specifications at construction time.

`D = fdesign.rsrc(...,Fs)` provides the sampling frequency of the signal to be filtered. `Fs` must be specified as a scalar trailing the other numerical values provided. `Fs` is assumed to be in Hz as are all other frequency values provided.

`D = fdesign.rsrc(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units.
- `'dB'` — specify the magnitude in dB (decibels).
- `'squared'` — specify the magnitude in power units.

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

Construct a Sample-Rate Converter Using `fdesign` Object

Design a rational-factor sample-rate converter. Set the rational sample-rate change to 5/3. Use the default Nyquist design with a transition width of 0.05? radians/sample and stopband attenuation of 40 dB. The Lth band factor in the Nyquist design is equal to the interpolation factor.

```
d = fdesign.rsrc(5,3,'nyquist',5,.05,40);
hm = design(d,'kaiserwin','SystemObject',true); %design with Kaiser window
```

Design a rational-factor sample-rate converter. Set the rational sample-rate change to 5/3. Use a Nyquist design with the filter specification set to 'N,TW'. Set the order equal to 12 and the transition width to 0.1? radians/sample. The Lth band factor in the Nyquist design is equal to the interpolation factor.

```
d = fdesign.rsrc(5,3,'nyquist',5,'N,TW',12,0.1);
```

Design a rational-factor sample-rate converter. Assume the data are sampled at 10 kHz. Set the rational sample-rate change to 3/2. Use a Nyquist design with the filter specification set to 'N,TW'. Set the order equal to 12 and the transition width to 100 Hz. The Lth band factor in the Nyquist design is equal to the interpolation factor.

```
d = fdesign.rsrc(3,2,'nyquist',3,'N,TW',12,100,1e4);
```

```
hd = design(d,'equiripple','SystemObject',true);
```

See Also

design | designmethods | fdesign.rsrc | fdesign.interpolator | setspecs | fdesign.arbmag | fdesign.arbmagnphase

Introduced in R2011a

fftcoeffs

Frequency-domain coefficients

Syntax

```
c = fftcoeffs(hd)
```

Description

`c = fftcoeffs(hd)` return the frequency-domain coefficients used when filtering with the `dfilt.fftfir` object. c contains the coefficients

Examples

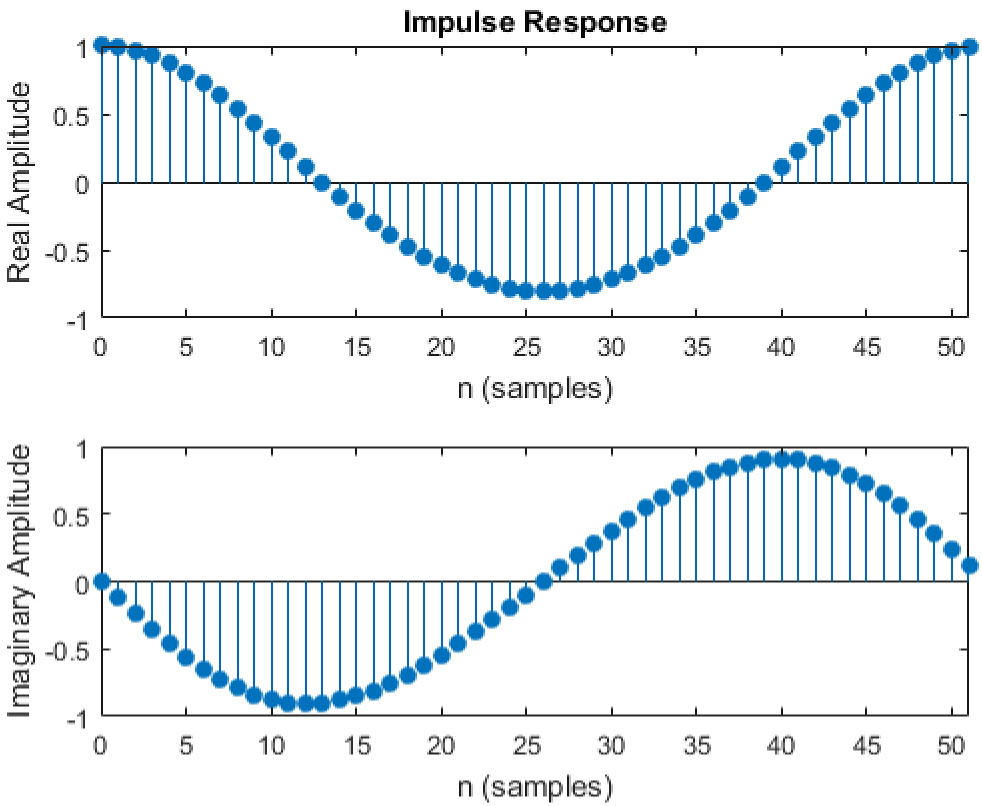
Frequency Domain Coefficients of FIR Filter

This example demonstrates returning the FFT coefficients from the discrete-time filter hd.

```
b = [0.05 0.9 0.05];
len = 50;
hd = dfilt.fftfir(b,len);
c=fftcoeffs(hd);
```

Plot the impulse response of the coefficients.

```
impz(c);
```



Introduced in R2011a

filter

Filter data with filter object

Syntax

Fixed-Point Filter Syntaxes

```
y = filter(hd,x)
y = filter(hd,x,dim)
```

Description

Fixed-Point Filter Syntaxes

`y = filter(hd,x)` filters a vector of real or complex input data `x` through a fixed-point filter `hd`, producing filtered output data `y`. The vectors `x` and `y` have the same length. filter stores the final conditions for the filter in the `States` property of `hd` — `hd.states`.

When you set the property `PersistentMemory` to `false` (the default setting), the initial conditions for the filter are set to zero before filtering starts. To use nonzero initial conditions for `hd`, set `PersistentMemory` to `true`. Then set `hd.states` to a vector of `nstates(hd)` elements, one element for each state to set. If you specify a scalar for `hd.states`, `filter` expands the scalar to a vector of the proper length for the states. All elements of the expanded vector have the value of the scalar.

If `x` is a matrix, `y = filter(hd,x)` filters along each column of `x` to produce a matrix `y` of independent channels. If `x` is a multidimensional array, `y = filter(hd,x)` filters `x` along the first nonsingleton dimension of `x`.

To use nonzero initial conditions when you are filtering a matrix `x`, set the filter states to a matrix of initial condition values. Set the initial conditions by setting the `States` property for the filter (`hd.states`) to a matrix of `nstates(hd)` rows and `size(x,2)` columns.

`y = filter(hd,x,dim)` applies the filter `hd` to the input data located along the specific dimension of `x` specified by `dim`.

When you are filtering multichannel data, `dim` lets you specify which dimension of the input matrix to filter along — whether a row represents a channel or a column represents a channel. When you provide the `dim` input argument, the filter operates along the dimension specified by `dim`. When your input data `x` is a vector or matrix and `dim` is 1, each column of `x` is treated as a one input channel. When `dim` is 2, the filter treats each row of the input `x` as a channel.

To filter multichannel data in a loop environment, you must use the `dim` input argument to set the proper processing dimension.

You specify the initial conditions for each channel individually, when needed, by setting `hm.states` to a matrix of `nstates(hd)` rows (one row containing the states for one channel of input data) and `size(x,2)` columns (one column containing the filter states for each channel).

Examples

Filter a Signal Using the `filter` function

Filter a signal using a filter with various initial conditions (IC) or no initial conditions.

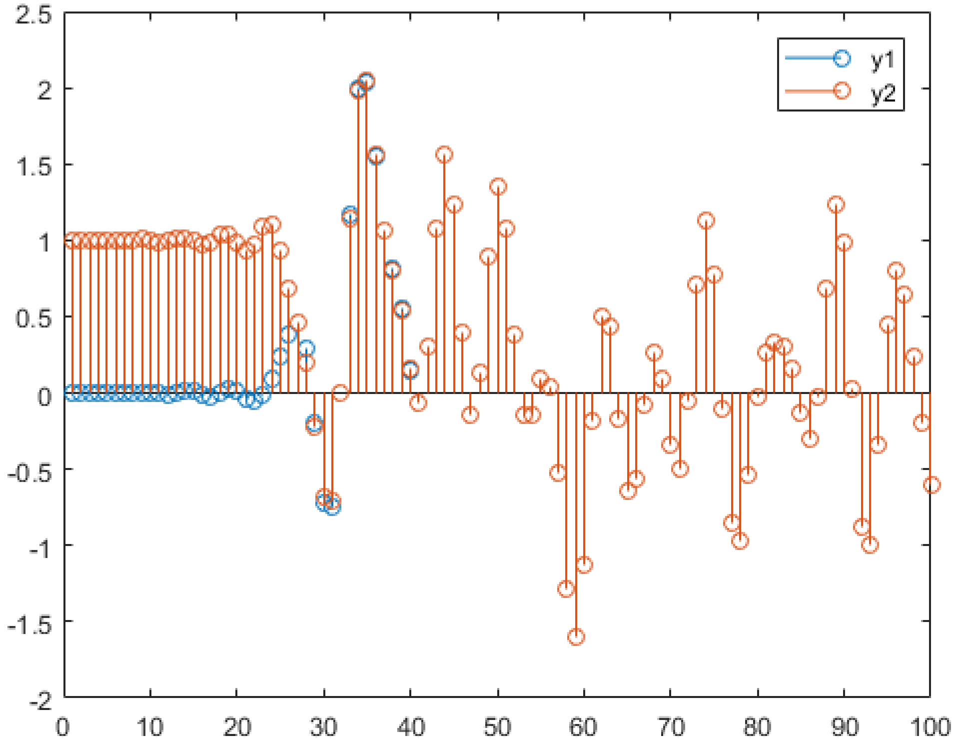
```
x = randn(100,1);    % Original signal.
b = fir1(50,.4);     % 50th-order linear-phase FIR filter.
hd = dfilt.dffir(b);    % Direct-form FIR implementation.
```

Do not set specific initial conditions.

```
y1 = filter(hd,x);   % 'PersistentMemory'='false'(default).
zf = hd.states;      % Final conditions.
```

Now use nonzero initial conditions by setting ICs after before you filter.

```
hd.persistentmemory = true;
hd.states = 1;       % Uses scalar expansion.
y2 = filter(hd,x);
stem([y1 y2])        % Different sequences at beginning.
legend('y1','y2');
```



Looking at the stem plot shows that the sequences are different at the beginning of the filter process.

Here is one way to use filter with streaming data.

```
reset(hd);            % Clear filter history.
y3 = filter(hd,x);    % Filter entire signal in one block.
```

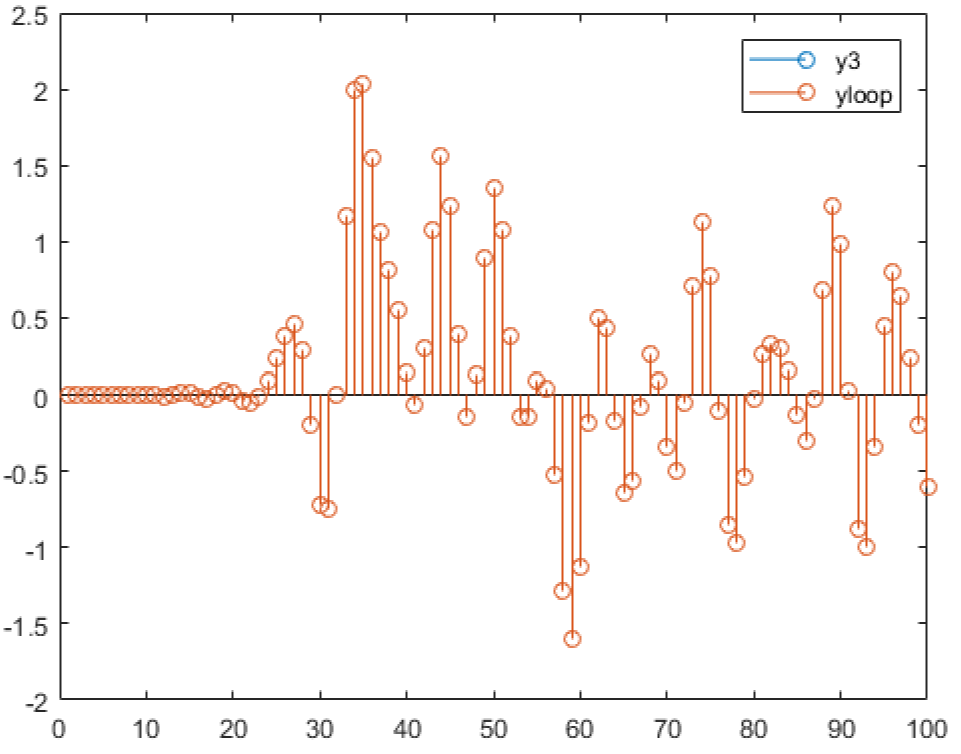
As an experiment, repeat the process, filtering the data as sections, rather than in streaming form.

```
reset(hd);               % Clear filter history.
yloop = zeros(20,5);     % Preallocate output array.
xblock = reshape(x,[20 5]);
```

```
for i=1:5
   yloop(:,i) = filter(hd,xblock(:,i));
end
```

Use a stem plot to see the comparison between streaming and block-by-block filtering.

```
stem([y3 yloop(:)]);
legend('y3','yloop')
```



Filtering the signal section-by-section is equivalent to filtering the entire signal at once.

Algorithms

Quantized Filters

The `filter` command implements fixed- or floating-point arithmetic on the quantized filter structure you specify.

The algorithm applied by `filter` when you use a discrete-time filter object on an input signal depends on the response you chose for the filter, such as lowpass or Nyquist or bandstop. To learn more about each filter algorithm, refer to the literature reference provided on the appropriate discrete-time filter reference page.

Note `dfilt/filter` does not normalize the filter coefficients automatically. Function `filter` supplied by MATLAB does normalize the coefficients.

Adaptive Filters

The algorithm used by `filter` when you apply an adaptive filter object to a signal depends on the algorithm you chose for your adaptive filter. To learn more about each adaptive filter algorithm, refer to the literature reference provided on the appropriate adaptive filter object reference page.

References

[1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

See Also

See Also

Functions
`impz` | `nstates`

Introduced in R2011a

filterBuilder

Interactive filter design

Syntax

```
filterBuilder(h)
filterBuilder('response')
```

Description

`filterBuilder` starts a interactive tool for building filters. It relies on the `fdesign` object-object oriented filter design paradigm, and is intended to reduce development time during the filter design process. `filterBuilder` uses a specification-centered approach to find the best algorithm for the desired response.

Note: You must have the Signal Processing Toolbox installed to use `fdesign` and `filterBuilder`. Some of the features described below may be unavailable if your installation does not additionally include the DSP System Toolbox. You can verify the presence of both toolboxes by typing `ver` at the command prompt.

For more information on how to use `filterBuilder`, see “Filter Builder Design Process”.

To use `filterBuilder`, enter `filterBuilder` at the MATLAB command line using one of three approaches:

- Simply enter `filterBuilder`. MATLAB opens a dialog for you to select a filter response type. After you select a filter response type, `filterBuilder` launches the appropriate filter design dialog box.
- Enter `filterBuilder(h)`, where h is an existing filter object. For example, if `h` is a bandpass filter, `filterBuilder(h)` opens the bandpass filter design dialog box. The `h` object must have been created using `filterBuilder` or using `fdesign`.

Note: You must have the DSP System Toolbox software to create and import filter System objects.

- Enter `filterBuilder('`*`response`*`')` to replace *`response`* with a response method from the following table. MATLAB opens a filter design dialog that corresponds to the specified response.

Note: You must have the DSP System Toolbox software to implement a number of the filter designs listed in the following table. If you only have the Signal Processing Toolbox software, you can design a limited set of the following filter-response types.

Response Method	Description of Resulting Filter Design	Filter Object
arbgrpdelay	Arbitrary group delay filter design	`fdesign.arbgrpdelay`
arbmag	Arbitrary magnitude filter design	`fdesign.arbmag`
arbmagnphase	Arbitrary response filter (magnitude and phase)	`fdesign.arbmagnphase`
audioweighting	Audio weighting filter	`fdesign.audioweighting`
bandpass or `bp`	Bandpass filter	`fdesign.bandpass`
bandstop or `bs`	Bandstop filter	`fdesign.bandstop`

cic	CIC filter	`fdesign.decimator(M,'cic',...)`
		or
		`fdesign.interpolator(L,'cic',...`
		See `fdesign.decimator`
		and
		`fdesign.interpolator`

ciccomp	CIC compensator	`fdesign.ciccomp`
comb	Comb filter	`fdesign.comb`
diff	Differentiator filter	`fdesign.differentiator`
fracdelay	Fractional delay filter	`fdesign.fracdelay`

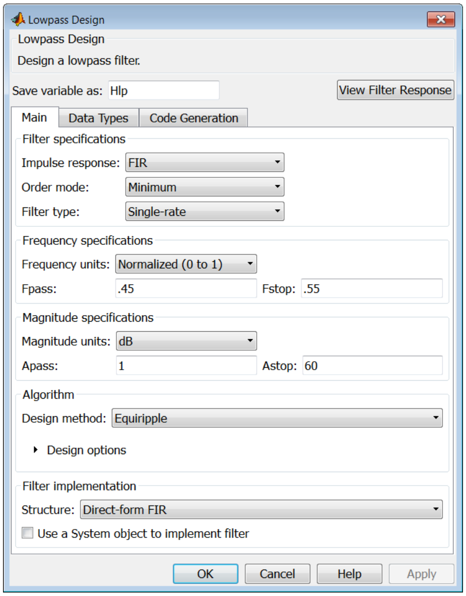
Response Method	Description of Resulting Filter Design	Filter Object
halfband or `hb`	Halfband filter	`fdesign.halfband`
highpass or `hp`	Highpass filter	`fdesign.highpass`
hilb	Hilbert filter	`fdesign.hilbert`
isinc, isinclp, or isinchp	Inverse sinc lowpass or highpass filter	`fdesign.isinclp` and `fdesign.isinchp`
lowpass or `lp`	Lowpass filter (default)	`fdesign.lowpass`
notch	Notch filter	`fdesign.notch`
nyquist	Nyquist filter	`fdesign.nyquist`
octave	Octave filter	`fdesign.octave`
parameq	Parametric equalizer filter	`fdesign.parameq`
peak	Peak filter	`fdesign.peak`

Note: Because they do not change the filter structure, the magnitude specifications and design method are tunable when using `filterBuilder`.

Filter Builder Design Panes

Main Design Pane

The main pane of Filter Builder varies depending on the filter response type, but the basic structure is the same. The following figure shows the basic layout of the dialog box.



As you choose the response for the filter, the available options and design parameters displayed in the dialog box change. This display allows you to focus only on parameters that make sense in the context of your filter design.

Every filter design dialog box includes the options displayed at the top of the dialog box, shown in the following figure.

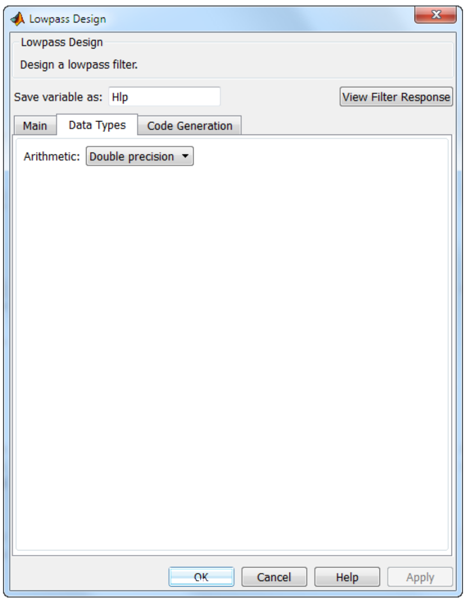
- **Save variable as** — When you click **Apply** to apply your changes or **OK** to close this dialog box, `filterBuilder` saves the current filter to your MATLAB workspace as a filter object with the name you enter.
- **View Filter Response** — Displays the magnitude response for the current filter specifications and design method by opening the Filter Visualization Tool (`fvtool`).

Note: The `filterBuilder` dialog box includes an **Apply** option. Each time you click **Apply**, `filterBuilder` writes the modified filter to your MATLAB workspace. This modified filter has the variable name you assign in **Save variable as**. To apply changes without overwriting the variable in you workspace, change the variable name in **Save variable as** before you click **Apply**.

There are three tabs in the Filter Builder dialog box, containing three panes: **Main**, **Data Types**, and **Code Generation**. The first pane changes according to the filter being designed. The last two panes are the same for all filters. These panes are discussed in the following sections.

Data Types Pane

The second tab in the Filter Builder dialog box is shown in the following figure.

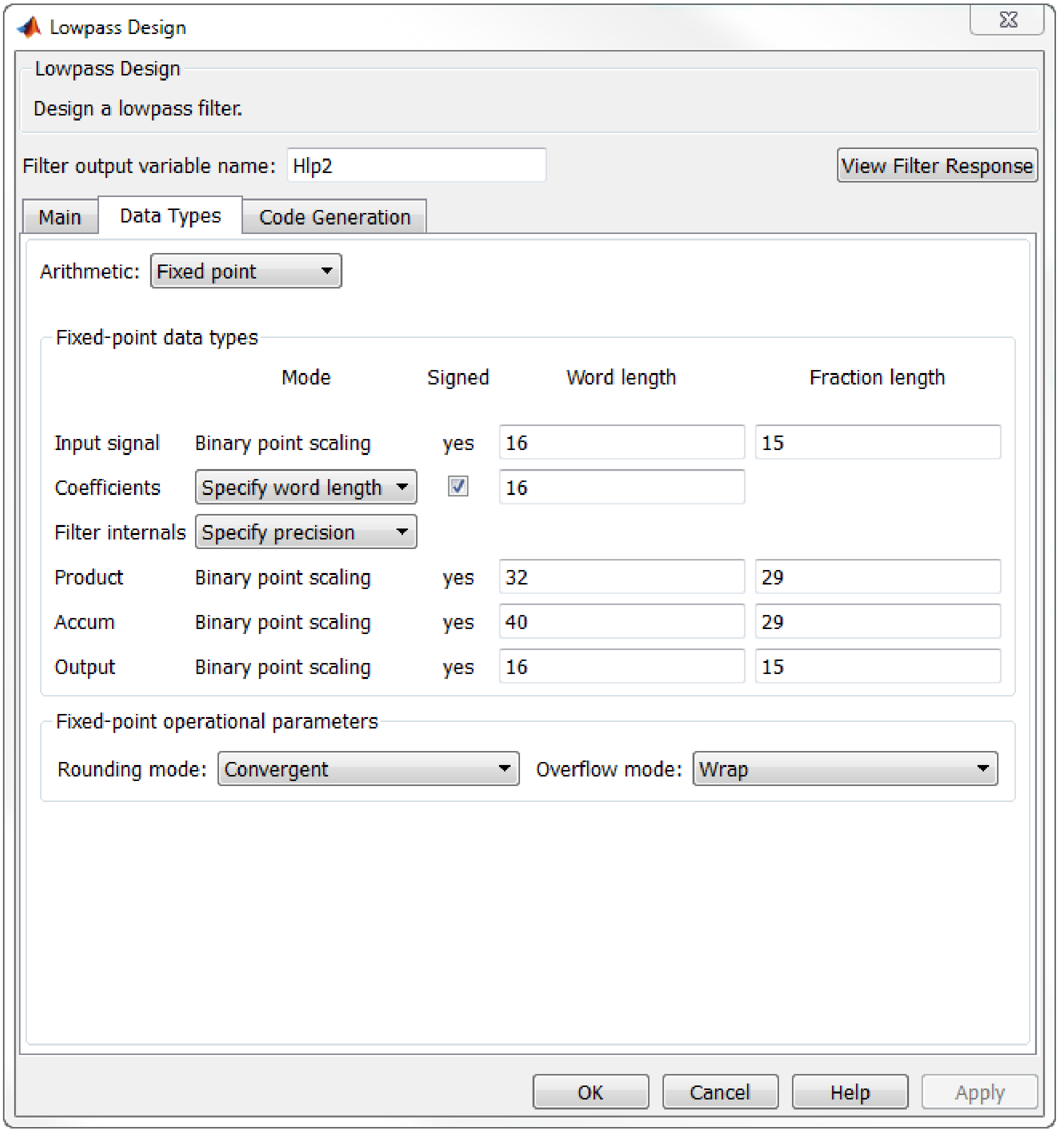


The **Arithmetic** drop down box allows the choice of `Double precision`, `Single precision`, or `Fixed point`. Some of these options may be unavailable depending on the filter parameters. The following table describes these options.

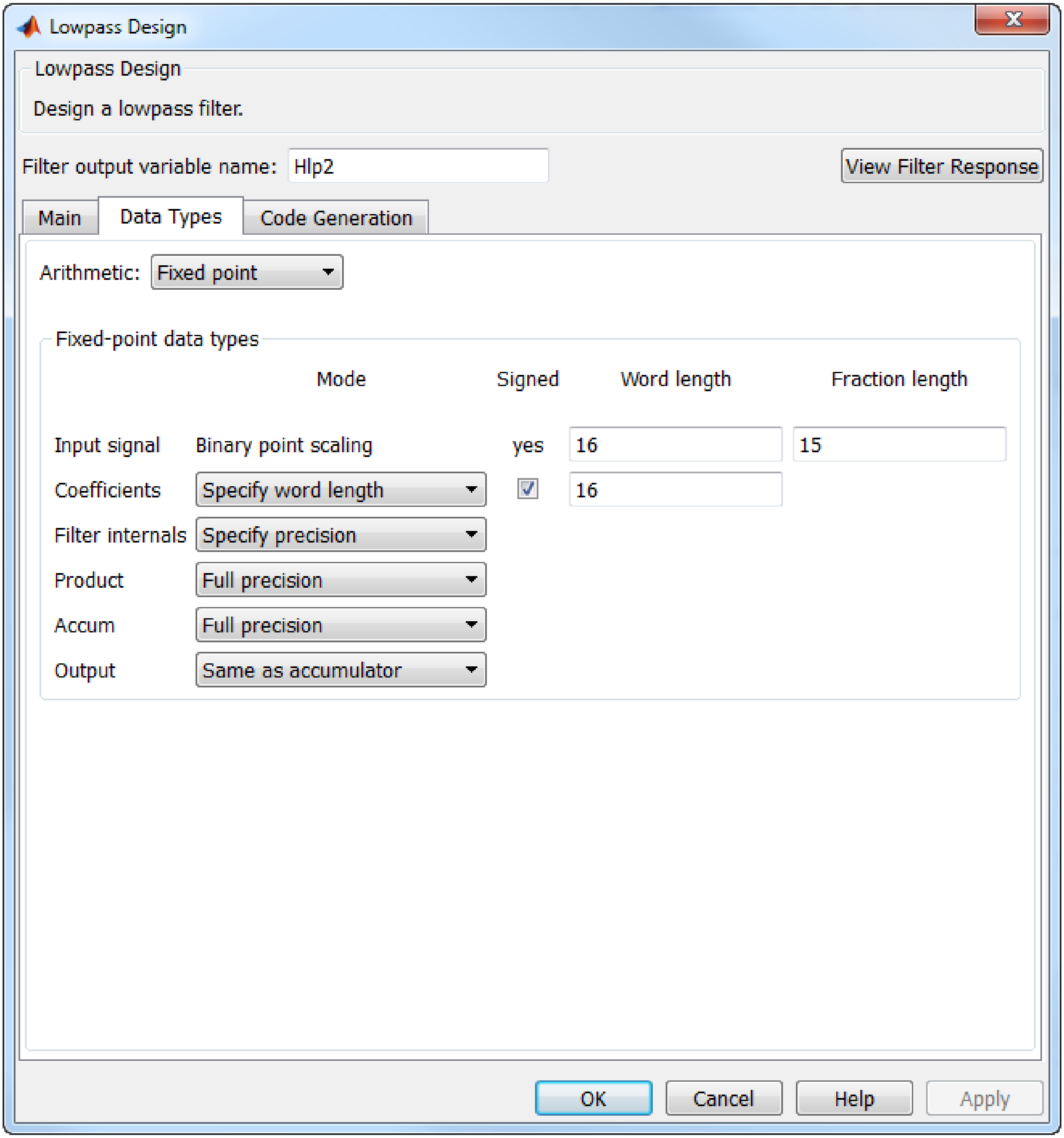
Arithmetic List Entry	Effect on the Filter
`Double precision`	All filtering operations and coefficients use double-precision, floating-point representations and math. When you use

Arithmetic List Entry	Effect on the Filter
	`filterBuilder` to create a filter, `double precision` is the default value for the Arithmetic property.
`Single precision`	All filtering operations and coefficients use single-precision floating-point representations and math.
`Fixed point`	This entry applies selected default values, typically used on many digital processors, for the properties in the fixed-point filter. These properties include coefficient word lengths, fraction lengths, and various operating modes. This setting allows signed fixed data types only. Fixed-point filter design with `filterBuilder` is available only when you install Fixed-Point Designer software along with DSP System Toolbox software.

The following figure shows the **Data Types** pane after you select `Fixed point` for **Arithmetic** and set **Filter internals** to `Specify precision`. This figure shows the **Data Types** pane for the case where the **Use a System object to implement filter** check box is not selected in the **Main** pane.



When you select **Use a System object to implement filter** check box in the **Main** pane, the **Data Types** pane appears as below:


Lowpass Design
Lowpass Design
Design a lowpass filter.
Filter output variable name: Hlp2
View Filter Response
Main
Data Types
Code Generation
Arithmetic: Fixed point
Fixed-point data types
Mode
Signed
Word length
Fraction length
Input signal
Binary point scaling
yes
16
15
Coefficients
Specify word length
16
Filter internals
Specify precision
Product
Full precision
Accum
Full precision
Output
Same as accumulator
OK
Cancel
Help
Apply

Not all parameters described in the following section apply to all filters. For example, FIR filters do not have the **Section input** and **Section output** parameters.

Input signal

Specify the format the filter applies to data to be filtered. For all cases, `filterBuilder` implements filters that use binary point scaling and signed input. You set the word length and fraction length as needed.

Coefficients

Choose how you specify the word length and the fraction length of the filter numerator and denominator coefficients:

- `Specify word length` enables you to enter the word length of the coefficients in bits. In this mode, `filterBuilder` automatically sets the fraction length of the coefficients to the binary-point only scaling that provides the best possible precision for the value and word length of the coefficients.
- `Binary point scaling` enables you to enter the word length and the fraction length of the coefficients in bits. If applicable, enter separate fraction lengths for the numerator and denominator coefficients.
- The filter coefficients do not obey the **Rounding mode** and **Overflow mode** parameters that are available when you select `Specify precision` from the Filter internals list. Coefficients are always saturated and rounded to `Nearest`.

Section Input

Choose how you specify the word length and the fraction length of the fixed-point data type going into each section of an SOS filter. This parameter is visible only when the selected filter structure is IIR and SOS.

- `Binary point scaling` enables you to enter the word and fraction lengths of the section input in bits.
- `Specify word length` enables you to enter the word lengths in bits.

Section Output

Choose how you specify the word length and the fraction length of the fixed-point data type coming out of each section of an SOS filter. This parameter is visible only when the selected filter structure is IIR and SOS.

- `Binary point scaling` enables you to enter the word and fraction lengths of the section output in bits.
- `Specify word length` enables you to enter the output word lengths in bits.

State

Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Use this parameter to specify how to designate the state word and fraction lengths. This parameter is not visible for direct form and direct form I filter structures because `filterBuilder` deduces the state directly from the input format. States always use signed representation:

- `Binary point scaling` enables you to enter the word length and the fraction length of the accumulator in bits.
- `Specify precision` enables you to enter the word length and fraction length in bits (if available).

Product

Determines how the filter handles the output of product operations. Choose from the following options:

- `Full precision` — Maintain full precision in the result.
- `Keep LSB` — Keep the least significant bit in the result when you need to shorten the data words.
- `Specify Precision` — Enables you to set the precision (the fraction length) used by the output from the multiplies.

Filter internals

Specify how the fixed-point filter performs arithmetic operations within the filter. The affected filter portions are filter products, sums, states, and output. Select one of these options:

- `Full precision` — Specifies that the filter maintains full precision in all calculations for products, output, and in the accumulator.
- `Specify precision` — Set the word and fraction lengths applied to the results of product operations, the filter output, and the accumulator. Selecting this option enables the word and fraction length controls.

Signed

Selecting this option directs the filter to use signed representations for the filter coefficients.

Word length

Sets the word length for the associated filter parameter in bits.

Fraction length

Sets the fraction length for the associate filter parameter in bits.

Accum

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths.

Determines how the accumulator outputs stored values. Choose from the following options:

- `Full precision` — Maintain full precision in the accumulator.
- `Keep MSB` — Keep the most significant bit in the accumulator.
- `Keep LSB` — Keep the least significant bit in the accumulator when you need to shorten the data words.
- `Specify Precision` — Enables you to set the precision (the fraction length) used by the accumulator.

Output

Sets the mode the filter uses to scale the output data after filtering. You have the following choices:

- `Avoid Overflow` — Set the output data fraction length to avoid causing the data to overflow. `Avoid overflow` is considered the conservative setting because it is independent of the input data values and range.
- `Best Precision` — Set the output data fraction length to maximize the precision in the output data.
- `Specify Precision` — Set the fraction length used by the filtered data.

Fixed-point operational parameters

Parameters in this group control how the filter rounds fixed-point values and how it treats values that overflow.

Rounding mode

Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).

- `ceil` — Round toward positive infinity.
- `convergent` — Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.

- `zero/fix` — Round toward zero.
- `floor` — Round toward negative infinity.
- `nearest` — Round toward nearest. Ties round toward positive infinity.
- `round` — Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

The choice you make affects everything except coefficient values and input data which always round. In most cases, products do not overflow—they maintain full precision.

Overflow mode

Sets the mode the filter uses to respond to overflow conditions in fixed-point arithmetic. Choose from the following options:

- `Saturate` — Limit the output to the largest positive or negative representable value.
- `Wrap` — Set overflowing values to the nearest representable value using modular arithmetic.

The choice you make affects everything except coefficient values and input data which always round. In most cases, products do not overflow—they maintain full precision.

Cast before sum

Specifies whether to cast numeric data to the appropriate accumulator format before performing sum operations. Selecting **Cast before sum** ensures that the results of the affected sum operations match most closely the results found on most digital signal processors. Performing the cast operation before the summation adds one or two additional quantization operations that can add error sources to your filter results.

If you clear **Cast before sum**, the filter prevents the addends from being cast to the sum format before the addition operation. Choose this setting to get the most accurate results from summations without considering the hardware your filter might use. The input format referenced by **Cast before sum** depends on the filter structure you are using.

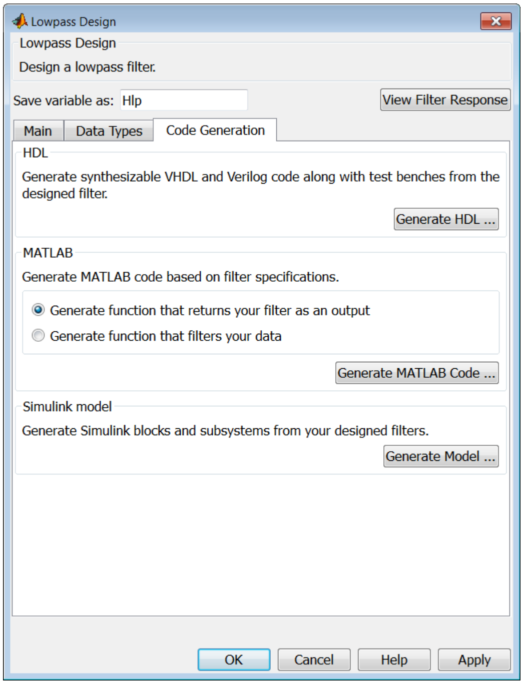
The effect of clearing or selecting **Cast before sum** is as follows:

- Cleared — Configures filter summation operations to retain the addends in the format carried from the previous operation.

- Selected — Configures filter summation operations to convert the input format of the addends to match the summation output format before performing the summation operation. Usually, selecting **Cast before sum** generates results from the summation that more closely match those found from digital signal processors.

Code Generation Pane

The code generation pane contains options for various implementations of the completed filter design. Depending on your installation, you can generate MATLAB, VHDL, and Verilog code from the designed filter. You can also choose to create or update a Simulink model from the designed filter. The following section explains these options.



HDL

For more information on this option, see “Opening the Filter Design HDL Coder GUI from the Filter Builder” (Filter Design HDL Coder).

MATLAB

Generate MATLAB code based on filter specifications

- **Generate function that returns your filter as an output**

  Selecting this option generates a function that designs a filter object using `fdesign`.

- **Generate function that filters your data**

  Selecting this option generates a function that takes data as input, and outputs data filtered with the designed filter. The data type of the filter output is set according to the data type settings in the **Data Types** pane.

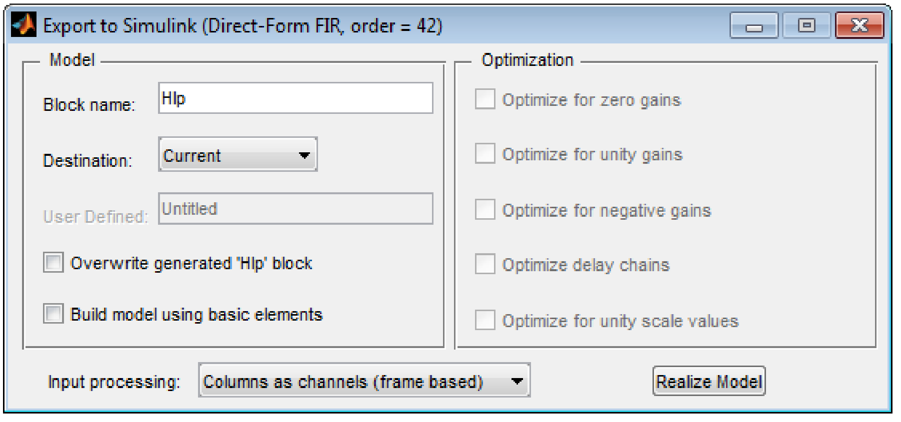
Clicking on the **Generate MATLAB code** button, brings up a Save File dialog. Specify the file name and location, and save. The filter is now contained in an editable file.

Simulink Model

Generate Simulink blocks and subsystems from your designed filters

When you click **Generate Model**, the filter builder generates Simulink blocks and subsystems from your designed filters.

Clicking on the **Generate Model** button opens the Export to Simulink dialog box.



- **Block Name** — The name for the new subsystem block, set to **Filter** by default.
- **Destination** — `Current` saves the generated model to the current Simulink model. `New` creates a new model to contain the generated block. `User Defined` creates a new model or subsystem at the location specified in `User Defined`.

- **Overwrite generated 'Filter' block** — Overwrites an existing block with the name specified in **Block Name**. Clear this check box to create a new block with the same name.
- **Build model using basic elements** — Builds the model using only basic blocks.
- **Optimize for zero gains** — Removes all zero-gain blocks from the model.
- **Optimize for unity gains** — Replaces all unity gains with direct connections.
- **Optimize for negative gains** — Removes all negative unity gain blocks, and changes sign at the nearest summation block.
- **Optimize delay chains** — Replaces delay chains made up of *n* unit delays with a single delay by *n*.
- **Optimize for unity scale values** — Removes all scale value multiplications by 1 from the filter structure.
- **Input processing** — Specify how the generated filter block or subsystem block processes the input. Depending on the type of filter you are designing, one or both of the following options may be available:

  - `Columns as channels (frame based)` — The block treats each column of the input as a separate channel.
  - `Elements as channels (sample based)` — The block treats each element of the input as a separate channel.

  For more information about sample-based and frame-based processing, see "Sample- and Frame-Based Concepts".
- **Realize Model** — Builds the model with the set parameters.

When the **Use a System object to implement filter** check box is selected in the **Main** pane, the **Generate Model** button in the **Simulink model** panel is disabled under the following conditions:

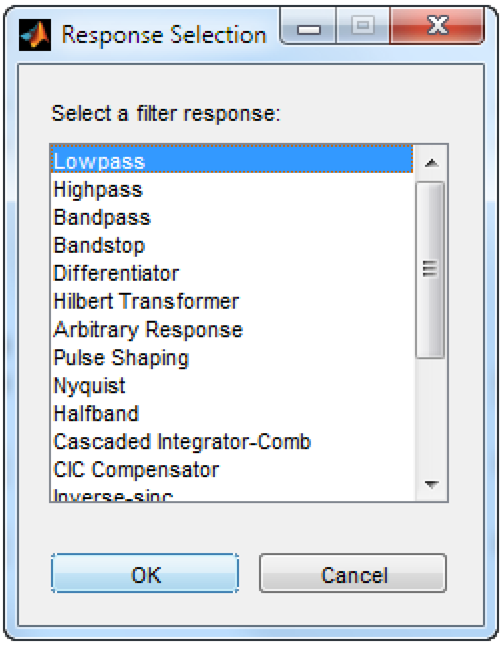
- Select **Filter response** as `Comb` and **Arithmetic** on the **Data Types** pane as `Fixed point`.
- Select **Filter response** as `Arbitrary Response`, **Impulse response** as `IIR`, set **Specify response as** to either `Magnitudes and phases` or `Frequency response`, and **Arithmetic** on the **Data Types** pane as `Fixed point`.

These settings design a `dsp.IIRFilter` System object with fixed point arithmetic. Generating a Simulink model for fixed point `dsp.IIRFilter` object is not supported.

Filter Responses

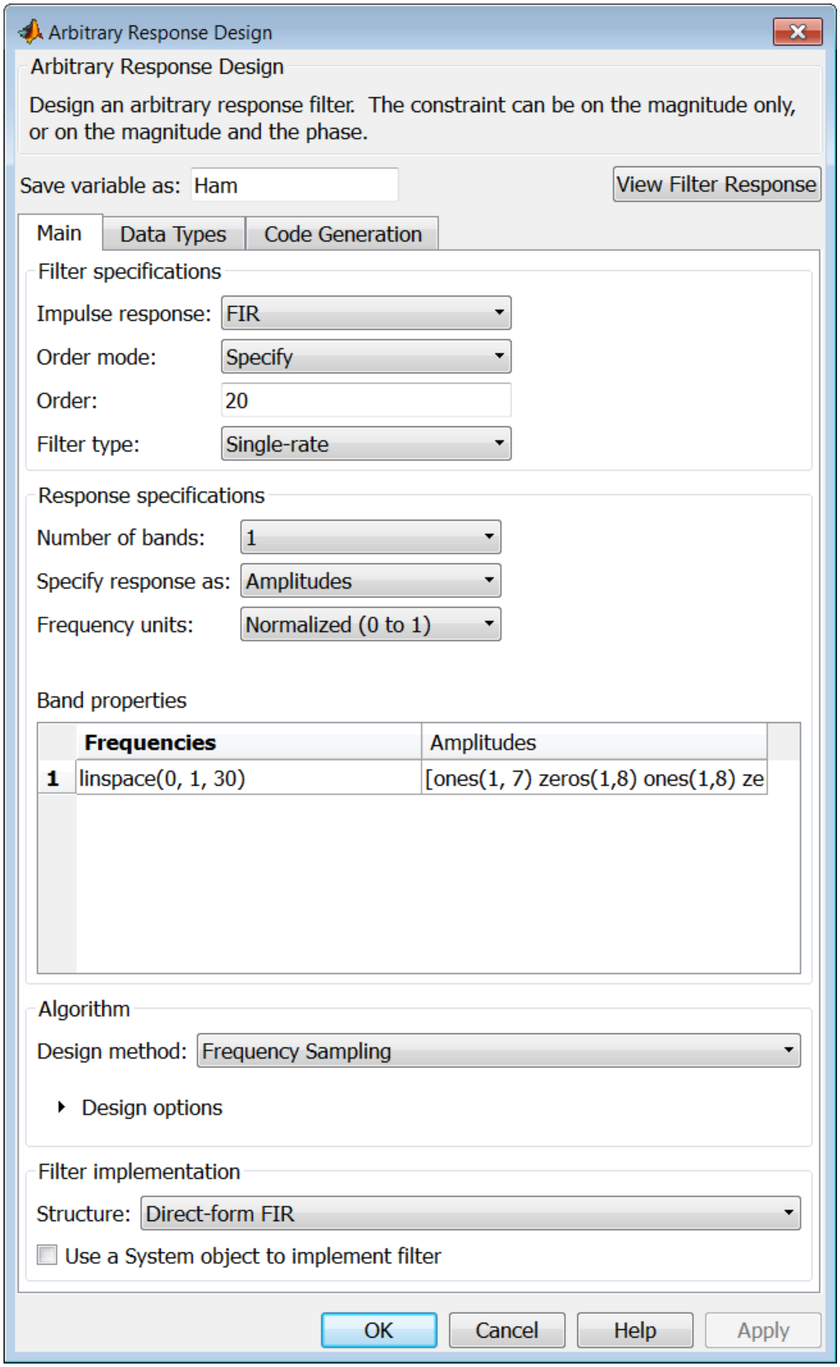
Select your filter response from the `filterBuilder` **Response Selection** main menu.

If you have the DSP System Toolbox software, the following **Response Selection** menu appears.



Select your desired filter response from the menu and design your filter.

The following sections describe the options available for each response type.


Arbitrary Response Design
Arbitrary Response Design
Design an arbitrary response filter. The constraint can be on the magnitude only, or on the magnitude and the phase.
Save variable as: Ham
View Filter Response
Main
Data Types
Code Generation
Filter specifications
Impulse response: FIR
Order mode: Specify
Order: 20
Filter type: Single-rate
Response specifications
Number of bands: 1
Specify response as: Amplitudes
Frequency units: Normalized (0 to 1)
Band properties
Frequencies
Amplitudes
1
linspace(0, 1, 30)
[ones(1, 7) zeros(1,8) ones(1,8) ze
Algorithm
Design method: Frequency Sampling
Design options
Filter implementation
Structure: Direct-form FIR
Use a System object to implement filter
OK
Cancel
Help
Apply

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

This dialog only applies if you have the DSP System Toolbox software. Select either `FIR` or `IIR` from the drop down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly. Arbitrary group delay designs are only available if **Impulse response** is `IIR`. Without the DSP System Toolbox, the only available arbitrary response filter design is FIR.

Order mode

This dialog only applies if you have the DSP System Toolbox software. Choose `Minimum` or `Specify`. Choosing `Specify` enables the **Order** dialog.

Order

This dialog only applies when **Order mode** is `Specify`. For an FIR design, specify the filter order. For an IIR design, you can specify an equal order for the numerator and denominator, or you can specify different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box. Because the Signal Processing Toolbox only supports FIR arbitrary-magnitude filters, you do not have the option to specify a denominator order.

Denominator order

Select the check box and enter the denominator order. This option is enabled only if `IIR` is selected for **Impulse response**.

Filter type

This dialog only applies if you have the DSP System Toolbox software and is only available for FIR filters. Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2 for `Decimator` and 3 for `Sample-rate converter`.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Response Specification

Number of Bands

Select the number of bands in the filter. Multiband design is available for both FIR and IIR filters.

Specify response as:

Specify the response as `Amplitudes`, `Magnitudes and phase`, `Frequency response`, or `Group delay`. `Amplitudes` is the only option if you do not have the DSP System Toolbox software. `Group delay` is only available for IIR designs.

Frequency units

Specify frequency units as either `Normalized`, `Hz`, `kHz`, `MHz`, or `GHz`.

Input Fs

Enter the input sampling frequency in the units specified in the **Frequency units** drop-down box. This option is enabled when **Frequency units** is set to an option in hertz.

Band Properties

These properties are modified automatically depending on the response chosen in the **Specify response as** drop-down box. Two or three columns are presented for input. The first column is always Frequencies. The other columns are either Amplitudes, Magnitudes, Phases, or Frequency Response. Enter the corresponding vectors of values for each column.

- **Frequencies** and **Amplitudes** — These columns are presented for input if you select `Amplitudes` in the **Specify response as** drop-down box.

- **Frequencies**, **Magnitudes**, and **Phases** — These columns are presented for input if the response chosen in the **Specify response as** drop-down box is `Magnitudes and phases`.
- **Frequencies** and **Frequency response** — These columns are presented for input if the response chosen in the **Specify response as** drop-down box is `Frequency response`.

Algorithm

The options for each design are specific for each design method. In the arbitrary response design, the available options also depend on the **Response specifications**. This section does not present all of the available options for all designs and design methods.

Design Method

Select the design method for the filter. Different methods are enabled depending on the defining parameters entered in the previous sections.

Design Options

- **Window** — Valid when the **Design method** is `Frequency Sampling`. Replace the square brackets with the name of a `window` function or function handle. For example, `'hamming'` or `@hamming`. If the window function takes parameters other than the length, use a cell array. For example, `{‘kaiser’,3.5}` or `{@chebwin,60}`.
- **Density factor** — Valid when the **Design method** is `equiripple`. Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

  Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

  The default changes to 20 for an IIR arbitrary group delay design.
- **Phase constraint** — Valid when the **Design method** is `equiripple`, you have the DSP System Toolbox installed, and **Specify response as** is set to `Amplitudes`. Choose one of `Linear`, `Minimum`, or `Maximum`.
- **Weights** — Uses the weights in **Weights** to weight the error for a single-band design. If you have multiple frequency bands, the **Weights** design option changes

to **B1 Weights**, **B2 Weights** to designate the separate bands. Use **Bi Weights** to specify weights for the i-th band. The **Bi Weights** design option is only available when you specify the i-th band as an unconstrained.

- **Bi forced frequency point** — This option is only available in a multi-band constrained equiripple design when **Specify response as** is `Amplitudes`. **Bi forced frequency point** is the frequency point in the i-th band at which the response is forced to be zero. The index **i** corresponds to the frequency bands in **Band properties**. For example, if you specify two bands in **Band properties**, you have **B1 forced frequency point** and **B2 forced frequency point**.
- **Norm** — Valid only for IIR arbitrary group delay designs. **Norm** is the norm used in the optimization. The default value is 128, which essentially equals the L-infinity norm. The norm must be even.
- **Max pole radius** — Valid only for IIR arbitrary group delay designs. Constrains the maximum pole radius. The default is 0.999999. Reducing the **Max pole radius** can produce a transfer function more resistant to quantization.
- **Init norm** — Valid only for IIR arbitrary group delay designs. The initial norm used in the optimization. The default initial norm is 2.
- **Init numerator** — Specifies an initial estimate of the filter numerator coefficients.
- **Init denominator** — Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems. In allpass filters, you only have to specify either the denominator or numerator coefficients. If you specify the denominator coefficients, you can obtain the numerator coefficients.

Filter implementation

Structure

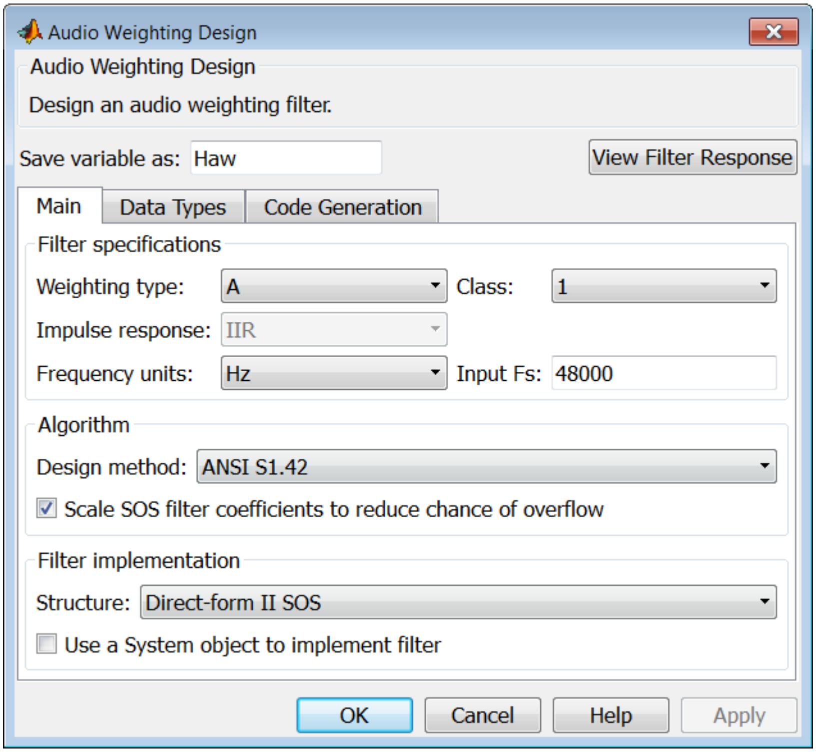
Select the structure for the filter. The available filter structures depend on the parameters you select for your filter.

Use a System object to implement filter

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, this check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Audio Weighting Filter Design — Main Pane



Filter specifications

- **Weighting type** — The weighting type defines the frequency response of the filter. The valid weighting types are: A, C , C-message, ITU-T 0.41, and ITU-R 468–4 weighting. See `fdesign.audioweighting` for definitions of the weighting types.
- **Class** — Filter class is only applicable for A weighting and C weighting filters. The filter class describes the frequency-dependent tolerances specified in the relevant standards. There are two possible class values: 1 and 2. Class 1 weighting filters have stricter tolerances than class 2 filters. The filter class value does not affect the design. The class value is only used to provide a specification mask in `fvtool` for the analysis of the filter design.

- **Impulse response** — Impulse response type as one of `IIR` or `FIR`. For A, C , C-message, and ITU-R 468–4 filter, IIR is the only option. For a ITU-T 0.41 weighting filter, FIR is the only option.
- **Frequency units** — Choose `Hz`, `kHz`, `MHz`, or `GHz`. Normalized frequency designs are not supported for audio weighting filters.
- **Input Fs** — The sampling frequency in **Frequency units**. For example, if **Frequency units** is set to `kHz`, setting **Input Fs** to 40 is equivalent to a 40 kHz sampling frequency.

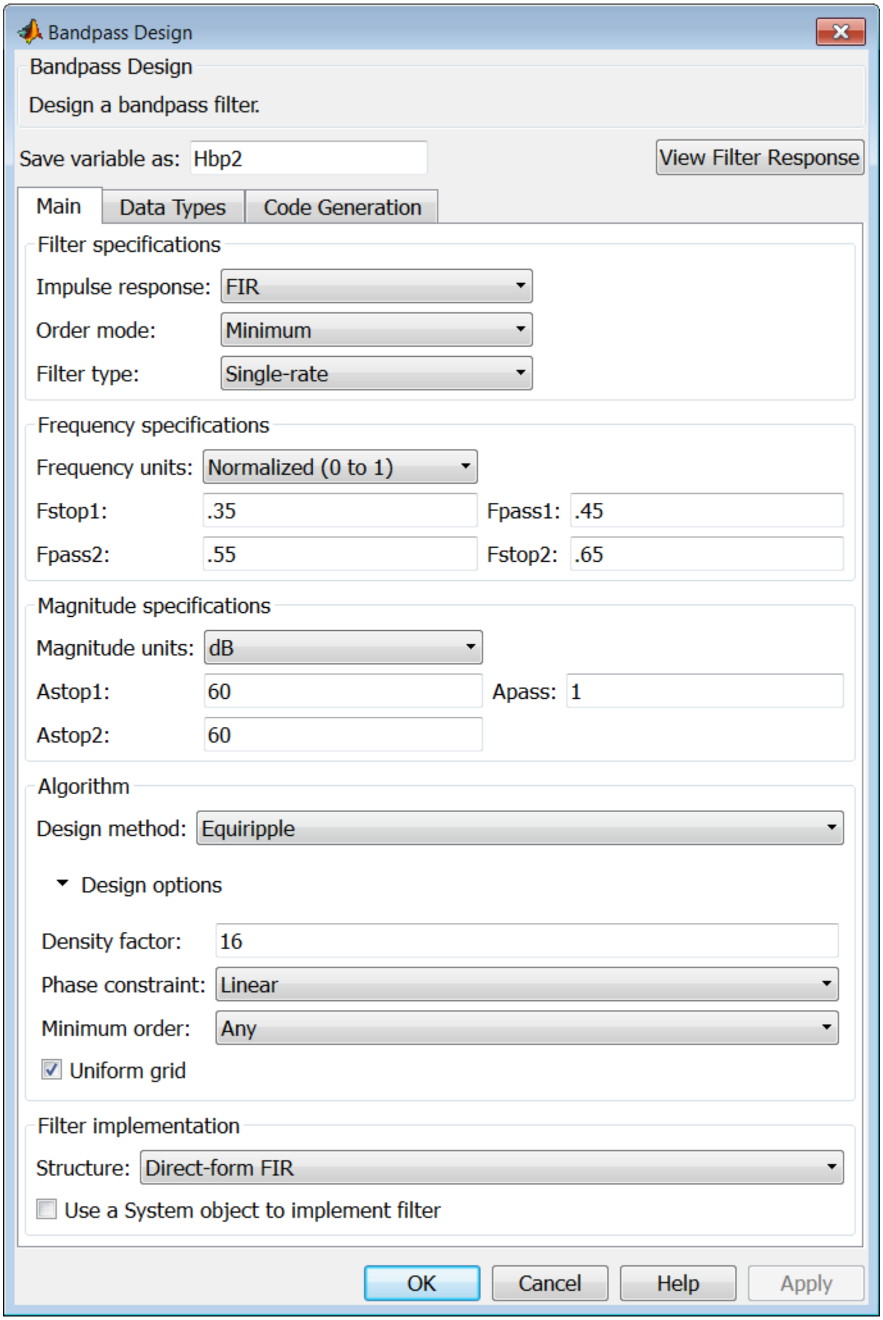
Algorithm

- **Design method** — Valid design methods depend on the weighting type. For type A and C weighting filters, the only valid design type is `ANSI S1.42`. This is an IIR design method that follows ANSI standard S1.42–2001. For a C message filter, the only valid design method is `Bell 41009`, which is an IIR design method following the Bell System Technical Reference PUB 41009. For a ITU-R 468–4 weighting filter, you can design an IIR or FIR filter. If you choose an IIR design, the design method is `IIR least p-norm`. If you choose an FIR design, the design method choices are: `Equiripple` or `Frequency Sampling`. For an ITU-T 0.41 weighting filter, the available FIR design methods are `Equiripple` or `Frequency Sampling`
- **Scale SOS filter coefficients to reduce chance of overflow** — Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Filter implementation

- **Structure** — For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. For audio weighting IIR filter designs, you can choose direct form I or II biquad (SOS). You can also choose to implement these structures in transposed form.

  For FIR designs, you can choose direct form, direct-form transposed, direct-form symmetric, direct-form asymmetric structures, or an overlap and add structure.
- **Use a System object to implement filter** — Selecting this check box gives you the choice of using a System object to implement the filter. By default, this check box is cleared. When the current design method or structure is not supported by a System object filter, then this check box is disabled.


Bandpass Design
Bandpass Design
Design a bandpass filter.
Save variable as: Hbp2
View Filter Response
Main
Data Types
Code Generation
Filter specifications
Impulse response: FIR
Order mode: Minimum
Filter type: Single-rate
Frequency specifications
Frequency units: Normalized (0 to 1)
Fstop1: .35
Fpass1: .45
Fpass2: .55
Fstop2: .65
Magnitude specifications
Magnitude units: dB
Astop1: 60
Apass: 1
Astop2: 60
Algorithm
Design method: Equiripple
Design options
Density factor: 16
Phase constraint: Linear
Minimum order: Any
Uniform grid
Filter implementation
Structure: Direct-form FIR
Use a System object to implement filter
OK
Cancel
Help
Apply

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note: The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select `Minimum` (the default) or `Specify` from the drop-down box. Selecting `Specify` enables the **Order** option so you can enter the filter order.

If you have the DSP System Toolbox software installed, you can specify IIR filters with different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

Filter type — This dialog only applies if you have the DSP System Toolbox software.

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

Order

Enter the filter order. This option is enabled only if you select `Specify` for **Order mode**.

Decimation Factor

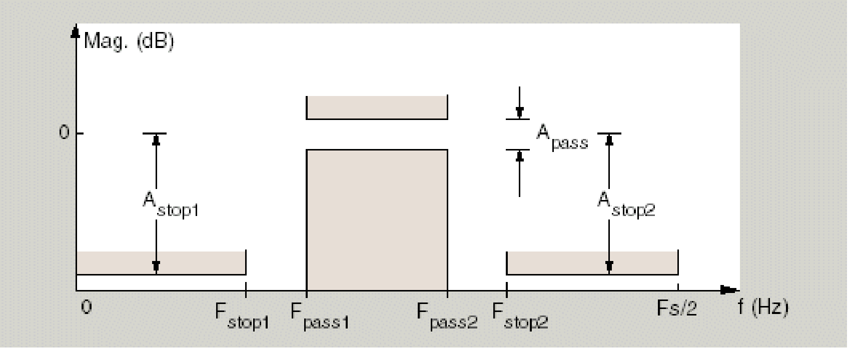
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, regions between specification values such as Fstop1 and Fpass1 represent transition regions where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. This dialog applies only when **Order mode** is **Specify**.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.
- `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.
- `3dB points` — Define the filter response by specifying the locations of the 3 dB points (IIR filters). The 3-dB point is the frequency for the point 3 dB below the passband value.
- `3dB points and passband width` — Define the filter by specifying frequencies for the 3-dB points in the filter response and the width of the passband. (IIR filters)

- `3dB points and stopband widths` — Define the filter by specifying frequencies for the 3-dB points in the filter response and the width of the stopband. (IIR filters)
- `6dB points` — Define the filter response by specifying the locations of the 6-dB points. The 6-dB point is the frequency for the point 6 dB below the passband value. (FIR filters)

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0—1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in hertz, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fstop1

Enter the frequency at the edge of the end of the first stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fpass1

Enter the frequency at the edge of the start of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fpass2

Enter the frequency at the edge of the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop2

Enter the frequency at the edge of the start of the second stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude constraints

Specify as `Unconstrained` or `Constrained bands`. You must have the DSP System Toolbox software to select `Constrained bands`. Selecting `Constrained bands` enables dialogs for both stopbands and the passband: **Astop1**, **Astop2**, and **Apass**. You cannot specify constraints for all three bands simultaneously.

Setting **Magnitude constraints** to `Constrained bands` enables the **Wstop** and **Wpass** options under **Design options**.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in dB (decibels). This is the default setting.
- `Squared` — Specify the magnitude in squared units.

Astop1

Enter the filter attenuation in the first stopband in the units you choose for **Magnitude units**, either linear or decibels.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop2

Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Valid when the **Design method** is `equiripple` and you have the DSP System Toolbox installed. Choose one of `Linear`, `Minimum`, or `Maximum`.

Minimum order

This option only applies when you have the DSP System Toolbox software and **Order mode** is `Minimum`.

Select `Any` (default), `Even`, or `Odd`. Selecting `Even` or `Odd` forces the minimum-order design to be an even or odd order.

Wstop1

Weight for the first stopband.

Wpass

Passband weight.

Wstop2

Weight for the second stopband.

Max pole radius

Valid only for IIR designs. Constrains the maximum pole radius. The default is 1. Reducing the max pole radius can produce a transfer function more resistant to quantization.

Init norm

Valid only for IIR designs. The initial norm used in the optimization. The default initial norm is 2.

Init numerator

Specifies an initial estimate of the filter numerator coefficients. This may be useful in difficult optimization problems.

Init denominator

Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems.

Filter implementation

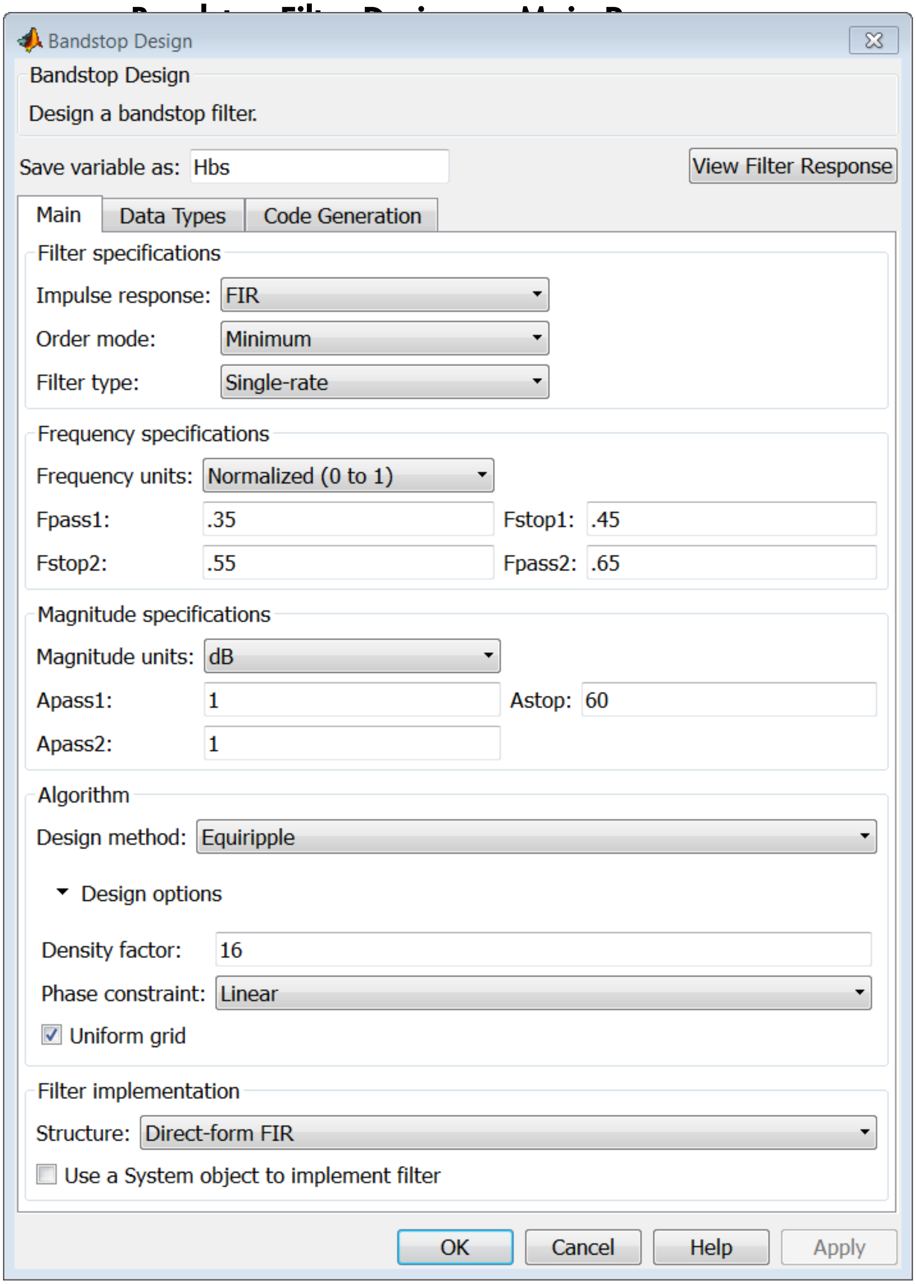
Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, this check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.


Bandstop Design
Bandstop Design
Design a bandstop filter.
Save variable as: Hbs
View Filter Response
Main
Data Types
Code Generation
Filter specifications
Impulse response: FIR
Order mode: Minimum
Filter type: Single-rate
Frequency specifications
Frequency units: Normalized (0 to 1)
Fpass1: .35
Fstop1: .45
Fstop2: .55
Fpass2: .65
Magnitude specifications
Magnitude units: dB
Apass1: 1
Astop: 60
Apass2: 1
Algorithm
Design method: Equiripple
Design options
Density factor: 16
Phase constraint: Linear
Uniform grid
Filter implementation
Structure: Direct-form FIR
Use a System object to implement filter
OK
Cancel
Help
Apply

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note: The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option so you can enter the filter order.

If you have the DSP System Toolbox software installed, you can specify IIR filters with different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

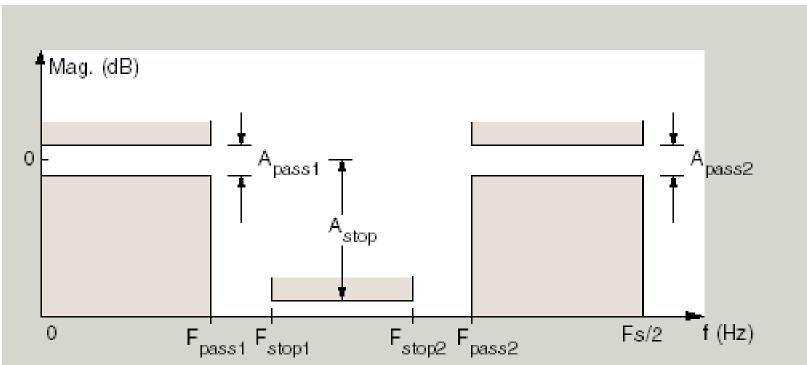
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



Frequency constraints

Select the filter features to use to define the frequency response characteristics. This dialog applies only when **Order mode** is **Specify**.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.
- `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.
- `3dB points` — Define the filter response by specifying the locations of the 3 dB points (IIR filters). The 3 dB point is the frequency for the point 3 dB point below the passband value.

- `3dB points and passband width` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband (IIR filters).
- `3dB points and stopband widths` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband (IIR filters).
- `6dB points` — Define the filter response by specifying the locations of the 6-dB points (FIR filters). The 6-dB point is the frequency for the point 6 dB point below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0—1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

When you design an interpolator, Fs represents the sampling frequency at the filter output rather than the filter input. This option is available only when you set **Filter type** is `interpolator`.

Fpass1

Enter the frequency at the edge of the end of the first passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop1

Enter the frequency at the edge of the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop2

Enter the frequency at the edge of the end of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fpass2

Enter the frequency at the edge of the start of the second passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude constraints

Specify as `Unconstrained` or `Constrained bands`. You must have the DSP System Toolbox software to select `Constrained bands`. Selecting `Constrained bands` enables dialogs for both passbands and the stopband: **Apass1**, **Apass2**, and **Astop**. You cannot specify constraints for all three bands simultaneously.

Setting **Magnitude constraints** to `Constrained bands` enables the **Wstop** and **Wpass** options under **Design options**.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Apass1

Enter the filter ripple allowed in the first passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels

Apass2

Enter the filter ripple allowed in the second passband in the units you choose for **Magnitude units**, either linear or decibels

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Valid when the **Design method** is `equiripple` and you have the DSP System Toolbox installed. Choose one of `Linear`, `Minimum`, or `Maximum`.

Minimum order

This option only applies when you have the DSP System Toolbox software and **Order mode** is `Minimum`.

Select `Any` (default), `Even`, or `Odd`. Selecting `Even` or `Odd` forces the minimum-order design to be an even or odd order.

Wpass1

Weight for the first passband.

Wstop

Stopband weight.

Wpass2

Weight for the second passband.

Match exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` .

Max pole radius

Valid only for IIR designs. Constrains the maximum pole radius. The default is 1. Reducing the max pole radius can produce a transfer function more resistant to quantization.

Init norm

Valid only for IIR designs. The initial norm used in the optimization. The default initial norm is 2.

Init numerator

Specifies an initial estimate of the filter numerator coefficients. This may be useful in difficult optimization problems.

Init denominator

Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems.

Filter implementation

Structure

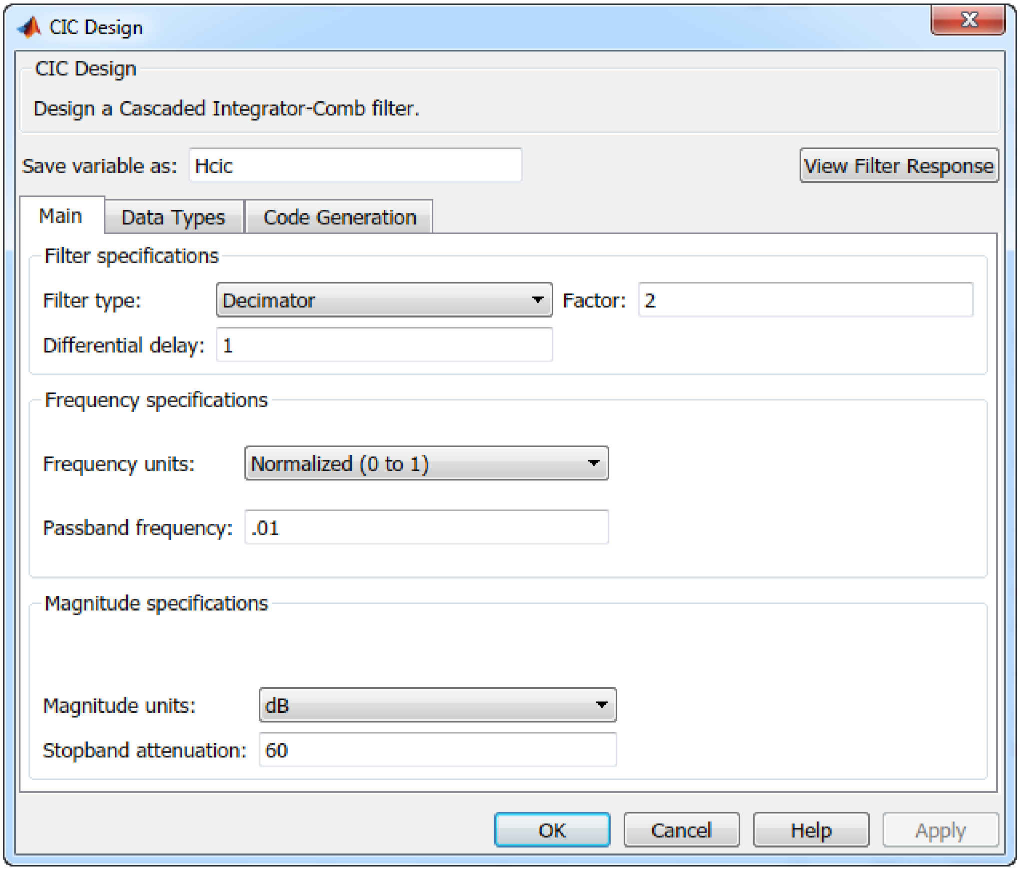
For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, this check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

CIC Filter Design — Main Pane



Filter specifications

Parameters in this group enable you to specify your CIC filter format, such as the filter type and the differential delay.

Filter type

Select whether your filter will be a `decimator` or an `interpolator`. Your choice determines the type of filter and the design methods and structures that are available to implement your filter. Selecting `decimator` or `interpolator` activates the **Factor** option. When you design an interpolator, you enable the **Output Fs** parameter.

When you design either a decimator or interpolator, the resulting filter is a CIC filter that decimates or interpolates your input signal.

Differential Delay

Specify the differential delay of your CIC filter as an integer value greater than or equal to 1. The default value is 1. The differential delay changes the shape, number, and location of nulls in the filter response. Increasing the differential delay increases the sharpness of the nulls and the response between the nulls. In practice, differential delay values of 1 or 2 are the most common.

Factor

Specify the decimation or interpolation factor for your filter as an integer value greater than or equal to 1. The default value is 2.

Frequency specifications

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

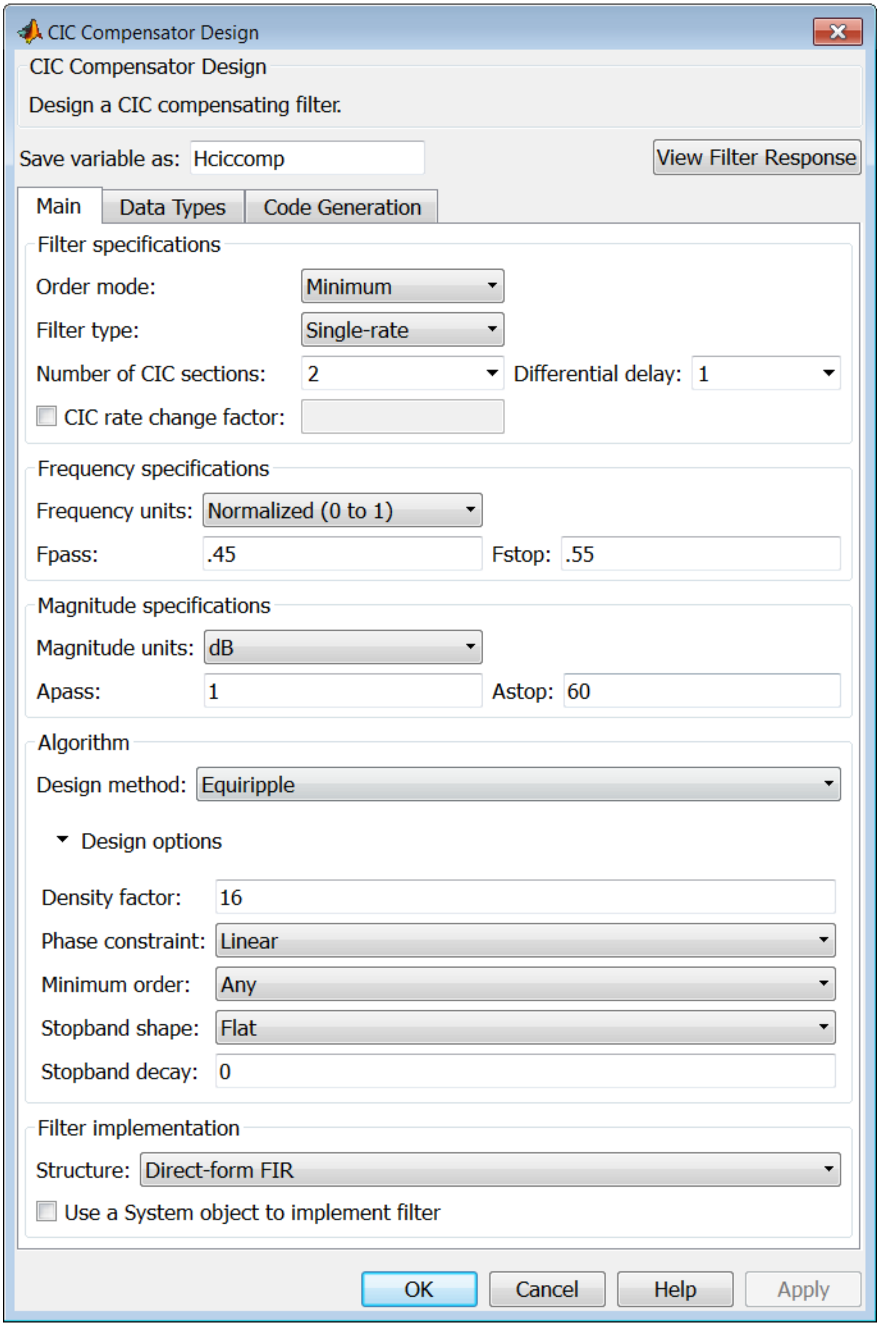
Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.


CIC Compensator Design
CIC Compensator Design
Design a CIC compensating filter.
Save variable as: Hciccomp
View Filter Response
Main
Data Types
Code Generation
Filter specifications
Order mode: Minimum
Filter type: Single-rate
Number of CIC sections: 2
Differential delay: 1
CIC rate change factor:
Frequency specifications
Frequency units: Normalized (0 to 1)
Fpass: .45
Fstop: .55
Magnitude specifications
Magnitude units: dB
Apass: 1
Astop: 60
Algorithm
Design method: Equiripple
Design options
Density factor: 16
Phase constraint: Linear
Minimum order: Any
Stopband shape: Flat
Stopband decay: 0
Filter implementation
Structure: Direct-form FIR
Use a System object to implement filter
OK
Cancel
Help
Apply

Filter specifications

Parameters in this group enable you to specify your filter format, such as the filter order mode and the filter type.

Order mode

Select `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Number of CIC sections

Specify the number of sections in the CIC filter for which you are designing this compensator. Select the number of sections from the drop-down list or enter the number.

Differential Delay

Specify the differential delay of your target CIC filter. The default value is `1`. Most CIC filters use 1 or 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve.

Frequency specifications

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.

- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note: Generally, **Minimum order** designs are not available for IIR filters.

Match exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterBuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterBuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter implementation

Structure

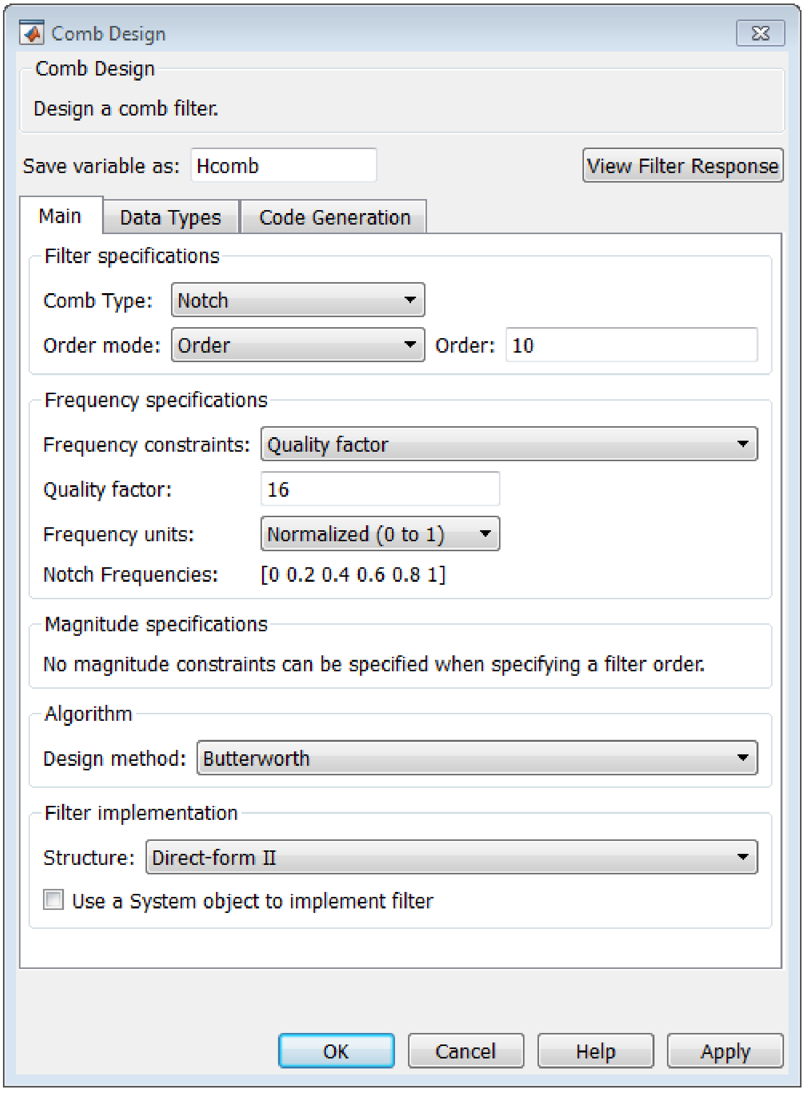
For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, this check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Comb Filter Design —Main Pane



Filter specifications

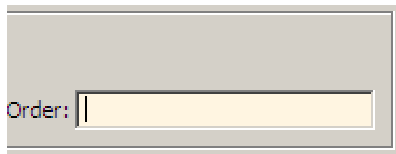
Parameters in this group enable you to specify the type of comb filter and the number of peaks or notches.

Comb Type

Select `Notch` or `Peak` from the drop-down list. `Notch` creates a comb filter that attenuates a set of harmonically related frequencies. `Peak` creates a comb filter that amplifies a set of harmonically related frequencies.

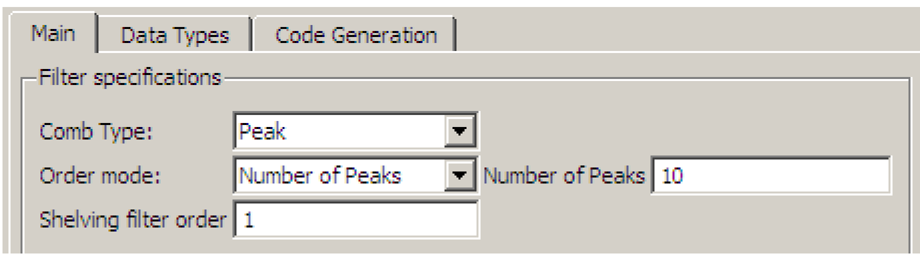
Order mode

Select `Order` or `Number of Peaks/Notches` from the drop-down menu.



Select `Order` to enter the desired filter order in the dialog box. The comb filter has notches or peaks at increments of 2/`Order` in normalized frequency units.

Select `Number of Peaks` or `Number of Notches` to specify the number of peaks or notches and the `Shelving filter order`



.

Shelving filter order

The `Shelving filter order` is a positive integer that determines the sharpness of the peaks or notches. Larger values result in sharper peaks or notches.

Frequency specifications

Parameters in this group enable you to specify the frequency constraints and frequency units.

Frequency specifications

Select `Quality factor` or `Bandwidth`.

`Quality factor` is the ratio of the center frequency of the peak or notch to the bandwidth calculated at the –3 dB point.

`Bandwidth` specifies the bandwidth of the peak or notch. By default the bandwidth is measured at the –3 dB point. For example, setting the bandwidth equal to 0.1 results in 3 dB frequencies at normalized frequencies 0.05 above and below the center frequency of the peak or notch.

Frequency Units

Specify the frequency units. The default is normalized frequency. Choosing an option in Hz enables the **Input Fs** dialog box.

Magnitude specifications

Specify the units for the magnitude specification and the gain at which the bandwidth is measured. This menu is disabled if you specify a filter order. Select one of the following magnitude units from the drop down list:

- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Bandwidth gain — Specify the gain at which the bandwidth is measured. The default is –3 dB.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

Design Method

The IIR Butterworth design is the only option for peaking or notching comb filters.

Filter implementation

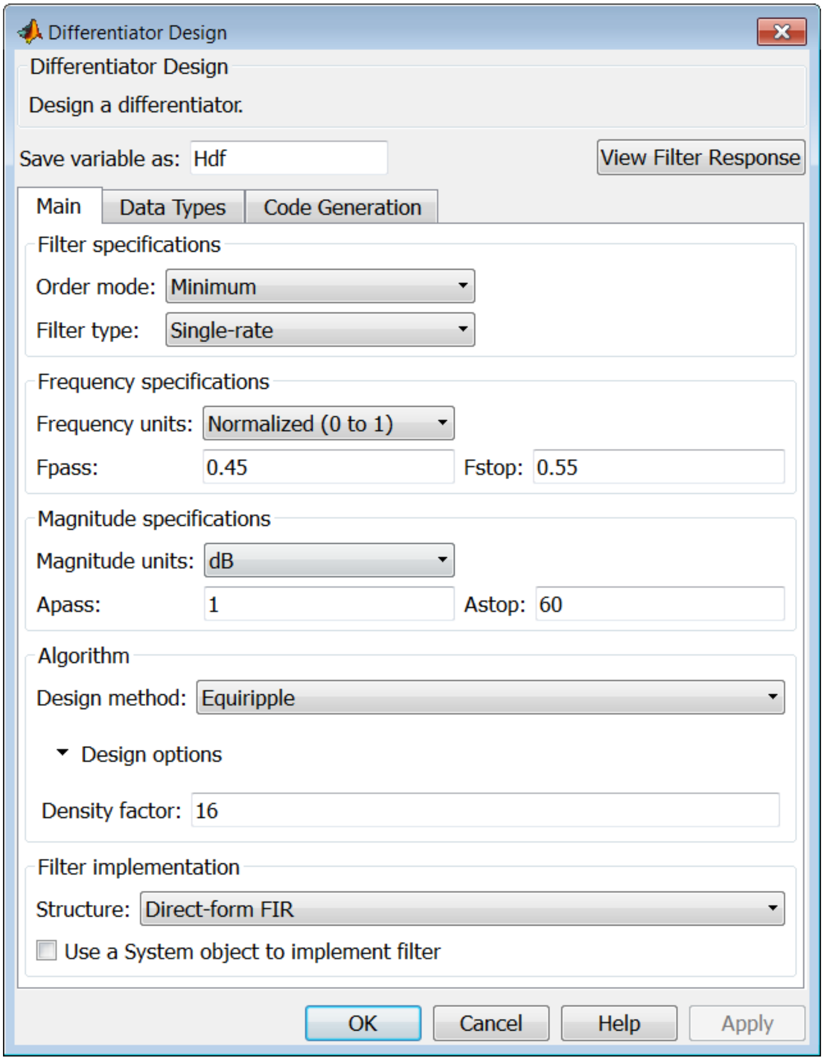
Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

Use a System object to implement filter

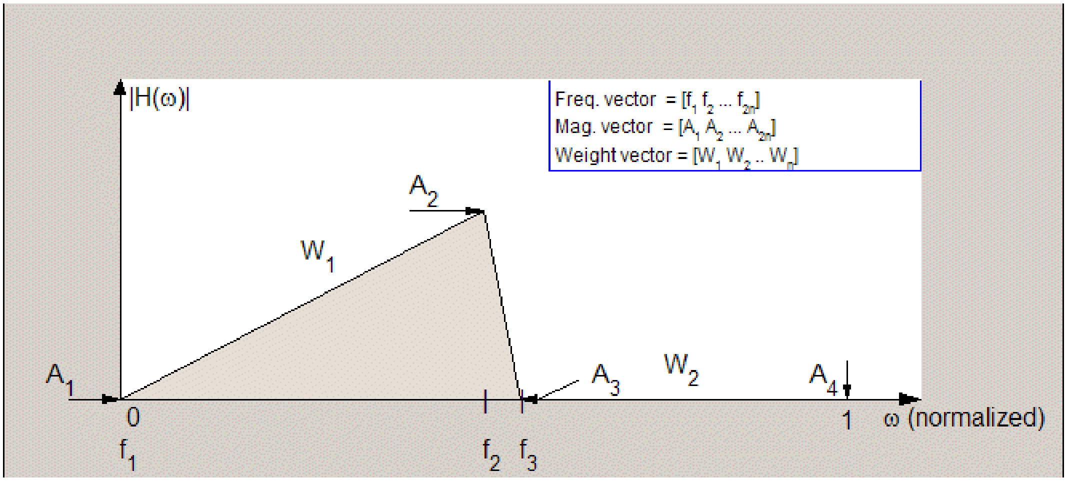
Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

Differentiator Filter Design — Main Pane



Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, regions between specification values such as **Fpass** (f_1) and **Fstop** (f_3) represent transition regions where the filter response is not explicitly defined.

Order mode

Select `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve.

Frequency constraints

This option is only available when you specify the order of the filter design. Supported options are `Unconstrained` and `Passband edge and stopband edge`.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0—1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude constraints

This option is only available when you specify the order of your filter design. The options for **Magnitude constraints** depend on the value of the **Frequency constraints**. If the value of **Frequency constraints** is `Unconstrained`, **Magnitude constraints** must be `Unconstrained`. If the value of **Frequency constraints** is `Passband edge and stopband edge`, **Magnitude constraints** can be `Unconstrained`, `Passband ripple`, or `Stopband attenuation`.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop2

Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Wpass

Passband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to `Passband edge and stopband edge` and the **Design method** is `Equiripple`.

Wstop

Stopband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to `Passband edge and stopband edge` and the **Design method** is `Equiripple`.

Filter implementation

Structure

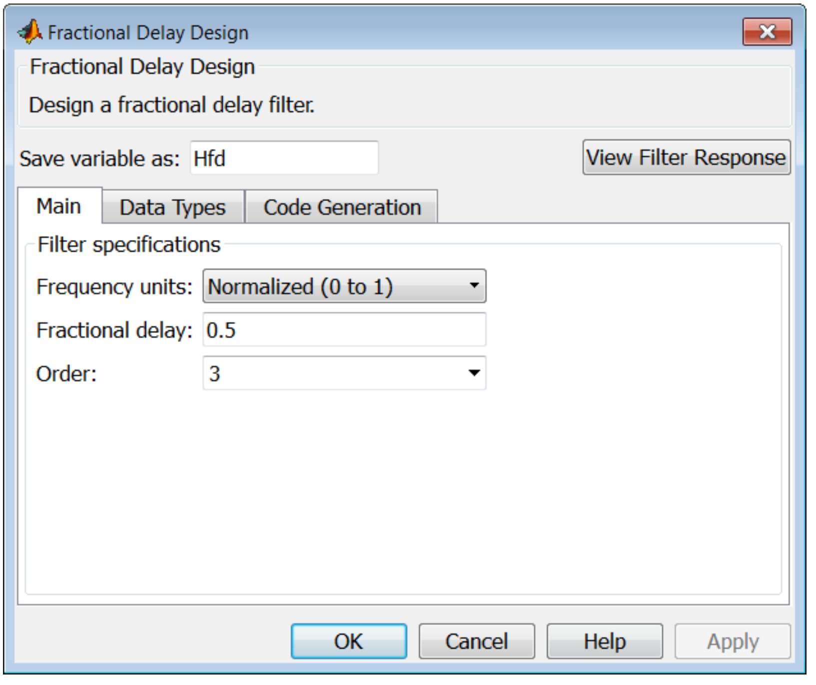
For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Fractional Delay Design — Main Pane



Frequency specifications

Parameters in this group enable you to specify your filter format, such as the fractional delay and the filter order.

Order

If you choose `Specify` for **Order mode**, enter your filter order in this field, or select the order from the drop-down list.`filterBuilder` designs a filter with the order you specify.

Fractional delay

Specify a value between 0 and 1 samples for the filter fractional delay. The default value is `0.5` samples.

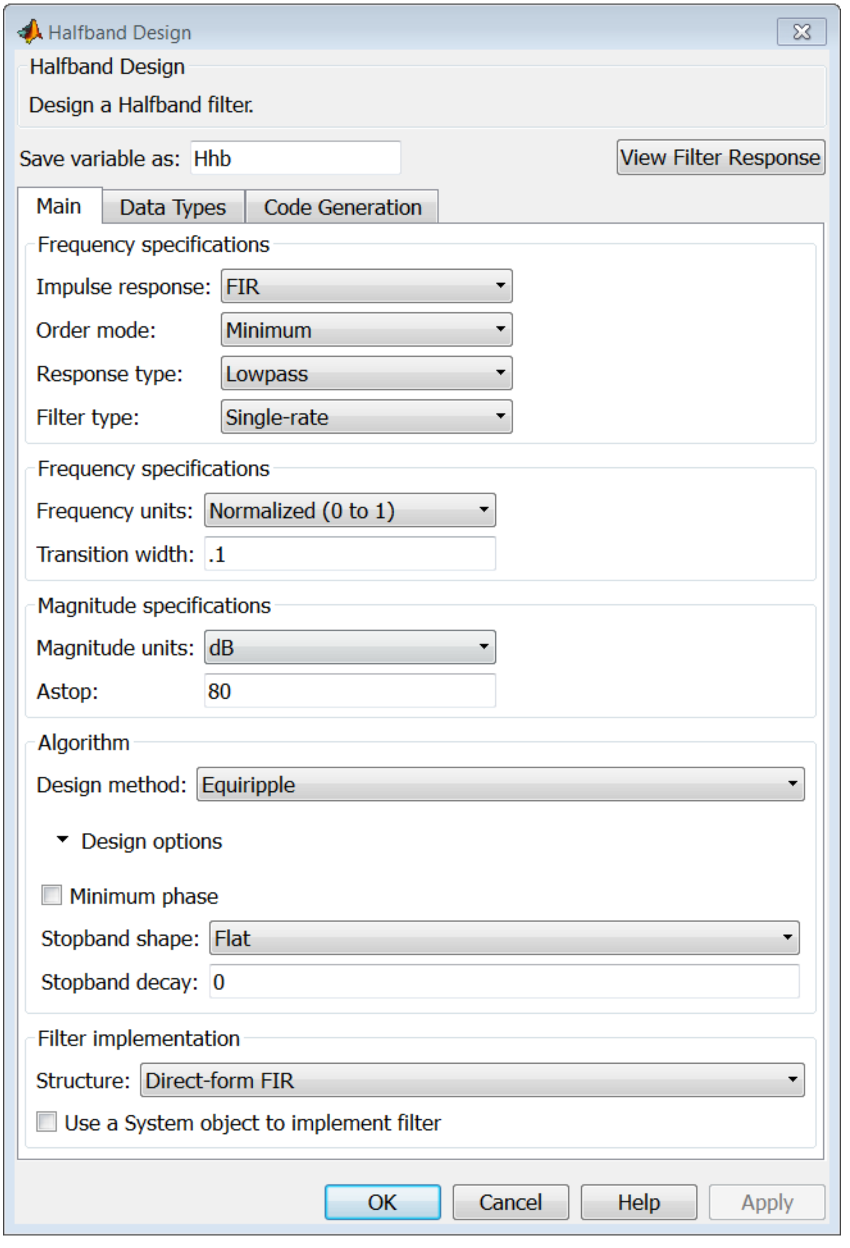
Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Halfband Filter Design — Main Pane



Filter specifications

Parameters in this group enable you to specify your filter type and order.

Impulse response

Select `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note: The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, or `Interpolator`. By default, `filterBuilder` specifies single-rate filters.

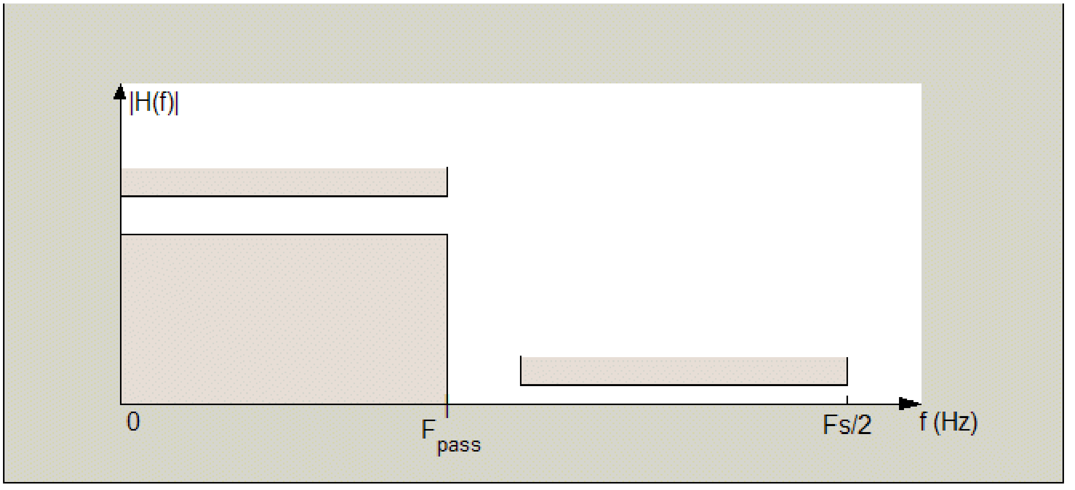
When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that decimates or interpolates your input by a factor of two.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications for a halfband lowpass filter look similar to those shown in the following figure.



In the figure, the transition region lies between the end of the passband and the start of the stopband. The width is defined explicitly by the value of **Transition width**.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Transition width

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. For FIR halfband filters, the available design options are `Equiripple` and `Kaiser window`. For IIR halfband filters, the available design options are `Butterworth`, `Elliptic`, and `IIR quasi-linear phase`.

Design Options

The following design options are available for FIR halfband filters when the user specifies an equiripple design:

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterBuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterBuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter implementation

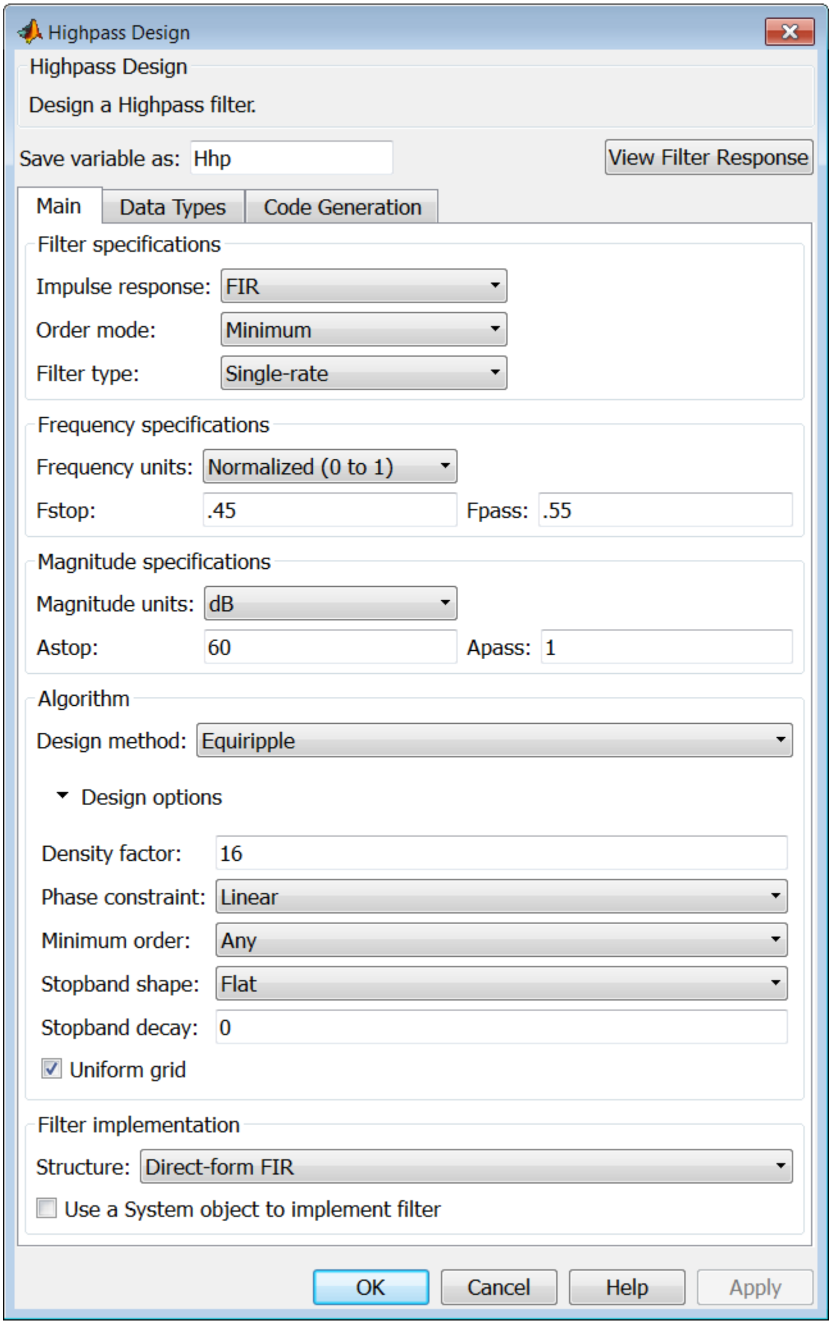
Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

Use a System object to implement filter

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to either `Interpolator` or `Decimator`. The filter builder always implements the filter as a System object.


Highpass Design
Highpass Design
Design a Highpass filter.
Save variable as: Hhp
View Filter Response
Main
Data Types
Code Generation
Filter specifications
Impulse response: FIR
Order mode: Minimum
Filter type: Single-rate
Frequency specifications
Frequency units: Normalized (0 to 1)
Fstop: .45
Fpass: .55
Magnitude specifications
Magnitude units: dB
Astop: 60
Apass: 1
Algorithm
Design method: Equiripple
Design options
Density factor: 16
Phase constraint: Linear
Minimum order: Any
Stopband shape: Flat
Stopband decay: 0
Uniform grid
Filter implementation
Structure: Direct-form FIR
Use a System object to implement filter
OK
Cancel
Help
Apply

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note: The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option so you can enter the filter order.

If your **Impulse response** is `IIR`, you can specify an equal order for the numerator and denominator, or different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

Filter type

This option is only available if you have the DSP System Toolbox software. Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a highpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

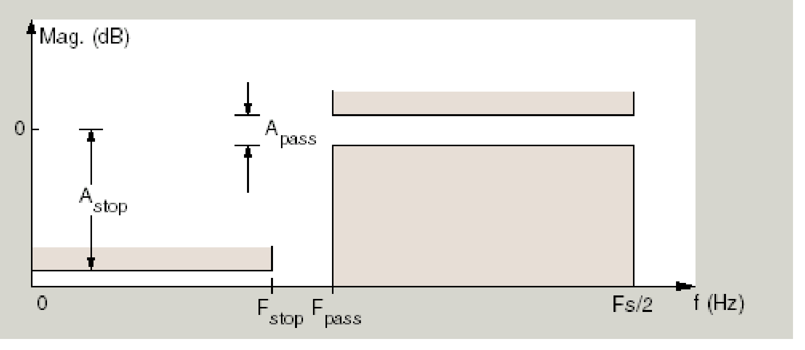
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, the region between specification values Fstop and Fpass represents the transition region where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Stopband edge and passband edge` — Define the filter by specifying the frequencies for the edges for the stopband and passband.
- `Passband edge` — Define the filter by specifying the frequency for the edge of the passband.
- `Stopband edge` — Define the filter by specifying the frequency for the edges of the stopband.
- `Stopband edge and 3dB point` — Define the filter by specifying the stopband edge frequency and the 3-dB down point (IIR designs).
- `3dB point and passband edge` — Define the filter by specifying the 3-dB down point and passband edge frequency (IIR designs).

- `3dB point` — Define the filter by specifying the frequency for the 3-dB point (IIR designs or maxflat FIR).
- `6dB point` — Define the filter by specifying the frequency for the 6-dB point in the filter response (FIR designs).

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default).
- `Squared` — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

This option only applies when you have the DSP System Toolbox software and when the **Design method** is `equiripple`. Select one of `Linear`, `Minimum`, or `Maximum`.

Minimum order — This option only applies when you have the DSP System Toolbox software and the **Order mode** is `Minimum`.

Select `Any` (default), `Even`, or `Odd`. Selecting `Even` or `Odd` forces the minimum-order design to be an even or odd order.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband when you select `Passband` or `Stopband`.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterBuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterBuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Wpass

Passband weight. This option only applies when **Impulse response** is `FIR` and **Order mode** is `Specify`.

Wstop

Stopband weight. This option only applies when **Impulse response** is `FIR` and **Order mode** is `Specify`.

Filter implementation

Structure

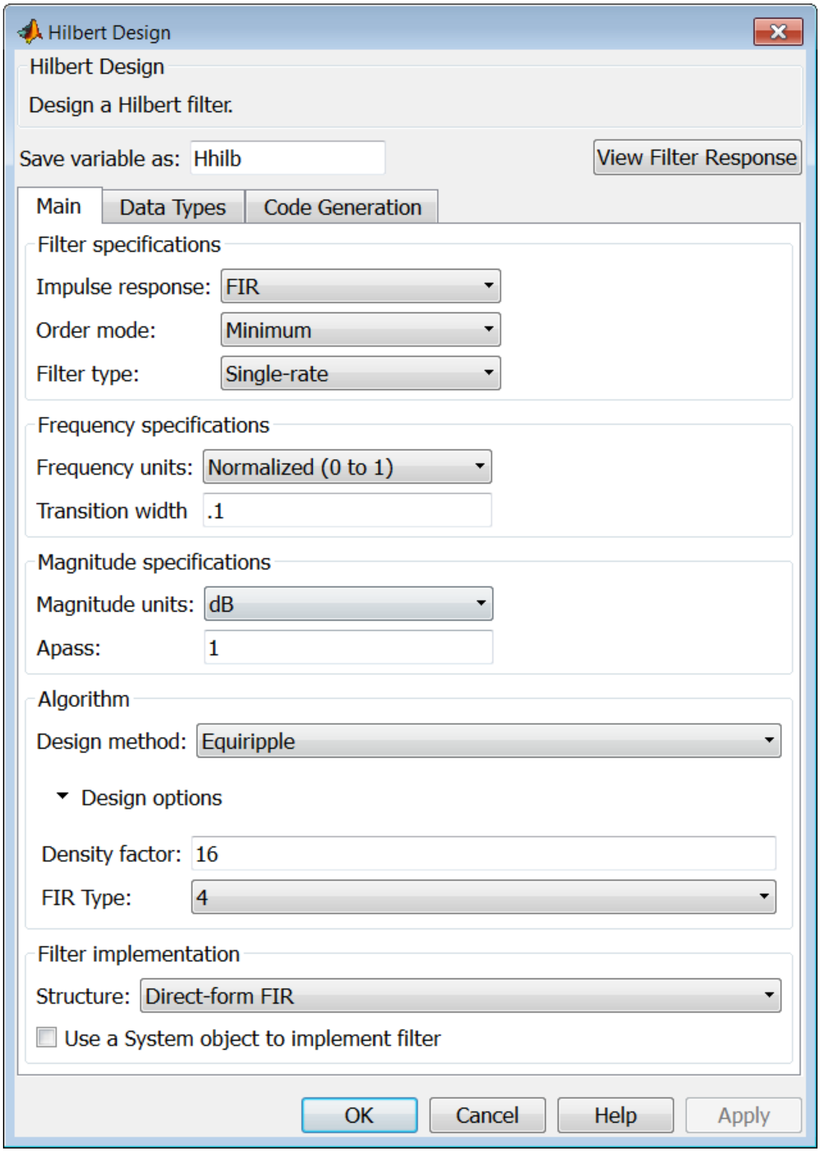
For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Hilbert Filter Design — Main Pane



Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note: The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

This option is only available if you have the DSP System Toolbox software. Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

This option is only available if you have the DSP System Toolbox software. Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

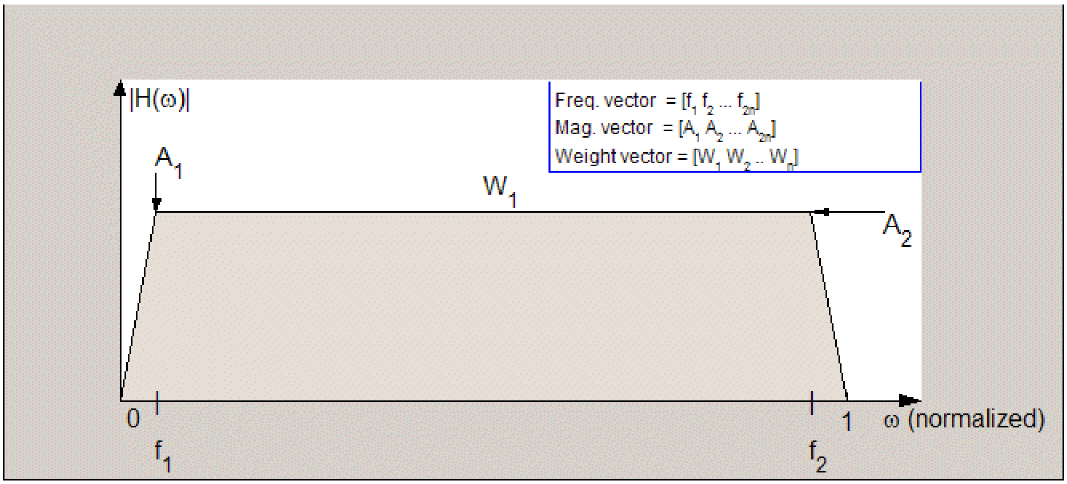
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, the regions between 0 and f_1 and between f_2 and 1 represent the transition regions where the filter response is explicitly defined by the transition width.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is

available when you select one of the frequency options from the **Frequency units** list.

Transition width

Specify the width of the transitions at the ends of the passband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default)
- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

FIR Type

This option is only available in a minimum-order design. Specify whether to design a type 3 or a type 4 FIR filter. The filter type is defined as follows:

- Type 3 — FIR filter with even order antisymmetric coefficients
- Type 4 — FIR filter with odd order antisymmetric coefficients

Select `3` or `4` from the drop-down list.

Filter implementation

Structure

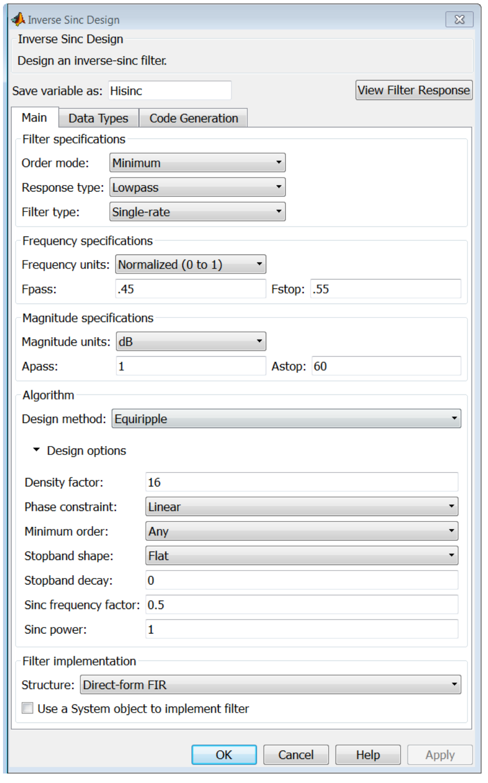
For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Inverse Sinc Filter Design — Main Pane



Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Order mode

Select `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Response type

Select `Lowpass` or `Highpass` to design an inverse sinc lowpass or highpass filter.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

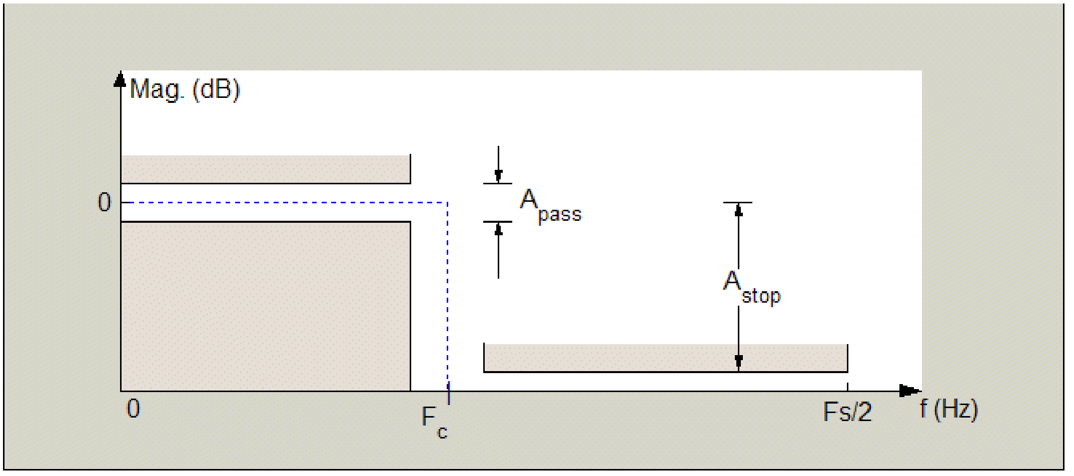
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



Regions between specification values such as Fpass and Fstop represent transition regions where the filter response is not explicitly defined.

Frequency constraints

This option is only available when you specify the filter order. The following options are available:

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
- `Passband edge` — Define the filter by specifying frequencies for the edges of the passband.
- `Stopband edge` — Define the filter by specifying frequencies for the edges of the stopbands.
- `6dB point` — The 6-dB point is the frequency for the point 6 dB point below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default)
- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Available options are `Linear`, `Minimum`, and `Maximum`.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options;

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterBuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent *n* in the relation $(1/f)^n$ to define the stopband decay. `filterBuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Sinc frequency factor

A frequency dilation factor. The sinc frequency factor, C , parameterizes the passband magnitude response for a lowpass design through $H(\omega) = \text{sinc}(C\omega)$^$(-P)$ and for a highpass design through $H(\omega) = \text{sinc}(C(1-\omega))$^$(-P)$.

Sinc power

Negative power of passband magnitude response. The sinc power, P, parameterizes the passband magnitude response for a lowpass design through $H(\omega) = \text{sinc}(C\omega)$^$(-P)$ and for a highpass design through $H(\omega) = \text{sinc}(C(1-\omega))$^$(-P)$.

Filter implementation

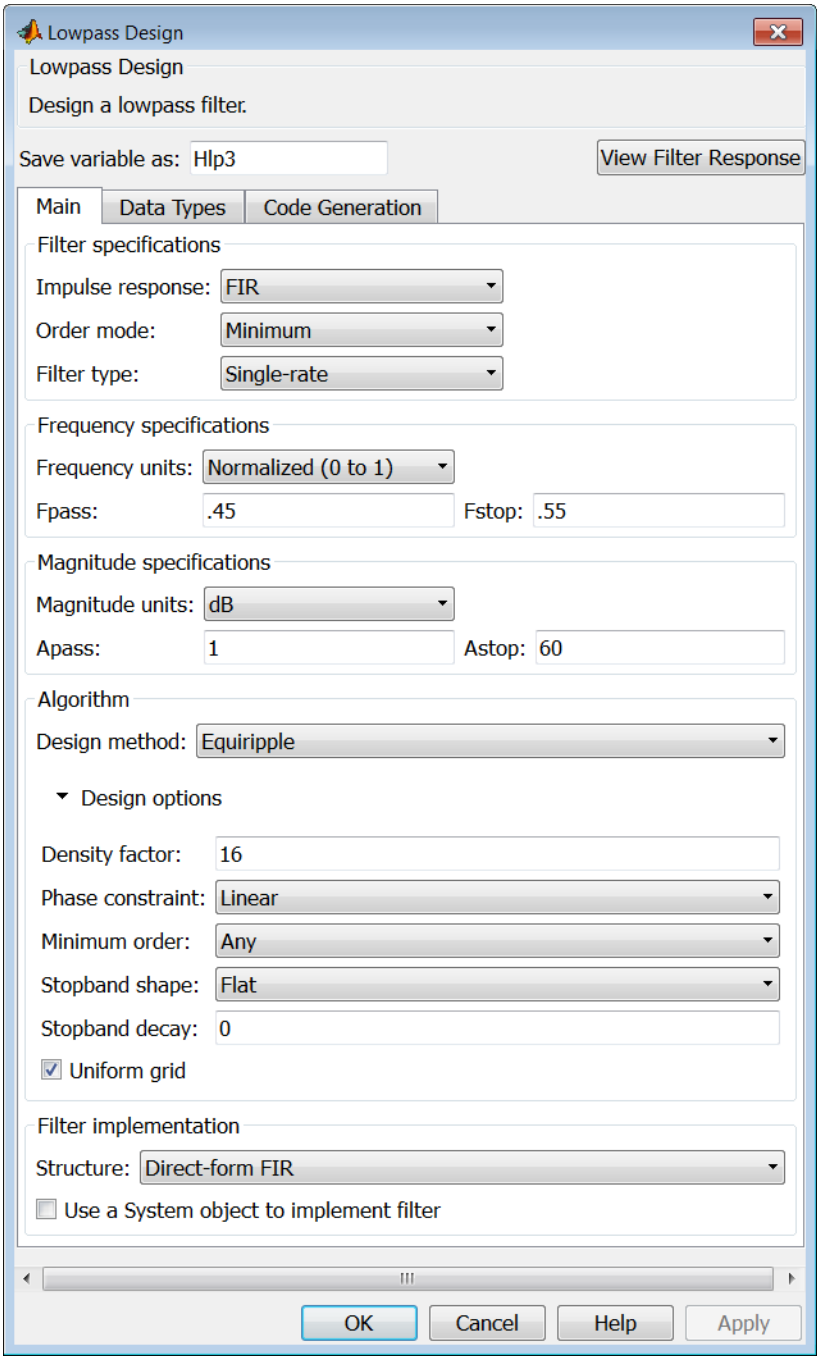
Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure.

Use a System object to implement filter

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.


Lowpass Design
Lowpass Design
Design a lowpass filter.
Save variable as: Hlp3
View Filter Response
Main
Data Types
Code Generation
Filter specifications
Impulse response: FIR
Order mode: Minimum
Filter type: Single-rate
Frequency specifications
Frequency units: Normalized (0 to 1)
Fpass: .45
Fstop: .55
Magnitude specifications
Magnitude units: dB
Apass: 1
Astop: 60
Algorithm
Design method: Equiripple
Design options
Density factor: 16
Phase constraint: Linear
Minimum order: Any
Stopband shape: Flat
Stopband decay: 0
Uniform grid
Filter implementation
Structure: Direct-form FIR
Use a System object to implement filter
OK
Cancel
Help
Apply

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note: The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

If your **Impulse response** is `IIR`, you can specify an equal order for the numerator and denominator, or different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

Filter type

This option is only available if you have the DSP System Toolbox. Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

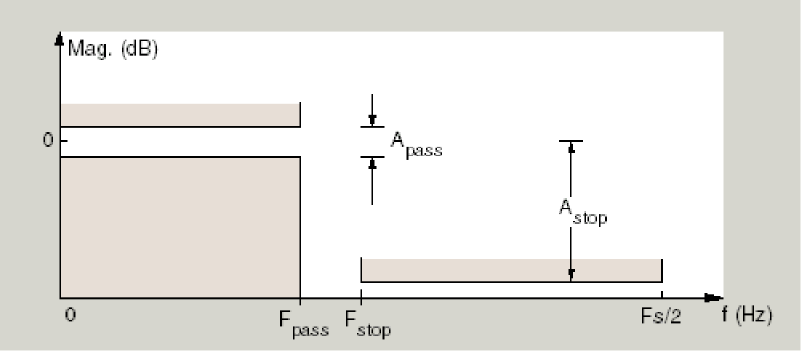
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to the one shown in the following figure.



In the figure, regions between specification values such as F_{pass} and F_{stop} represent transition regions where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edge` — Define the filter by specifying the frequencies for the edge of the stopband and passband.
- `Passband edge` — Define the filter by specifying the frequency for the edge of the passband.
- `Stopband edge` — Define the filter by specifying the frequency for the edges of the stopband.

- `Passband edge and 3dB point` — Define the filter by specifying the passband edge frequency and the 3-dB down point (IIR designs).
- `3dB point and stopband edge` — Define the filter by specifying the 3-dB down point and stopband edge frequency (IIR designs).
- `3dB point` — Define the filter by specifying the frequency for the 3-dB point (IIR designs or maxflat FIR).
- `6dB point` — Define the filter by specifying the frequency for the 6-dB point in the filter response (FIR designs).

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.

- `dB` — Specify the magnitude in decibels (default)
- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

This option only applies when you have the DSP System Toolbox software and when the **Design method** is `equiripple`. Select one of `Linear`, `Minimum`, or `Maximum`.

Minimum order — This option only applies when you have the DSP System Toolbox software and the **Order mode** is `Minimum`.

Select `Any` (default), `Even`, or `Odd`. Selecting `Even` or `Odd` forces the minimum-order design to be an even or odd order.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband when you select `Passband` or `Stopband`.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterBuilder` applies that slope to the stopband.

- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterBuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Wpass

Passband weight. This option only applies when **Impulse response** is `FIR` and **Order mode** is `Specify`.

Wstop

Stopband weight. This option only applies when **Impulse response** is `FIR` and **Order mode** is `Specify`.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

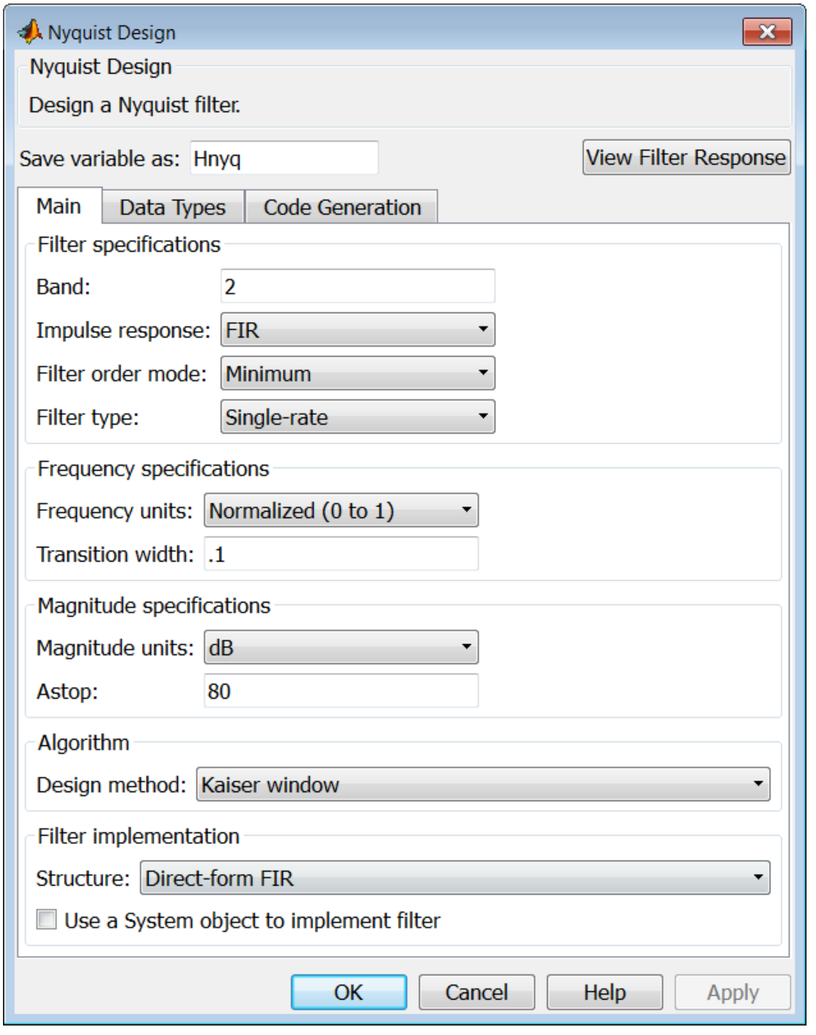
This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Notch

See “Peak/Notch Filter Design — Main Pane” on page 5-793.

Nyquist Filter Design — Main Pane



Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Band

Specifies the location of the center of the transition region between the passband and the stopband. The center of the transition region, bw, is calculated using the value for Band:

bw = Fs/(2 × Band).

Impulse response

Select `FIR` or `IIR` from the drop-down list, where `FIR` is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note: The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, `filterBuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

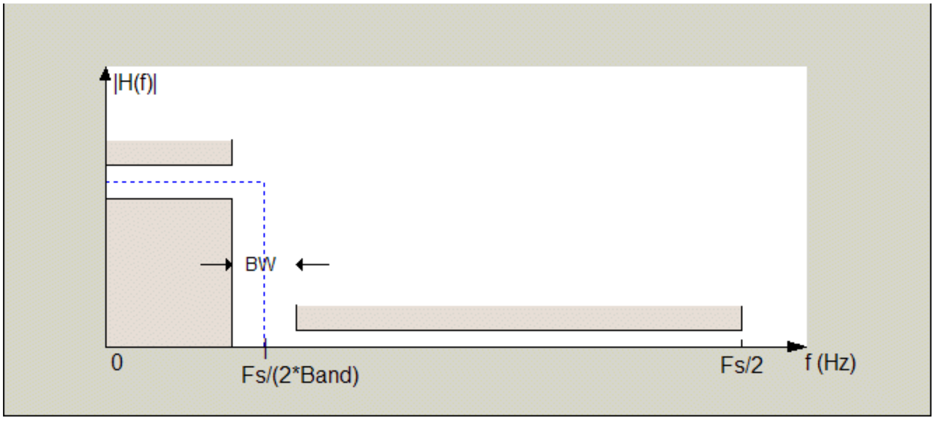
Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.



In the figure, BW is the width of the transition region and **Band** determines the location of the center of the region.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stopbands and passbands.
- `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.
- `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.
- `3 dB points` — Define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point 3 dB point below the passband value.

- `3 dB points and passband width` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.
- `3 dB points and stopband widths` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0–1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Transition width

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.
- `dB` — Specify the magnitude in decibels (default)
- `Squared` — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

When you select this parameter, the design method determines and designs the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note: Generally, **Minimum order** designs are not available for IIR filters.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterBuilder` applies that slope to the stopband.
- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterBuilder` applies the

$(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter implementation

Structure

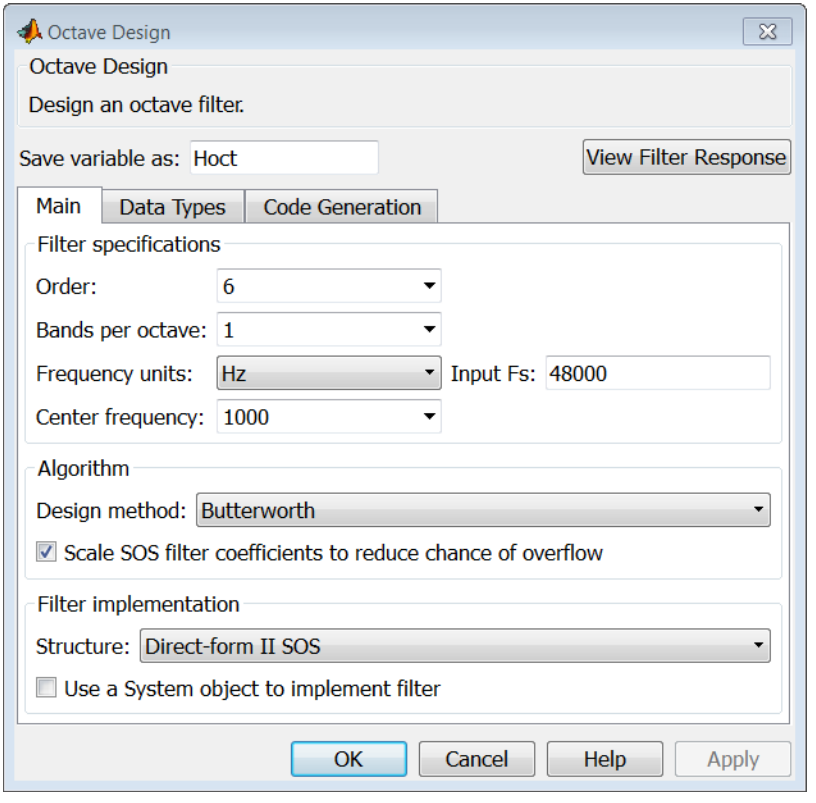
For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Octave Filter Design — Main Pane



Filter specifications

Order

Specify filter order. Possible values are: `4, 6, 8, 10`.

Bands per octave

Specify the number of bands per octave. Possible values are: `1, 3, 6, 12, 24`.

Frequency units

Specify frequency units as `Hz` or `kHz`.

Input Fs

Specify the input sampling frequency in the frequency units specified previously.

Center Frequency

Select from the drop-down list of available center frequency values.

Algorithm

Design Method

Butterworth is the design method used for this type of filter.

Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

Filter implementation

Structure

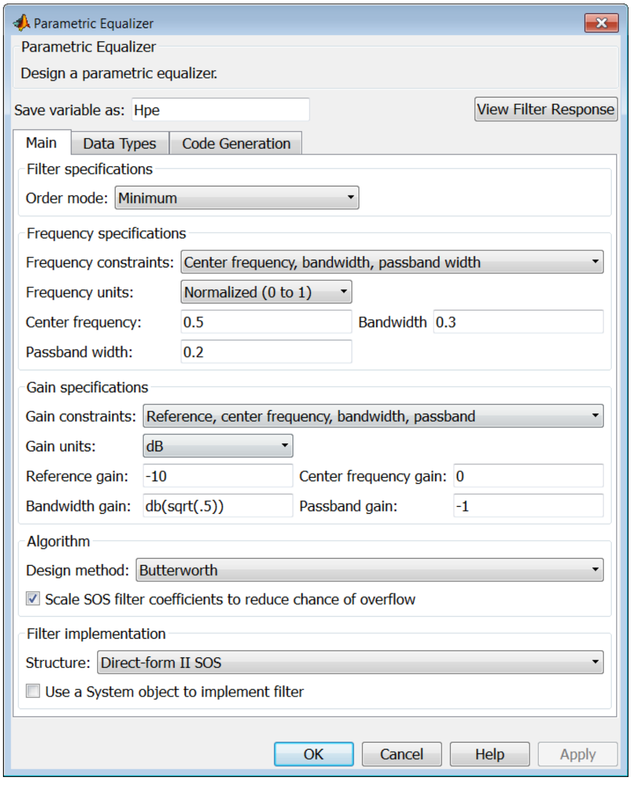
Specify filter structure. Choose from:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

Use a System object to implement filter

Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Parametric Equalizer Filter Design — Main Pane



Filter specifications

Order mode

Select `Minimum` to design a minimum order filter that meets the design specifications, or `Specify` to enter a specific filter order. The order mode also affects the possible frequency constraints, which in turn limit the gain specifications. For example, if you specify a `Minimum` order filter, the available frequency constraints are:

- `Center frequency, bandwidth, passband width`
- `Center frequency, bandwidth, stopband width`

If you select `Specify`, the available frequency constraints are:

- `Center frequency, bandwidth`
- `Center frequency, quality factor`
- `Shelf type, cutoff frequency, quality factor`
- `Shelf type, cutoff frequency, shelf slope parameter`
- `Low frequency, high frequency`

Order

This parameter is enabled only if the **Order mode** is set to `Specify`. Enter the filter order in this text box.

Frequency specifications

Depending on the filter order, the possible frequency constraints change. Once you choose the frequency constraints, the input boxes in this area change to reflect the selection.

Frequency constraints

Select the specification to represent the frequency constraints. The following options are available:

- `Center frequency, bandwidth, passband width` (available for minimum order only)
- `Center frequency, bandwidth, stopband width` (available for minimum order only)
- `Center frequency, bandwidth` (available for a specified order only)
- `Center frequency, quality factor` (available for a specified order only)
- `Shelf type, cutoff frequency, quality factor` (available for a specified order only)
- `Shelf type, cutoff frequency, shelf slope parameter` (available for a specified order only)
- `Low frequency, high frequency` (available for a specified order only)

Frequency units

Select the frequency units from the available drop down list (`Normalized, Hz, kHz, MHz, GHz`). If `Normalized` is selected, then the **Input Fs** box is disabled for input.

Input Fs

Enter the input sampling frequency. This input box is disabled for input if `Normalized` is selected in the **Frequency units** input box.

Center frequency

Enter the center frequency in the units specified by the value in **Frequency units**.

Bandwidth

The bandwidth determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Bandwidth gain** in the **Gain specifications** section. By default, the **Bandwidth gain** defaults to `db(sqrt(.5))`, or –3 dB relative to the center frequency. The **Bandwidth** property only applies when the **Frequency constraints** are: `Center frequency, bandwidth, passband width`, `Center frequency, bandwidth, stopband width`, or `Center frequency, bandwidth`.

Passband width

The passband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Passband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is `Center frequency, bandwidth, passband width`.

Stopband width

The stopband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Stopband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is `Center frequency, bandwidth, stopband width`.

Low frequency

Enter the low frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is `Low frequency, high frequency`. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

High frequency

Enter the high frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is `Low frequency, high frequency`. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

Gain specifications

Depending on the filter order and frequency constraints, the possible gain constraints change. Also, once you choose the gain constraints the input boxes in this area change to reflect the selection.

Gain constraints

Select the specification array to represent gain constraints, and remember that not all of these options are available for all configurations. The following is a list of all available options:

- `Reference, center frequency, bandwidth, passband`
- `Reference, center frequency, bandwidth, stopband`
- `Reference, center frequency, bandwidth, passband, stopband`
- `Reference, center frequency, bandwidth`

Gain units

Specify the gain units either `dB` or `squared`. These units are used for all gain specifications in the dialog box.

Reference gain

The reference gain determines the level to which the filter magnitude attenuates in **Gain units**. The reference gain is a *floor* gain for the filter magnitude response. For example, you may use the reference gain together with the **Center frequency gain** to leave certain frequencies unattenuated (reference gain of 0 dB) while boosting other frequencies.

Bandwidth gain

Specifies the gain in **Gain units** at which the bandwidth is defined. This property applies only when the **Frequency constraints** specification contains a `bandwidth` parameter, or is `Low frequency, high frequency`.

Center frequency gain

Specify the center frequency in **Gain units**

Passband gain

The passband gain determines the level in **Gain units** at which the passband is defined. The passband is determined either by the **Passband width** value, or the **Low frequency** and **High frequency** values in the **Frequency specifications** section.

Stopband gain

The stopband gain is the level in **Gain units** at which the stopband is defined. This property applies only when the **Order mode** is minimum and the **Frequency constraints** are `Center frequency, bandwidth, stopband width`.

Boost/cut gain

The boost/cut gain applies only when the designing a shelving filter. Shelving filters include the `Shelf type` parameter in the **Frequency constraints** specification. The gain in the passband of the shelving filter is increased by **Boost/cut gain** dB from a *floor* gain of 0 dB.

Algorithm

Design method

Select the design method from the drop-down list. Different IIR design methods are available depending on the filter constraints you specify.

Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

Filter implementation

Structure

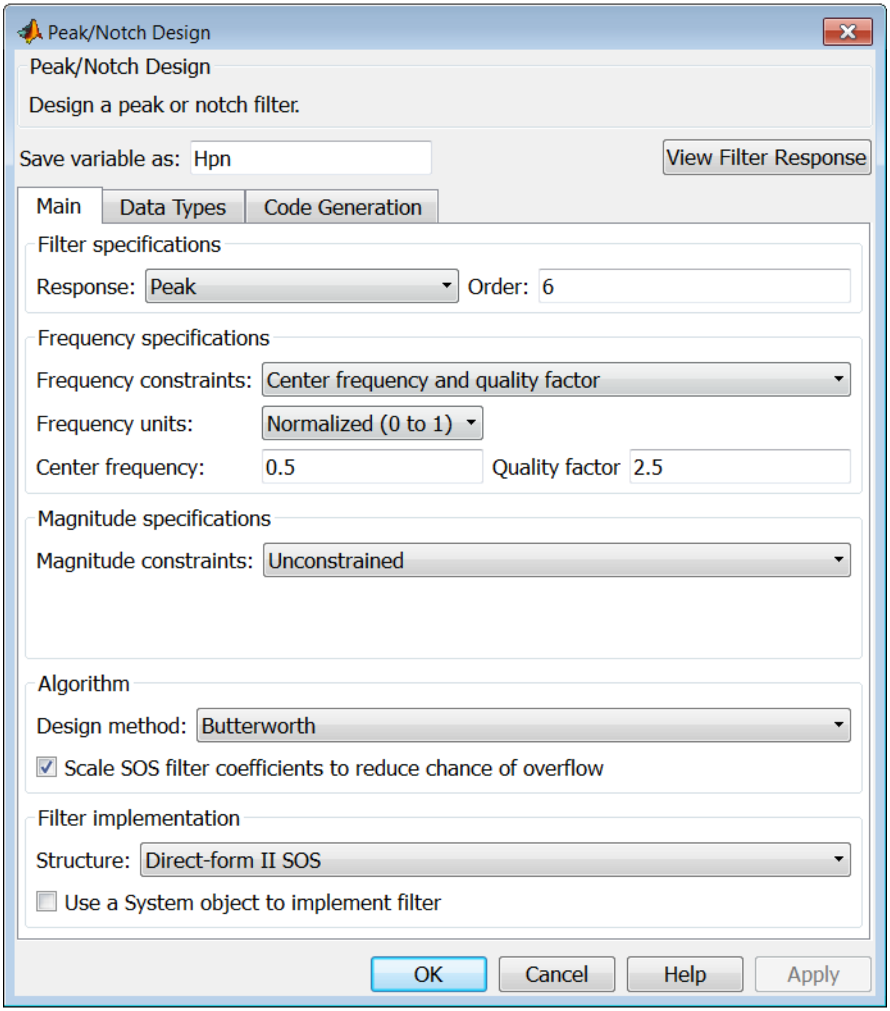
Select filter structure. The possible choices are:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

Use a System object to implement filter

Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared. When the current design method or structure is not supported by a System object filter, then this check box is disabled.

Peak/Notch Filter Design — Main Pane



Filter specifications

In this area you can specify whether you want to design a peaking filter or a notching filter, as well as the order of the filter.

Response

Select `Peak` or `Notch` from the drop-down box.

Order

Enter the filter order. The order must be even.

Frequency specifications

This group of parameters allows you to specify frequency constraints and units.

Frequency Constraints

Select the frequency constraints for filter specification. There are two choices as follows:

- `Center frequency and quality factor`
- `Center frequency and bandwidth`

Frequency units

The frequency units are normalized by default. If you specify units other than normalized, `filterBuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: `Normalized (0 to 1), Hz, kHz, MHz, GHz`.

Input Fs

This input box is enabled if **Frequency units** other than `Normalized (0 to 1)` are specified. Enter the input sampling frequency.

Center frequency

Enter the center frequency in the units you specified in **Frequency units**.

Quality Factor

This input box is enabled only when `Center frequency and quality factor` is chosen for the **Frequency Constraints**. Enter the quality factor.

Bandwidth

This input box is enabled only when `Center frequency and bandwidth` is chosen for the **Frequency Constraints**. Enter the bandwidth.

Magnitude specifications

This group of parameters allows you to specify the magnitude constraints, as well as their values and units.

Magnitude Constraints

Depending on the choice of constraints, the other input boxes are enabled or disabled. Select from four magnitude constraints available:

- `Unconstrained`
- `Passband ripple`
- `Stopband attenuation`
- `Passband ripple and stopband attenuation`

Magnitude units

Select the magnitude units: either `dB` or `squared`.

Apass

This input box is enabled if the magnitude constraints selected are `Passband ripple` or `Passband ripple and stopband attenuation`. Enter the passband ripple.

Astop

This input box is enabled if the magnitude constraints selected are `Stopband attenuation` or `Passband ripple and stopband attenuation`. Enter the stopband attenuation.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterBuilder` uses to implement your filter.

Design Method

Lists all design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter the methods available to design filters changes as well.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Filter implementation

Structure

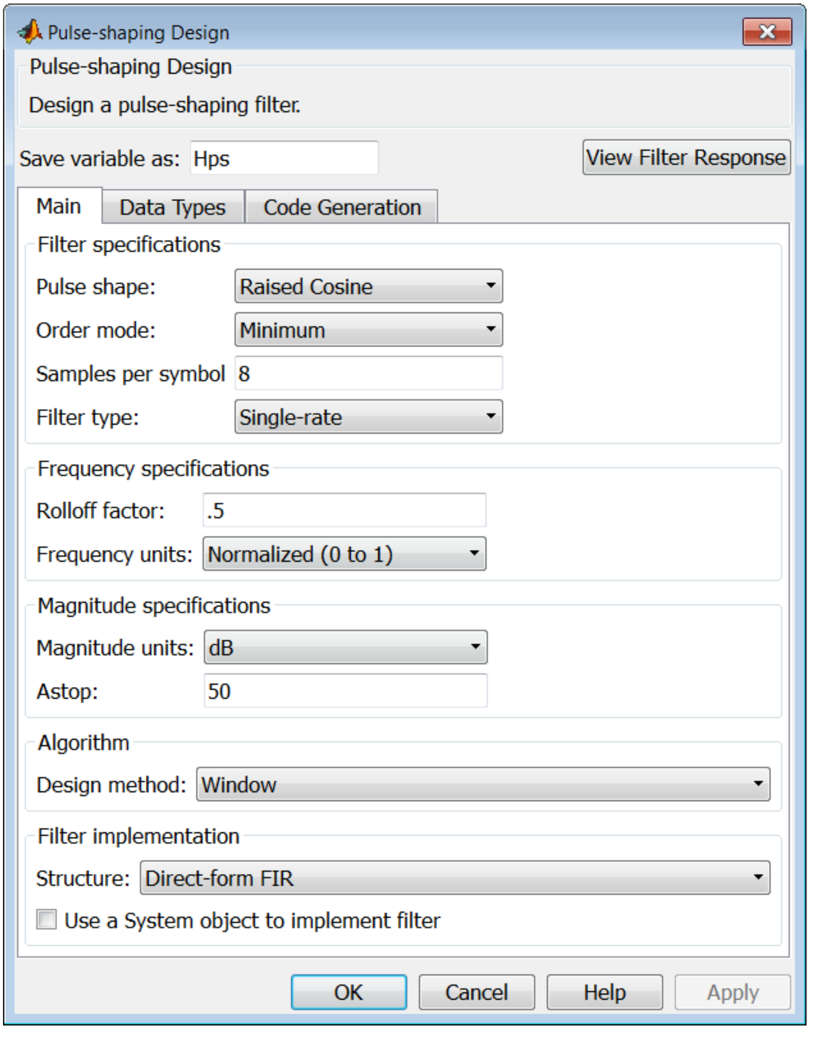
Lists all available filter structures for the filter specifications and design method you select. The typical options are:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

Use a System object to implement filter

Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared. When the current design method or structure is not supported by a System object filter, then this check box is disabled.

Pulse-shaping Filter Design —Main Pane



Filter specifications

Parameters in this group enable you to specify the shape and length of the filter.

Pulse shape

Select the shape of the impulse response from the following options:

- `Raised Cosine`
- `Square Root Raised Cosine`
- `Gaussian`

Order mode

This specification is only available for raised cosine and square root raised cosine filters. For these filters, select one of the following options:

- `Minimum`— This option will result in the minimum-length filter satisfying the user-specified **Frequency specifications**.
- `Specify order`—This option allows the user to construct a raised cosine or square root cosine filter of a specified order by entering an even number in the **Order** input box. The length of the impulse response will be `Order+1` .
- `Specify symbols`—This option enables the user to specify the length of the impulse response in an alternative manner. If `Specify symbols` is chosen, the **Order** input box changes to the **Number of symbols** input box.

Samples per symbol

Specify the oversampling factor. Increasing the oversampling factor guards against aliasing and improves the FIR filter approximation to the ideal frequency response. If **Order** is specified in **Number of symbols**, the filter length will be **Number of symbols*Samples per symbol**+1. The product **Number of symbols*Samples per symbol** must be an even number.

If a Gaussian filter is specified, the filter length must be specified in **Number of symbols** and **Samples per symbol**. The product **Number of symbols*Samples per symbol** must be an even number. The filter length will be **Number of symbols*Samples per symbol**+1.

Filter Type

This option is only available if you have the DSP System Toolbox software. Choose `Single rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. If you select `Decimator` or `Interpolator`, the decimation and interpolation factors default to the value of the **Samples per symbol**. If you select `Sample-rate converter`, the interpolation factor defaults to **Samples per symbol** and the decimation factor defaults to 3.

Frequency specifications

Parameters in this group enable you to specify the frequency response of the filter. For raised cosine and square root raised cosine filters, the frequency specifications include:

Rolloff factor

The rolloff factor takes values in the range [0,1]. The smaller the rolloff factor, the steeper the transition in the stopband.

Frequency units

The frequency units are normalized by default. If you specify units other than normalized, `filterBuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: `Normalized (0 to 1)`, `Hz`, `kHz`, `MHz`, `GHz`

For a Gaussian pulse shape, the available frequency specifications are:

Bandwidth-time product

This option allows the user to specify the width of the Gaussian filter. Note that this is independent of the length of the filter. The bandwidth-time product (BT) must be a positive real number. Smaller values of the bandwidth-time product result in larger pulse widths in time and steeper stopband transitions in the frequency response.

Frequency units

The frequency units are normalized by default. If you specify units other than normalized, `filterBuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: `Normalized (0 to 1)`, `Hz`, `kHz`, `MHz`, `GHz`

Magnitude specifications

If the **Order mode** is specified as `Minimum`, the **Magnitude units** may be selected from:

- `dB`— Specify the magnitude in decibels (default).
- `Linear`— Specify the magnitude in linear units.

Algorithm

The only **Design method** available for FIR pulse-shaping filters is the `Window` method.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure.

Use a System object to implement filter

This check box appears when you set **Filter type** to `Single-rate`. Selecting this check box gives you the choice of using a System object to implement the filter. By default, the check box is cleared.

This check box no longer appears when you set **Filter type** to `Interpolator`, `Decimator`, or `Sample-rate converter`. The filter builder always implements the filter as a System object.

Introduced in R2009a

filtstates.cic

Store CIC filter states

Description

`filtstates.cic` objects hold the states information for CIC filters. Once you create a CIC filter, the states for the filter are stored in `filtstates.cic` objects, and you can access them and change them as you would any property of the filter. This arrangement parallels that of the `filtstates` object that IIR direct-form I filters use (refer to `filtstates` for more information).

Each `States` property in the CIC filter comprises two properties — `Numerator` and `Comb` — that hold `filtstates.cic` objects. Within the `filtstates.cic` objects are the numerator-related and comb-related filter states. The states are column vectors, where each column represents the states for one section of the filter. For example, a CIC filter with four decimator sections and four interpolator sections has `filtstates.cic` objects that contain four columns of states each.

Examples

Integrator and Comb States of a CIC Filter

Construct an object with integrator and comb states as vectors of zeros.

```
h = filtstates.cic(zeros(4,1),zeros(4,1));
```

h has zero states now. You can use int to see the states as 32-bit integers.

```
intStates = int(h.Integrator)
combStates = int(h.Comb)
```

```
intStates =

  4×1 int32 column vector

   0
   0
```

```
   0
   0


combStates =

  4×1 int32 column vector

   0
   0
   0
   0
```

See Also
dsp.CICInterpolator | dsp.CICDecimator | `filtstates`

Introduced in R2011a

fircband

Constrained-band equiripple FIR filter

Syntax

```
b = fircband(n,f,a,w,c)
b = fircband(n,f,a,s)
b = fircband(...,'1')
b = fircband(...,'minphase')
b = fircband(..., 'check')
b = fircband(...,{lgrid})
[b,err] = fircband(...)
[b,err,res] = fircband(...)
```

Description

`fircband` is a minimax filter design algorithm that you use to design the following types of real FIR filters:

- Types 1-4 linear phase
  - Type 1 is even order, symmetric
  - Type 2 is odd order, symmetric
  - Type 3 is even order, antisymmetric
  - Type 4 is odd order, antisymmetric
- Minimum phase
- Maximum phase
- Minimum order (even or odd), extra ripple
- Maximal ripple
- Constrained ripple
- Single-point band (notching and peaking)
- Forced gain
- Arbitrary shape frequency response curve filters

`b = fircband(n,f,a,w,c)` designs filters having constrained error magnitudes (ripples). `c` is a cell array of character vectors of the same length as `w`. The entries of `c` must be either `'c'` to indicate that the corresponding element in `w` is a constraint (the ripple for that band cannot exceed that value) or `'w'` indicating that the corresponding entry in `w` is a weight. There must be at least one unconstrained band — `c` must contain at least one **`w`** entry. For instance, the example 'Design a Constrained Lowpass Filter' uses a weight of one in the passband, and constrains the stopband ripple not to exceed 0.2 (about 14 dB).

A hint about using constrained values: if your constrained filter does not touch the constraints, increase the error weighting you apply to the unconstrained bands.

Notice that, when you work with constrained stopbands, you enter the stopband constraint according to the standard conversion formula for power — the resulting filter attenuation or constraint equals 20*log(*constraint*) where *constraint* is the value you enter in the function. For example, to set 20 dB of attenuation, use a value for the constraint equal to 0.1. This applies to constrained stopbands only.

`b = fircband(n,f,a,s)` is used to design filters with special properties at certain frequency points. `s` is a cell array of character vectors and must be the same length as `f` and `a`. Entries of `s` must be one of:

- `'n'` — normal frequency point.
- `'s'` — single-point band. The frequency band is given by a single point. You must specify the corresponding gain at this frequency point in `a`.
- `'f'` — forced frequency point. Forces the gain at the specified frequency band to be the value specified.
- `'i'` — indeterminate frequency point. Use this argument when bands abut one another (no transition region).

`b = fircband(...,'1')` designs a type 1 filter (even-order symmetric). You could also specify type 2 (odd-order symmetric), type 3 (even-order antisymmetric), or type 4 (odd-order antisymmetric) filters. Note there are restrictions on `a` at `f` = 0 or `f` = 1 for types 2, 3, and 4.

`b = fircband(...,'minphase')` designs a minimum-phase FIR filter. There is also `'maxphase'`.

`b = fircband(..., 'check')` produces a warning when there are potential transition-region anomalies in the filter response.

`b = fircband(...,{lgrid})`, where {`lgrid`} is a scalar cell array containing an integer, controls the density of the frequency grid.

`[b,err] = fircband(...)` returns the unweighted approximation error magnitudes. `err` has one element for each independent approximation error.

`[b,err,res] = fircband(...)` returns a structure `res` of optional results computed by `fircband`, and contains the following fields:

Structure Field	Contents
`res.fgrid`	Vector containing the frequency grid used in the filter design optimization
`res.des`	Desired response on `fgrid`
`res.wt`	Weights on `fgrid`
`res.h`	Actual frequency response on the frequency grid
`res.error`	Error at each point (desired response - actual response) on the frequency grid
`res.iextr`	Vector of indices into `fgrid` of external frequencies
`res.fextr`	Vector of extremely frequencies
`res.order`	Filter order
`res.edgecheck`	Transition-region anomaly check. One element per band edge. Element values have the following meanings: 1 = OK , 0 = probable transition-region anomaly , -1 = edge not checked. Computed when you specify the `'check'` input option in the function syntax.
`res.iterations`	Number of Remez iterations for the optimization
`res.evals`	Number of function evaluations for the optimization

Examples

Design a Constrained Lowpass Filter

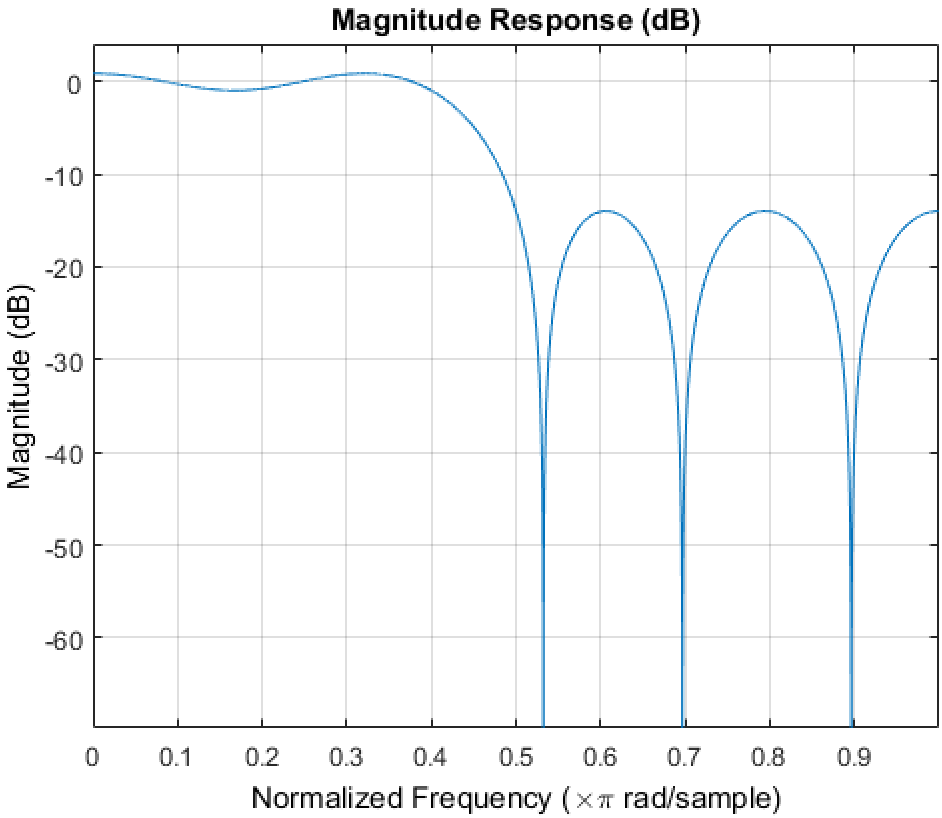
Design a 12th-order lowpass filter with a constraint on the filter response.

```
b = fircband(12,[0 0.4 0.5 1], [1 1 0 0], ...
```

```
[1 0.2], {'w' 'c'});
```

Using fvtool to display the result b shows you the response of the filter you designed.

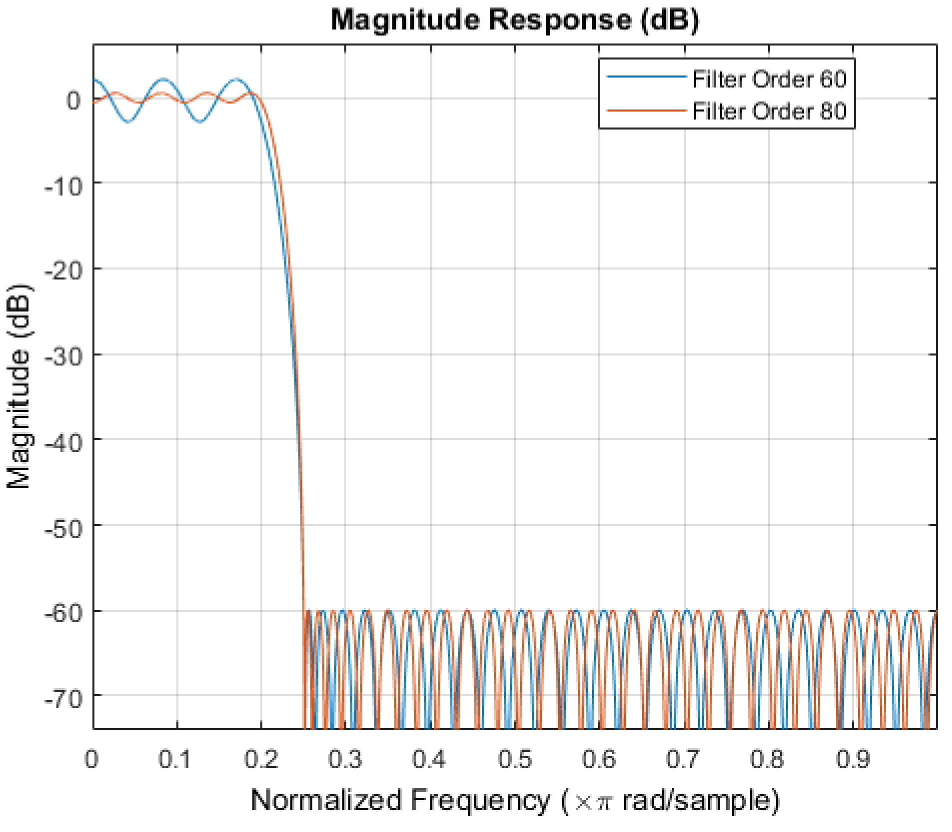
```
fvtool(b)
```



Design two filters of different order with the stopband constrained to 60 dB. Use excess order (80) in the second filter to improve the passband ripple.

```
b1 = fircband(60,[0 .2 .25 1],[1 1 0 0],...
[1 .001],{'w','c'});
b2 = fircband(80,[0 .2 .25 1],[1 1 0 0],...
[1 .001],{'w','c'});
hfvt = fvtool(b1,1,b2,1);
```

```
legend(hfvt,'Filter Order 60','Filter Order 80');
```



See Also

firceqrip | firgr | firls | firpm

Introduced in R2011a

fireqint

Equiripple FIR interpolators

Syntax

```
b = fireqint(n,l,alpha)
b = fireqint(n,l,alpha,w)
b = fireqint('minorder',l,alpha,r)
b = fireqint({'minorder',initord},l,alpha,r)
```

Description

`b = fireqint(n,l,alpha)` designs an FIR equiripple filter useful for interpolating input signals. `n` is the filter order and it must be an integer. `l`, also an integer, is the interpolation factor. `alpha` is the bandlimitedness factor, identical to the same feature in `intfilt`.

`alpha` is inversely proportional to the transition bandwidth of the filter. It also affects the bandwidth of the don't-care regions in the stopband. Specifying `alpha` allows you to control how much of the Nyquist interval your input signal occupies. This can be beneficial for signals to be interpolated because it allows you to increase the transition bandwidth without affecting the interpolation, resulting in better stopband attenuation for a given `l`. If you set `alpha` to 1, `fireqint` assumes that your signal occupies the entire Nyquist interval. Setting `alpha` to a value less than one allows for don't-care regions in the stopband. For example, if your input occupies half the Nyquist interval, you could set `alpha` to 0.5.

The signal to be interpolated is assumed to have zero (or negligible) power in the frequency band between (`alpha*π`) and π. `alpha` must therefore be a positive scalar between 0 and 1. `fireqint` treat such bands as don't-care regions for assessing filter design.

`b = fireqint(n,l,alpha,w)` allows you to specify a vector of weights in `w`. The number of weights required in `w` is given by 1 + `floor`(l/2). The weights in `w` are applied to the passband ripple and stopband attenuations. Using weights (values between 0 and 1) enables you to specify different attenuations in different parts of the stopband, as well as providing the ability to adjust the compromise between passband ripple and stopband attenuation.

`b = fireqint('minorder',l,alpha,r)` allows you to design a minimum-order filter that meets the design specifications. `r` is a vector of maximum deviations or ripples from the ideal filter magnitude response. When you use the input argument **`minorder`**, you must provide the vector `r`. The number of elements required in `r` is given by 1 + `floor`(`l`/2).

`b = fireqint({'minorder',initord},l,alpha,r)` adds the argument `initord` so you can provide an initial estimate of the filter order. Any positive integer is valid here. Again, you must provide `r`, the vector of maximum deviations or ripples, from the ideal filter magnitude response.

Examples

Design an FIR Equiripple Filter

Design a minimum order interpolation filter with `l` set to 6, and `alpha` set to 0.8. A vector of ripples must be supplied with the input argument `minorder`.

```
b = fireqint('minorder',6,.8,[0.01 .1 .05 .02]);
```

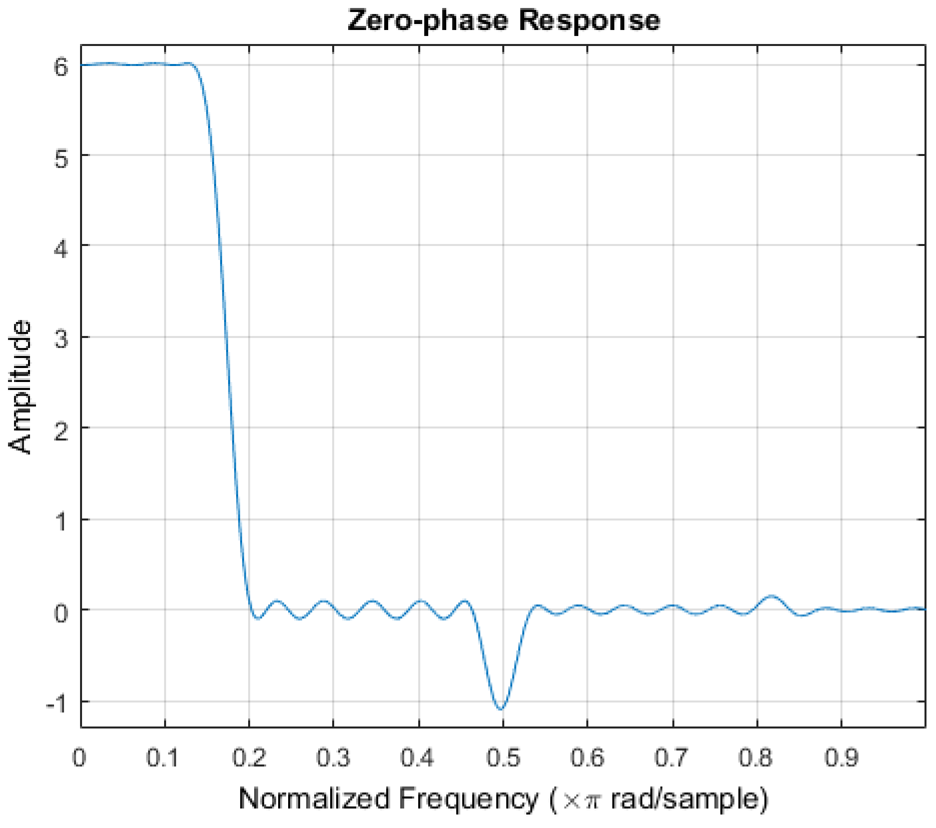
Create a polyphase interpolation filter.

```
hm = dsp.FIRInterpolator(6,'Numerator',b)
```

```
hm = 

  dsp.FIRInterpolator with properties:

        NumeratorSource: 'Property'
              Numerator: [1×70 double]
    InterpolationFactor: 6

  Use get to show all properties
```

Here is the zerophase response of the interpolator.

```
zerophase(hm);
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

`firgr` | `firhalfband` | `firls` | `firnyquist` | `intfilt`

Introduced in R2011a

firceqrip

Constrained equiripple FIR filter

Syntax

```
B = firceqrip(n,Fo,DEV)
B = firceqrip(...,'slope',r)
B = firceqrip(...,'passedge')
B = firceqrip(...,'stopedge')
B = firceqrip(...,'high')
B = firceqrip(...,'min')
B = firceqrip(...,'invsinc',C)
B = firceqrip(...,'invdiric',C)
```

Description

`B = firceqrip(n,Fo,DEV)` designs an order `n` filter (filter length equal `n` + 1) lowpass FIR filter with linear phase.

`firceqrip` produces the same equiripple lowpass filters that `firpm` produces using the Parks-McClellan algorithm. The difference is how you specify the filter characteristics for the function.

The input argument `Fo` specifies the frequency at the upper edge of the passband in normalized frequency (0<`Fo`<1). The two-element vector `dev` specifies the peak or maximum error allowed in the passband and stopbands. Enter `[d1 d2]` for `dev` where `d1` sets the passband error and `d2` sets the stopband error.

`B = firceqrip(...,'slope',r)` uses the input keyword '`slope`' and input argument `r` to design a filter with a nonequiripple stopband. `r` is specified as a positive constant and determines the slope of the stopband attenuation in dB/normalized frequency. Greater values of `r` result in increased stopband attenuation in dB/normalized frequency.

`B = firceqrip(...,'passedge')` designs a filter where `Fo` specifies the frequency at which the passband starts to rolloff.

`B = firceqrip(...,'stopedge')` designs a filter where `Fo` specifies the frequency at which the stopband begins.

`B = firceqrip(...,'high')` designs a high pass FIR filter instead of a lowpass filter.

`B = firceqrip(...,'min')` designs a minimum-phase filter.

`B = firceqrip(...,'invsinc',C)` designs a lowpass filter whose magnitude response has the shape of an inverse sinc function. This may be used to compensate for sinc-like responses in the frequency domain such as the effect of the zero-order hold in a D/A converter. The amount of compensation in the passband is controlled by `C`, which is specified as a scalar or two-element vector. The elements of `C` are specified as follows:

- If `C` is supplied as a real-valued scalar or the first element of a two-element vector, `firceqrip` constructs a filter with a magnitude response of 1/sinc(`C*`**`pi`**`*F`) where `F` is the normalized frequency.
- If `C` is supplied as a two-element vector, the inverse-sinc shaped magnitude response is raised to the positive power `C(2)`. If we set `P=C(2)`, `firceqrip` constructs a filter with a magnitude response 1/sinc(`C*`**`pi`**`*F`)^{P}.

If this FIR filter is used with a cascaded integrator-comb (CIC) filter, setting `C(2)` equal to the number of stages compensates for the multiplicative effect of the successive sinc-like responses of the CIC filters.

Note: Since the value of the inverse sinc function becomes unbounded at `C=1/F`, the value of `C` should be greater the reciprocal of the passband edge frequency. This can be expressed as `Fo<1/C`. For users familiar with CIC decimators, `C` is equal to 1/2 the product of the differential delay and decimation factor.

`B = firceqrip(...,'invdiric',C)` designs a lowpass filter with a passband that has the shape of an inverse Dirichlet sinc function. The frequency response of the inverse Dirichlet sinc function is given by

$$\{rC(\frac{\sin(f/2r)}{\sin(Cf/2)}\}^{p}$$

where C, r, and p are scalars. The input `C` can be a scalar or vector containing 2 or 3 elements. If `C` is a scalar, `p` and `r` equal 1. If `C` is a two-element vector, the first element is `C` and the second element is `p`, `[C p]`. If `C` is a three-element vector, the third element is `r`, `[C p r]`.

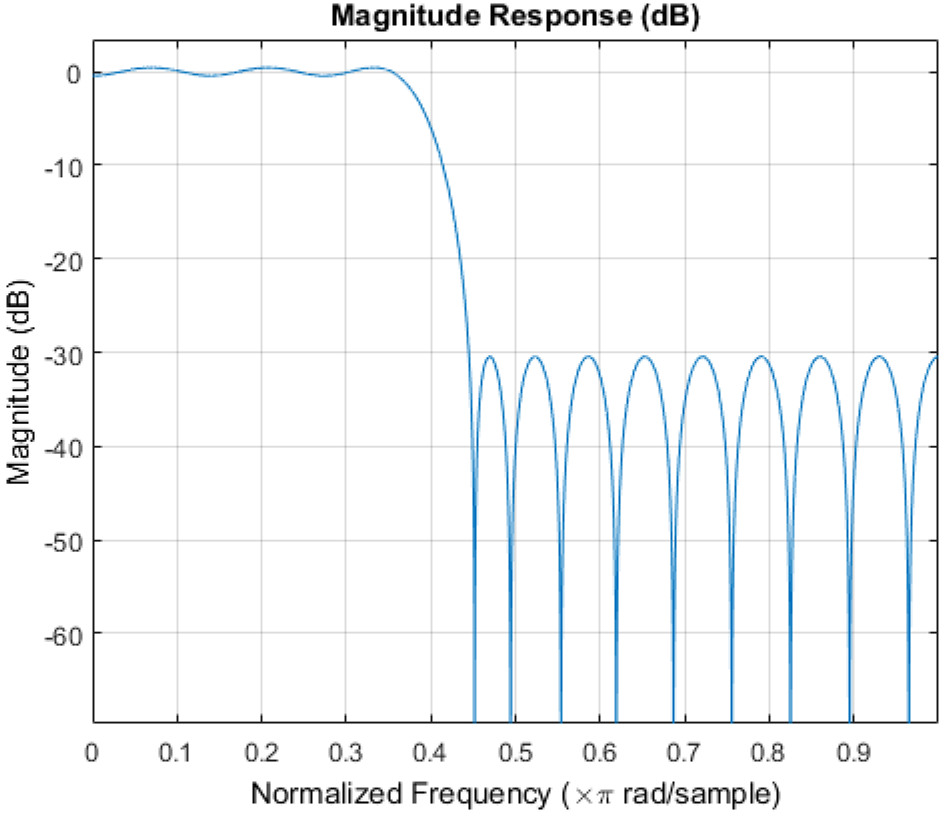
Examples

To introduce a few of the variations on FIR filters that you design with `firceqrip`, these five examples cover both the default syntax `b = firceqrip(n,wo,del)` and some of the optional input arguments. For each example, the input arguments `n`, `wo`, and `del` remain the same.

Filter design using `firceqrip`

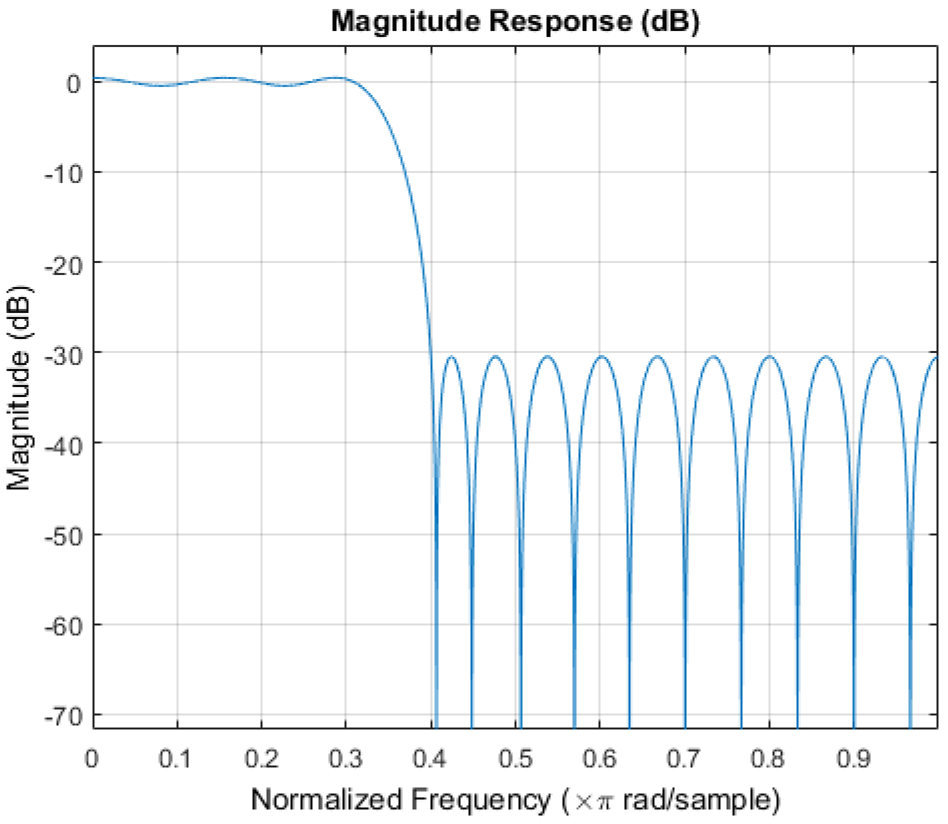
Design a 30th order FIR filter using `firceqrip`.

```
b = firceqrip(30,0.4,[0.05 0.03]); fvtool(b)
```



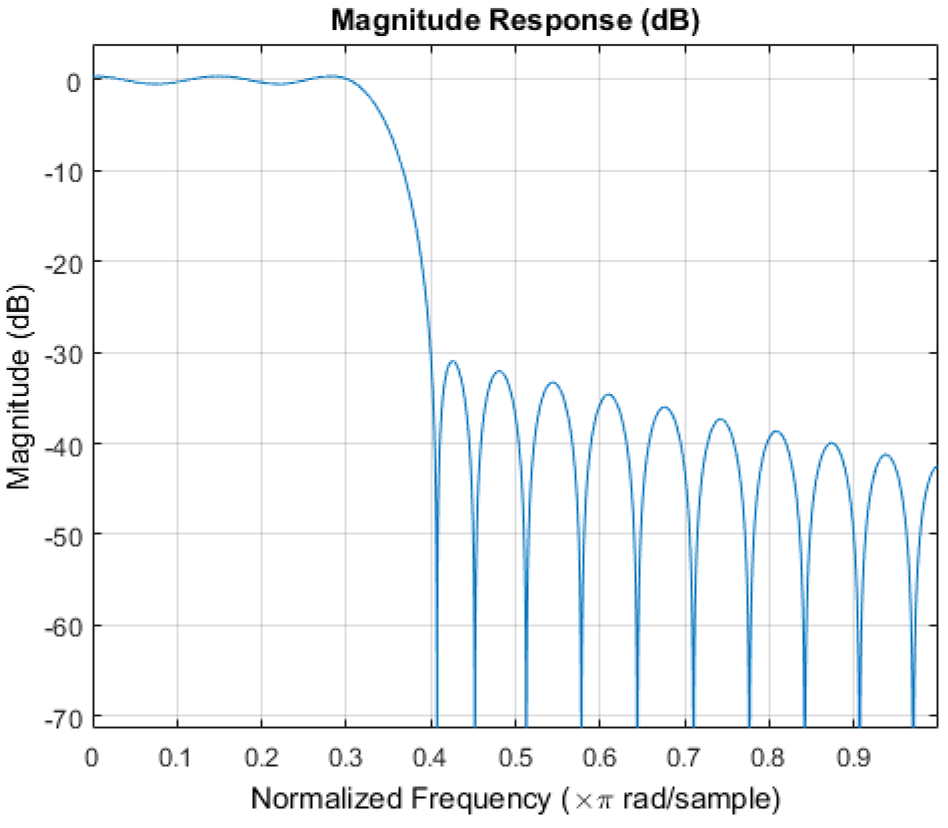
Design a 30th order FIR filter with the `stopedge` keyword to define the response at the edge of the filter stopband.

```
b = firceqrip(30,0.4,[0.05 0.03],'stopedge'); fvtool(b)
```



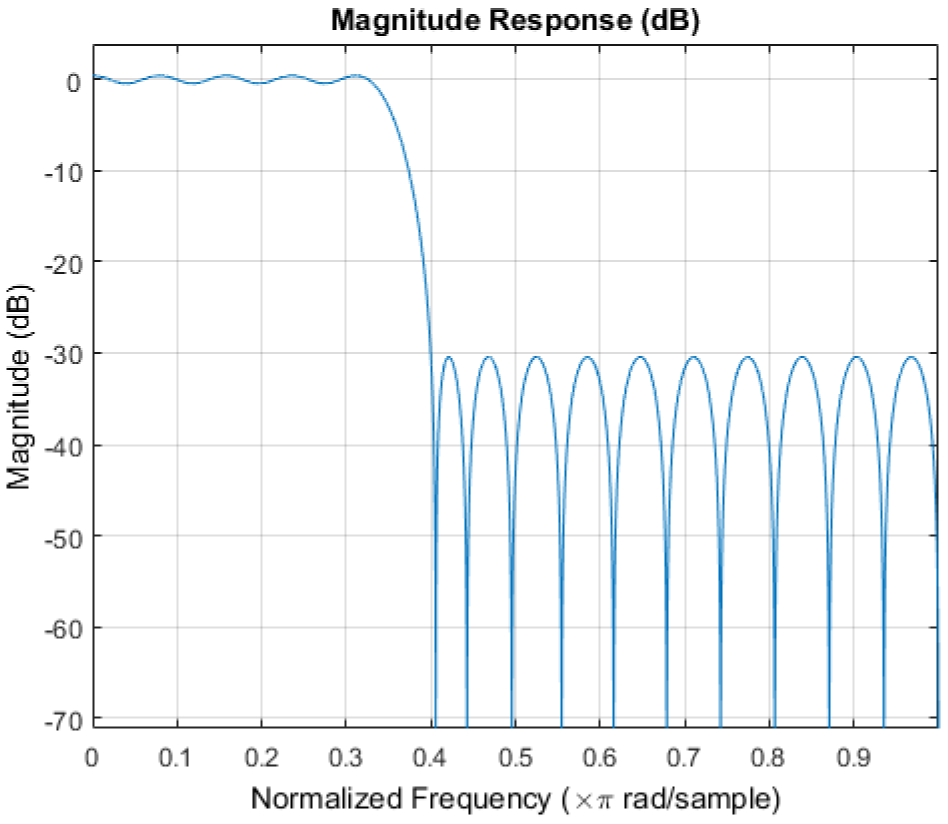
Design a 30th order FIR filter with the `slope` keyword and r = 20.

```
b = firceqrip(30,0.4,[0.05 0.03],'slope',20,'stopedge'); fvtool(b)
```



Design a 30th order FIR filter defining the stopband and specifying that the resulting filter is minimum phase with the `min` keyword.

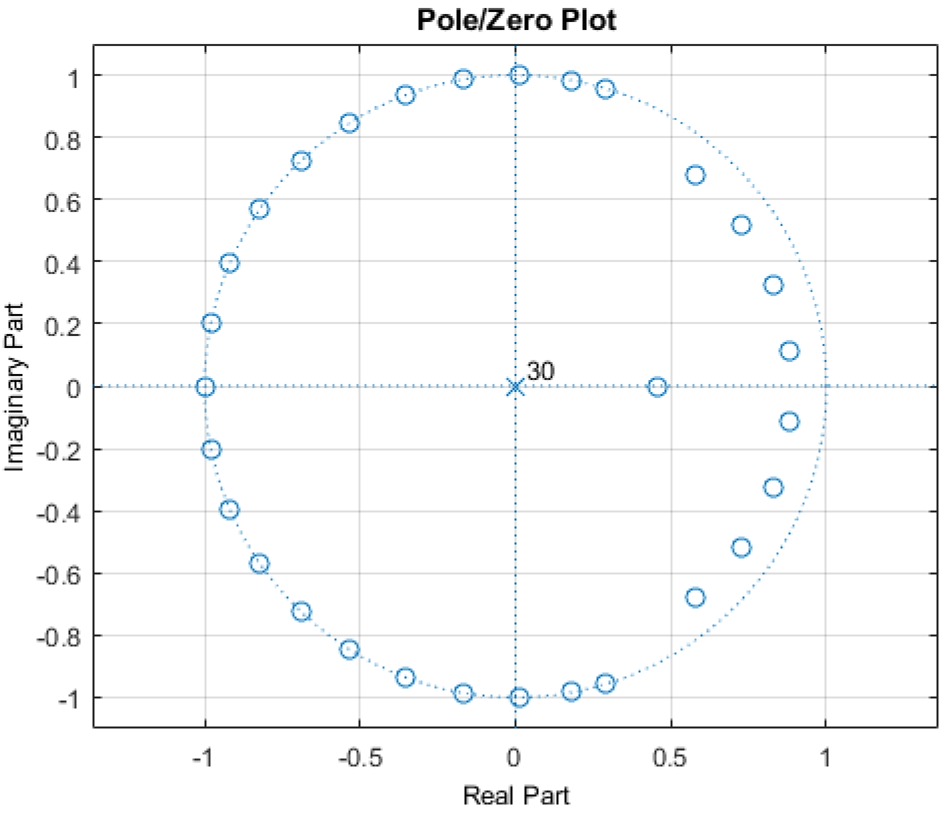
```
b = firceqrip(30,0.4,[0.05 0.03],'stopedge','min'); fvtool(b)
```



Comparing this filter to the filter in Figure 1. The cutoff frequency `wo = 0.4` now applies to the edge of the stopband rather than the point at which the frequency response magnitude is 0.5.

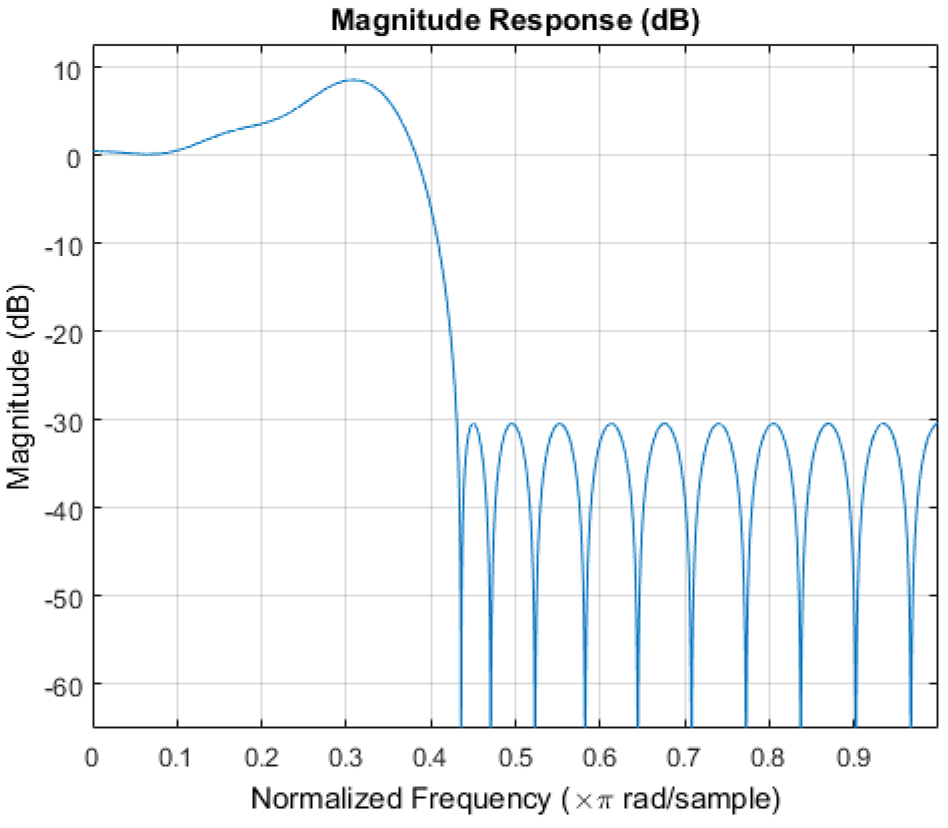
Viewing the zero-pole plot shown here reveals this is a minimum phase FIR filter - the zeros lie on or inside the unit circle, z = 1

```
fvtool(b,'polezero')
```



Design a 30th order FIR filter with the `invsinc` keyword to shape the filter passband with an inverse sinc function.

```
b = firceqrip(30,0.4,[0.05 0.03],'invsinc',[2 1.5]); fvtool(b)
```



The inverse sinc function being applied is defined as 1/sinc(2*w)^1.5.

Inverse-Dirichlet-Sinc-Shaped Passband

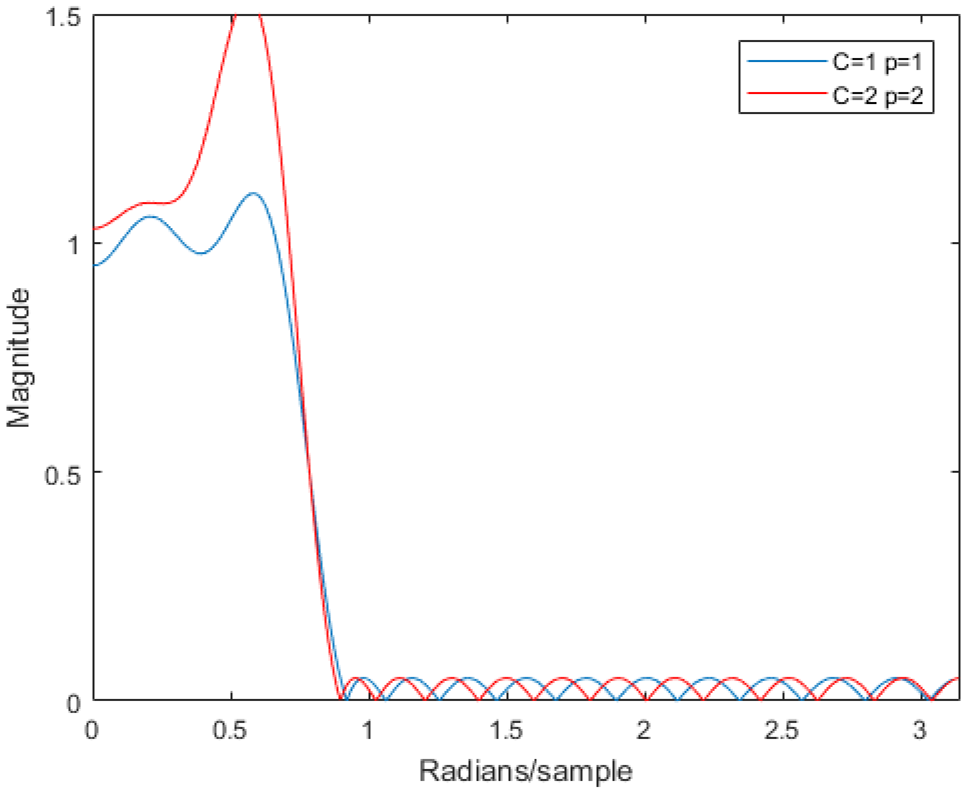
Design two order 30 constrained equiripple FIR filters with inverse-Dirichlet-sinc-shaped passbands. The cutoff frequency in both designs is pi/4 radians/sample. Set `C=1` in one design `C=2` in the second design. The maximum passband and stopband ripple is 0.05. Set `p=1` in one design and `p=2` in the second design.

Design the filters.

```
b1 = firceqrip(30,0.25,[0.05 0.05],'invdiric',[1 1]);
b2 = firceqrip(30,0.25,[0.05 0.05],'invdiric',[2 2]);
```

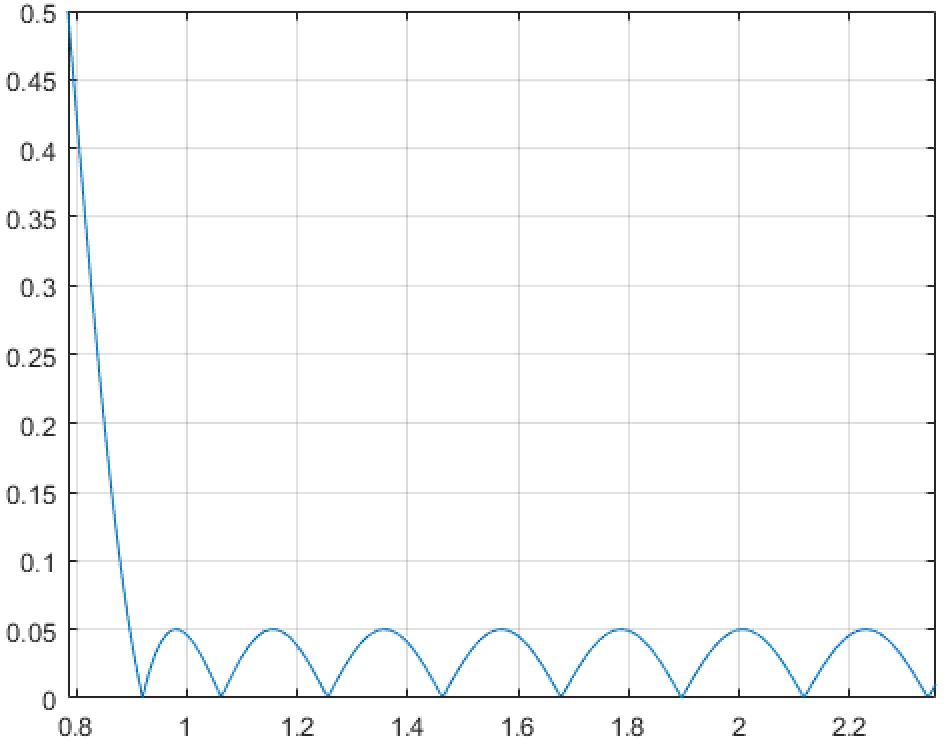
Obtain the filter frequency responses using `freqz`. Plot the magnitude responses.

```
[h1,~] = freqz(b1,1);
[h2,w] = freqz(b2,1);
plot(w,abs(h1)); hold on;
plot(w,abs(h2),'r');
axis([0 pi 0 1.5]);
xlabel('Radians/sample');
ylabel('Magnitude');
legend('C=1 p=1','C=2 p=2');
```



Inspect the stopband ripple in the design with `C=1` and `p=1`. The constrained design sets the maximum ripple to be 0.05. Zoom in on the stopband from the cutoff frequency of pi/4 radians/sample to 3pi/4 radians/sample.

```
figure;
plot(w,abs(h1));
set(gca,'xlim',[pi/4 3*pi/4]);
grid on;
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

`diric` | `fdesign.decimator` | `firhalfband` | `firnyquist` | `firgr` | `ifir` | `iirgrpdelay` | `iirlpnorm` | `iirlpnormc` | `fircls` | `firls` | `firpm` | `sinc`

Introduced in R2011a

fircls

FIR Constrained Least Squares filter

Syntax

```
clsFilter = design(d,'fircls','SystemObject',true)
clsFilter =
design(d,'fircls','FilterStructure',value,'SystemObject',true)
clsFilter =
design(d,'fircls','PassbandOffset',value,'SystemObject',true)
clsFilter = design(d,'fircls','zerophase',value,'SystemObject',true)
```

Description

`clsFilter = design(d,'fircls','SystemObject',true)` designs a FIR Constrained Least Squares (CLS) filter, `clsFilter`, from a filter specifications object, `d`.

`clsFilter = design(d,'fircls','FilterStructure',value,'SystemObject',true)` where `value` is one of the following filter structures:

- `'dffir'`, a discrete-time, direct-form FIR filter (the default value)
- `'dffirt'`, a discrete-time, direct-form FIR transposed filter
- `'dfsymfir'`, a discrete-time, direct-form symmetric FIR filter

`clsFilter = design(d,'fircls','PassbandOffset',value,'SystemObject',true)` where `value` sets the passband band gain in dB. The `PassbandOffset` and `Ap` values affect the upper and lower approximation bound in the passband as follows:

- Lower bound = (`PassbandOffset`-Ap/2)
- Upper bound = (`PassbandOffset`+A/2)

For bandstop filters, the `PassbandOffset` is a vector of length two that specifies the first and second passband gains. The `PassbandOffset` value defaults to 0 for lowpass, highpass and bandpass filters. The `PassbandOffset` value defaults to [0 0] for bandstop filters.

`clsFilter = design(d,'fircls','zerophase',value,'SystemObject',true)` where `value` is either `'true'` ('1') or `'false'` ('0') . If `zerophase` is true, the lower approximation bound in the stopband is forced to zero (i.e., the filter has a zero phase response). Zerophase is false (0) by default.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'fircls')
```

For complete help about using `fircls`, refer to the command line help system. For example, to get specific information about using `fircls` with `d`, the specification object, enter the following at the MATLAB prompt.

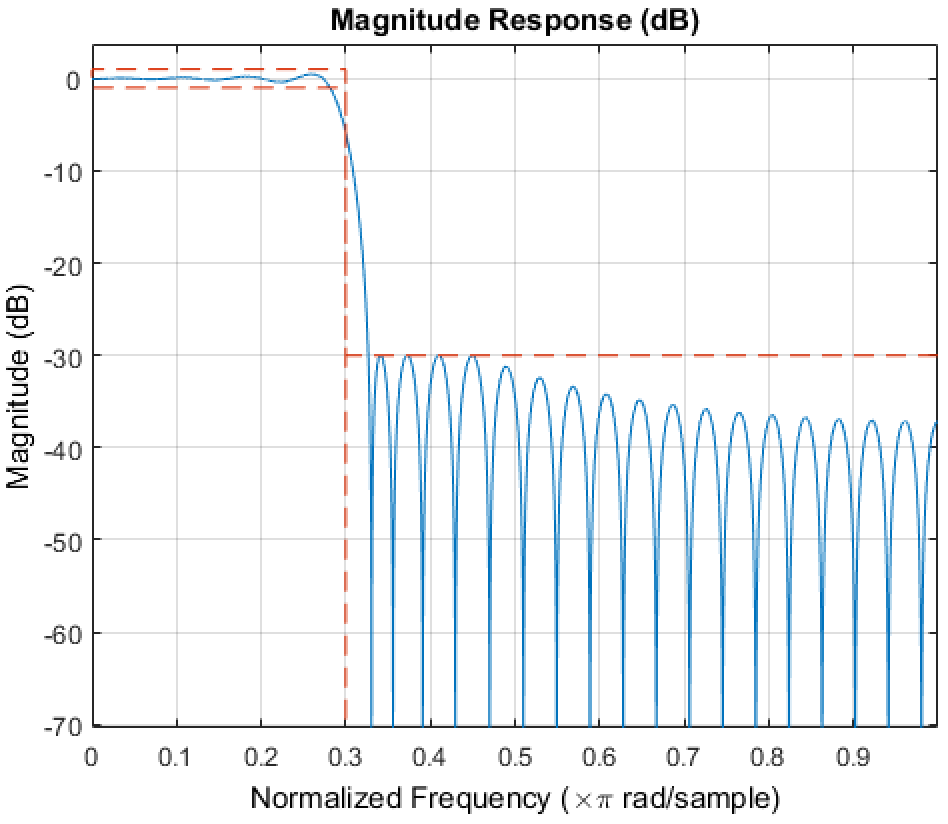
```
help(d,'fircls')
```

Examples

Create an FIR Constrained Least Squares (CLS) Filter

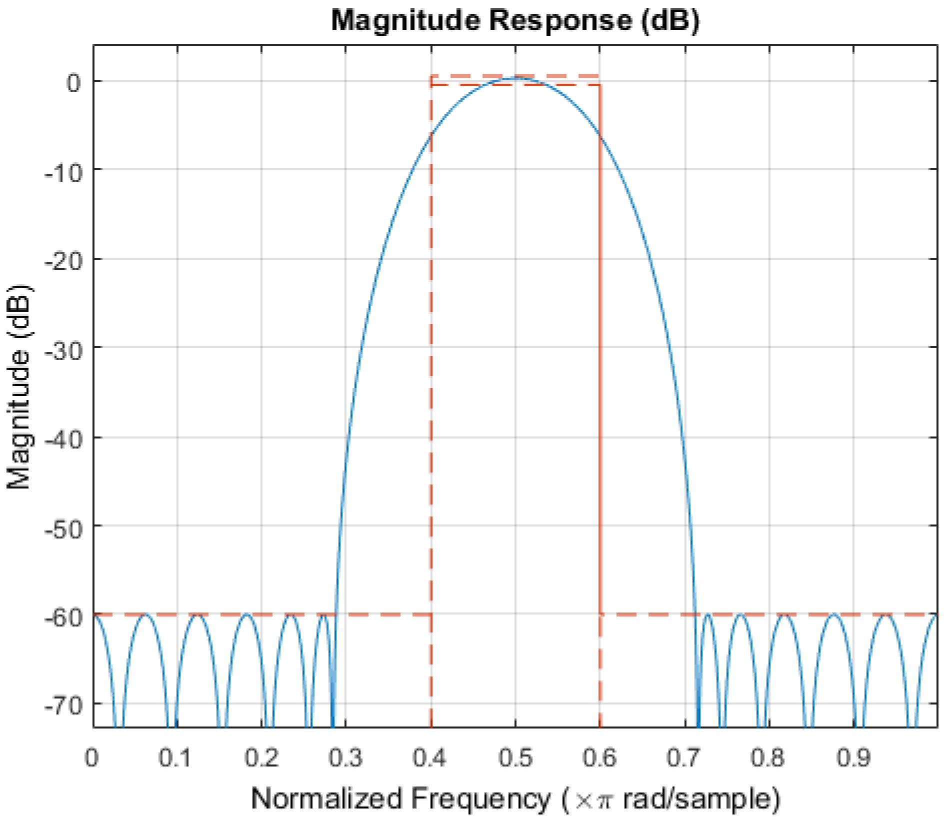
The following example designs a constrained least-squares FIR lowpass filter.

```
h  = fdesign.lowpass('n,fc,ap,ast', 50, 0.3, 2, 30);
firLPFilter = design(h, 'fircls','SystemObject',true);
fvtool(firLPFilter)
```



The following example constructs a constrained least-squares FIR bandpass filter.

```
d = fdesign.bandpass('N,Fc1,Fc2,Ast1,Ap,Ast2',...
30,0.4,0.6,60,1,60);
firBPFilter = design(d,'fircls','SystemObject',true);
fvtool(firBPFilter)
```



See Also

`cheby1` | `cheby2` | `ellip`

Introduced in R2011a

firgr

Parks-McClellan FIR filter

Syntax

```
b = firgr(n,f,a,w)
b = firgr(n,f,a,'hilbert')
b = firgr(m,f,a,r),
b = firgr({m,ni},f,a,r)
b = firgr(n,f,a,w,e)
b = firgr(n,f,a,s)
b = firgr(n,f,a,s,w,e)
b = firgr(...,'1')
b = firgr(...,'minphase')
b = firgr(..., 'check')
b = firgr(...,{lgrid}),
[b,err] = firgr(...)
[b,err,res] = firgr(...)
b = firgr(n,f,fresp,w)
b = firgr(n,f,{fresp,p1,p2,...},w)
b = firgr(n,f,a,w)
```

Description

`firgr` is a minimax filter design algorithm you use to design the following types of real FIR filters:

- Types 1-4 linear phase:
  - Type 1 is even order, symmetric
  - Type 2 is odd order, symmetric
  - Type 3 is even order, antisymmetric
  - Type 4 is odd order, antisymmetric
- Minimum phase
- Maximum phase

- Minimum order (even or odd)
- Extra ripple
- Maximal ripple
- Constrained ripple
- Single-point band (notching and peaking)
- Forced gain
- Arbitrary shape frequency response curve filters

`b = firgr(n,f,a,w)` returns a length `n+1` linear phase FIR filter which has the best approximation to the desired frequency response described by `f` and `a` in the minimax sense. `w` is a vector of weights, one per band. When you omit `w`, all bands are weighted equally. For more information on the input arguments, refer to `firpm` in *Signal Processing Toolbox User's Guide*.

`b = firgr(n,f,a,'hilbert')` and `b = firgr(n,f,a,'differentiator')` design FIR Hilbert transformers and differentiators. For more information on designing these filters, refer to `firpm` in *Signal Processing Toolbox User's Guide*.

`b = firgr(m,f,a,r)`, where `m` is one of '`minorder`', '`mineven`' or '`minodd`', designs filters repeatedly until the minimum order filter, as specified in `m`, that meets the specifications is found. `r` is a vector containing the peak ripple per frequency band. You must specify `r`. When you specify '`mineven`' or '`minodd`', the minimum even or odd order filter is found.

`b = firgr({m,ni},f,a,r)` where `m` is one of '`minorder`', '`mineven`' or '`minodd`', uses `ni` as the initial estimate of the filter order. `ni` is optional for common filter designs, but it must be specified for designs in which `firpmord` cannot be used, such as while designing differentiators or Hilbert transformers.

`b = firgr(n,f,a,w,e)` specifies independent approximation errors for different bands. Use this syntax to design extra ripple or maximal ripple filters. These filters have interesting properties such as having the minimum transition width. `e` is a cell array of character vectors specifying the approximation errors to use. Its length must equal the number of bands. Entries of `e` must be in the form '`e#`' where `#` indicates which approximation error to use for the corresponding band. For example, when `e = {'e1','e2','e1'}`, the first and third bands use the same approximation error `'e1'` and the second band uses a different one `'e2'`. Note that when all bands use the same approximation error, such as `{'e1','e1','e1',...}`, it is equivalent to omitting `e`, as in `b = firgr(n,f,a,w)`.

`b = firgr(n,f,a,s)` is used to design filters with special properties at certain frequency points. `s` is a cell array of character vectors and must be the same length as `f` and `a`. Entries of `s` must be one of:

- `'n'` — normal frequency point.
- `'s'` — single-point band. The frequency "band" is given by a single point. The corresponding gain at this frequency point must be specified in `a`.
- `'f'` — forced frequency point. Forces the gain at the specified frequency band to be the value specified.
- `'i'` — indeterminate frequency point. Use this argument when adjacent bands abut one another (no transition region).

For example, the following command designs a bandstop filter with zero-valued single-point stop bands (notches) at 0.25 and 0.55.

```
b = firgr(42,[0 0.2 0.25 0.3 0.5 0.55 0.6 1],...
[1 1 0 1 1 0 1 1],{'n' 'n' 's' 'n' 'n' 's' 'n' 'n'})
```

`b = firgr(82,[0 0.055 0.06 0.1 0.15 1],[0 0 0 0 1 1],...{'n' 'i' 'f' 'n' 'n' 'n'})` designs a highpass filter with the gain at 0.06 forced to be zero. The band edge at 0.055 is indeterminate since the first two bands actually touch. The other band edges are normal.

`b = firgr(n,f,a,s,w,e)` specifies weights and independent approximation errors for filters with special properties. The weights and properties are included in vectors `w` and `e`. Sometimes, you may need to use independent approximation errors to get designs with forced values to converge. For example,

```
b = firgr(82,[0 0.055 0.06 0.1 0.15 1], [0 0 0 0 1 1],...
{'n' 'i' 'f' 'n' 'n' 'n'}, [10 1 1] ,{'e1' 'e2' 'e3'});
```

`b = firgr(...,'1')` designs a type 1 filter (even-order symmetric). You can specify type 2 (odd-order symmetric), type 3 (even-order antisymmetric), and type 4 (odd-order antisymmetric) filters as well. Note that restrictions apply to `a` at f = 0 or f = 1 for FIR filter types 2, 3, and 4.

`b = firgr(...,'minphase')` designs a minimum-phase FIR filter. You can use the argument `'maxphase'` to design a maximum phase FIR filter.

`b = firgr(..., 'check')` returns a warning when there are potential transition-region anomalies.

`b = firgr(...,{lgrid})`, where `{lgrid}` is a scalar cell array. The value of the scalar controls the density of the frequency grid by setting the number of samples used along the frequency axis.

`[b,err] = firgr(...)` returns the unweighted approximation error magnitudes. `err` contains one element for each independent approximation error returned by the function.

`[b,err,res] = firgr(...)` returns the structure `res` comprising optional results computed by `firgr`. `res` contains the following fields.

Structure Field	Contents
`res.fgrid`	Vector containing the frequency grid used in the filter design optimization
`res.des`	Desired response on `fgrid`
`res.wt`	Weights on `fgrid`
`res.h`	Actual frequency response on the frequency grid
`res.error`	Error at each point (desired response - actual response) on the frequency grid
`res.iextr`	Vector of indices into `fgrid` of external frequencies
`res.fextr`	Vector of external frequencies
`res.order`	Filter order
`res.edgecheck`	Transition-region anomaly check. One element per band edge. Element values have the following meanings: 1 = OK, 0 = probable transition-region anomaly, -1 = edge not checked. Computed when you specify the `'check'` input option in the function syntax.
`res.iterations`	Number of `s` iterations for the optimization
`res.evals`	Number of function evaluations for the optimization

`firgr` is also a "function function," allowing you to write a function that defines the desired frequency response.

`b = firgr(n,f,fresp,w)` returns a length N + 1 FIR filter which has the best approximation to the desired frequency response as returned by the user-defined function `fresp`. Use the following `firgr` syntax to call `fresp`:

```
[dh,dw] = fresp(n,f,gf,w)
```

where:

- `fresp` identifies the function that you use to define your desired filter frequency response.
- `n` is the filter order.
- `f` is the vector of frequency band edges which must appear monotonically between 0 and 1, where 1 is one-half of the sampling frequency. The frequency bands span `f(k)` to `f(k+1)` for `k` odd. The intervals `f(k+1)` to `f(k+2)` for `k` odd are “transition bands” or “don't care” regions during optimization.
- `gf` is a vector of grid points that have been chosen over each specified frequency band by `firgr`, and determines the frequencies at which `firgr` evaluates the response function.
- `w` is a vector of real, positive weights, one per band, for use during optimization. `w` is optional in the call to `firgr`. If you do not specify `w`, it is set to unity weighting before being passed to `fresp`.
- `dh` and `dw` are the desired frequency response and optimization weight vectors, evaluated at each frequency in grid `gf`.

`firgr` includes a predefined frequency response function named `'firpmfrf2'`. You can write your own based on the simpler `'firpmfrf'`. See the help for `private/firpmfrf` for more information.

`b = firgr(n,f,{`*`fresp`*`,p1,p2,...},w)` specifies optional arguments `p1`, `p2`,..., `pn` to be passed to the response function `fresp`.

`b = firgr(n,f,a,w)` is a synonym for `b = firgr(n,f,{'firpmfrf2',a},w)`, where `a` is a vector containing your specified response amplitudes at each band edge in `f`. By default, `firgr` designs symmetric (even) FIR filters. `'firpmfrf2'` is the predefined frequency response function. If you do not specify your own frequency response function (the `fresp` variable), `firgr` uses `'firpmfrf2'`.

`b = firgr(...,'h')` and `b = firgr(...,'d')` design antisymmetric (odd) filters. When you omit the `'h'` or `'d'` arguments from the `firgr` command syntax, each frequency response function `fresp` can tell `firgr` to design either an even or odd filter. Use the command syntax `sym = `*`fresp`*`('defaults',{n,f,[],w,p1,p2,...})`.

`firgr` expects `fresp` to return `sym = 'even'` or `sym = 'odd'`. If `fresp` does not support this call, `firgr` assumes even symmetry.

For more information about the input arguments to `firgr`, refer to `firpm`.

Examples

Design a Filter Using firgr

Design an FIR filter with two single-band notches at 0.25 and 0.55.

```
b1 = firgr(42,[0 0.2 0.25 0.3 0.5 0.55 0.6 1],[1 1 0 1 1 0 1 1],...
{'n' 'n' 's' 'n' 'n' 's' 'n' 'n'});
```

Design a highpass filter whose gain at 0.06 is forced to be zero. The gain at 0.055 is indeterminate since it should abut the band.

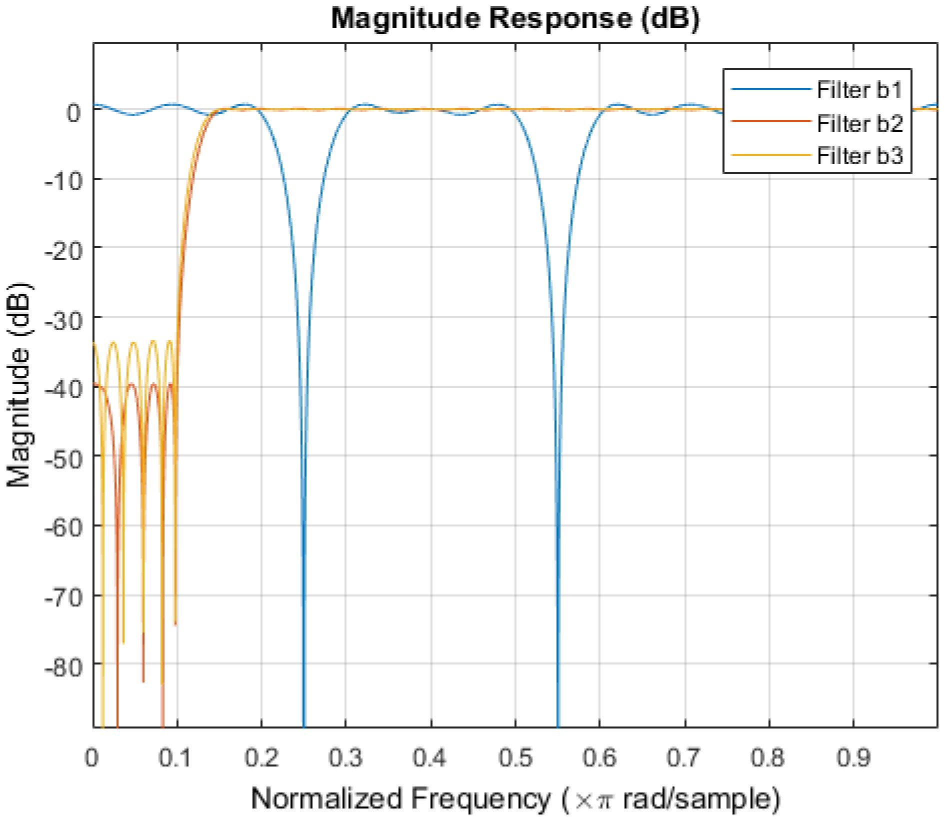
```
b2 = firgr(82,[0 0.055 0.06 0.1 0.15 1],[0 0 0 0 1 1],...
{'n' 'i' 'f' 'n' 'n' 'n'});
```

Design a second highpass filter with forced values and independent approximation errors.

```
b3 = firgr(82,[0 0.055 0.06 0.1 0.15 1], [0 0 0 0 1 1], ...
{'n' 'i' 'f' 'n' 'n' 'n'}, [10 1 1] ,{'e1' 'e2' 'e3'});
```

Use the filter visualization tool to view the results of the filters.

```
fvtool(b1,1,b2,1,b3,1)
legend('Filter b1','Filter b2','Filter b3');
```



References

Shpak, D.J. and A. Antoniou, "A generalized Remez method for the design of FIR digital filters," *IEEE Trans. Circuits and Systems*, pp. 161-174, Feb. 1990.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All inputs must be constant. Expressions or variables are allowed if their values do not change.
- Does not support syntaxes that have cell array input.

See Also

`butter` | `cheby1` | `cheby2` | `ellip` | `freqz` | `filter` | `firls` | `fircls` | `firpm`

Introduced in R2011a

firhalfband

Halfband FIR filter design

Syntax

```
b = firhalfband(n,fp)
b = firhalfband(n,win)
b = firhalfband(n,dev,'dev')
b = firhalfband('minorder',fp,dev)
b = firhalfband('minorder',fp,dev,'kaiser')
b = firhalfband(...,'high')
b = firhalfband(...,'minphase')
```

Description

`b = firhalfband(n,fp)` designs a lowpass halfband FIR filter of order `n` with an equiripple characteristic. `n` must be an even integer. `fp` determines the passband edge frequency, and it must satisfy 0 < `fp` < 1/2, where 1/2 corresponds to *π/2* rad/sample.

`b = firhalfband(n,win)` designs a lowpass Nth-order filter using the truncated, windowed-impulse response method instead of the equiripple method. `win` is an `n +1` length vector. The ideal impulse response is truncated to length `n + 1`, and then multiplied point-by-point with the window specified in `win`.

`b = firhalfband(n,dev,'dev')` designs an Nth-order lowpass halfband filter with an equiripple characteristic. Input argument `dev` sets the value for the maximum passband and stopband ripple allowed.

`b = firhalfband('minorder',fp,dev)` designs a lowpass minimum-order filter, with passband edge `fp`. The peak ripple is constrained by the scalar `dev`. This design uses the equiripple method.

`b = firhalfband('minorder',fp,dev,'kaiser')` designs a lowpass minimum-order filter, with passband edge `fp`. The peak ripple is constrained by the scalar `dev`. This design uses the Kaiser window method.

`b = firhalfband(...,'high')` returns a highpass halfband FIR filter.

`b = firhalfband(...,'minphase')` designs a minimum-phase FIR filter such that the filter is a spectral factor of a halfband filter (recall that `h = conv(b,fliplr(b))` is a halfband filter). This can be useful for designing perfect reconstruction, two-channel FIR filter banks. The **minphase** option for `firhalfband` is not available for the window-based halfband filter designs — `b = firhalfband(n,win)` and `b = firhalfband('minorder',fp,dev,'kaiser')`.

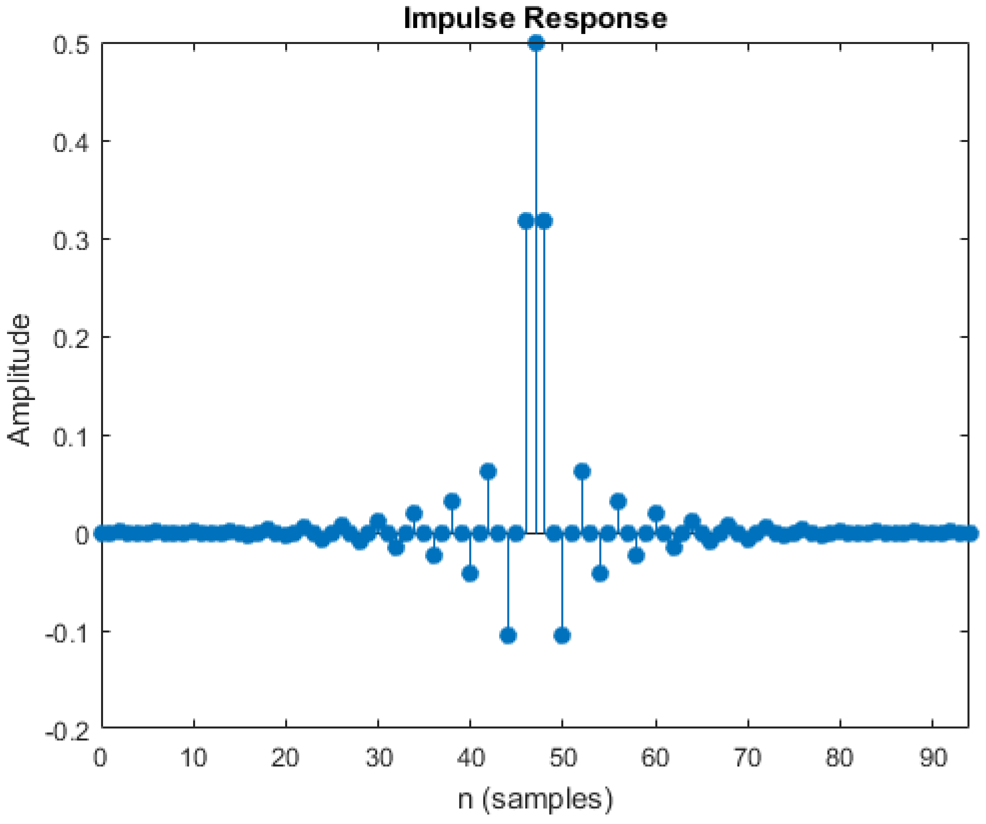
In the minimum phase cases, the filter order must be odd.

Examples

Design a Halfband Filter using halfband

This example designs a minimum order halfband filter with a specified maximum ripple.

```
b = firhalfband('minorder',.45,0.0001);
impz(b)
```



You can see that the impulse response is zero for every alternate sample.

References

[1] Saramaki, T, "Finite Impulse Response Filter Design," *Handbook for Digital Signal Processing*. S.K. Mitra and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

See Also

Functions

`fir1` | `firgr` | `firls` | `firnyquist` | `firpm`

Introduced in R2011a

firlp2lp

Convert FIR Type I lowpass to FIR Type 1 lowpass with inverse bandwidth

Syntax

```
g = firlp2lp(b)
```

Description

`g = firlp2lp(b)` transforms the Type I lowpass FIR filter `b` with zero-phase response Hr(w) to a Type I lowpass FIR filter `g` with zero-phase response [1 - Hr(π-w)].

When `b` is a narrowband filter, `g` will be a wideband filter and vice versa. The passband and stopband ripples of `g` will be equal to the stopband and passband ripples of `b`.

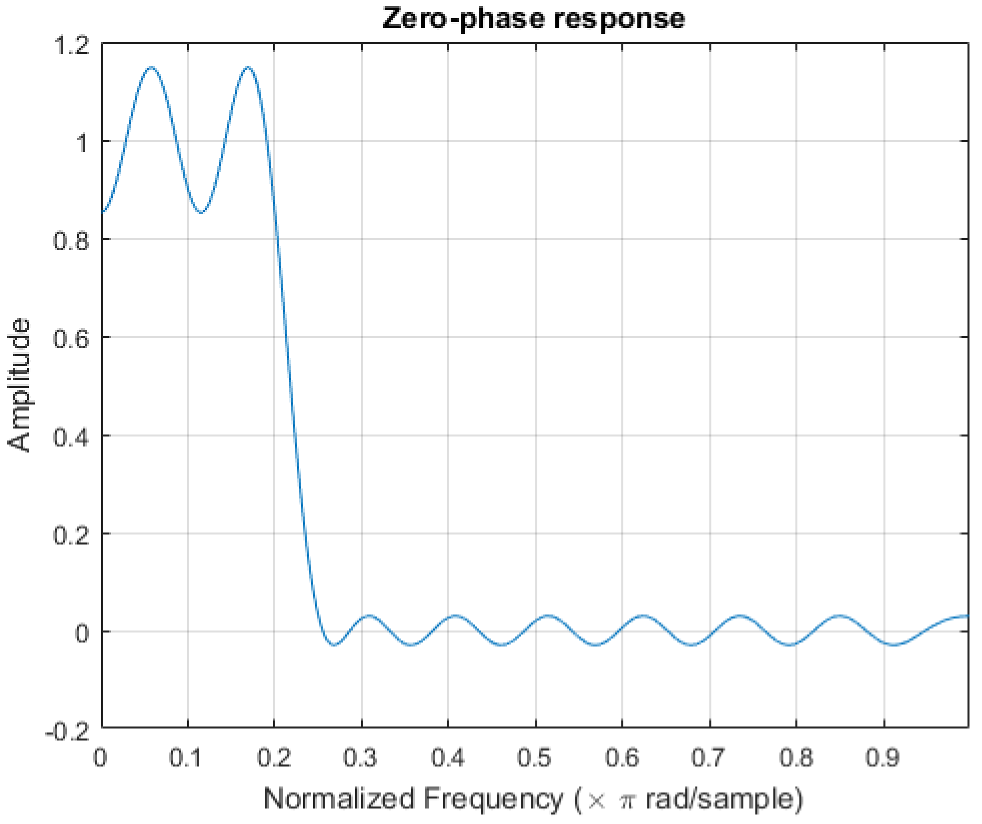
Examples

Convert Narrowband Lowpass Filter to Wideband Lowpass Filter

Create a narrowband lowpass filter to use as prototype. Display its zero-phase response.

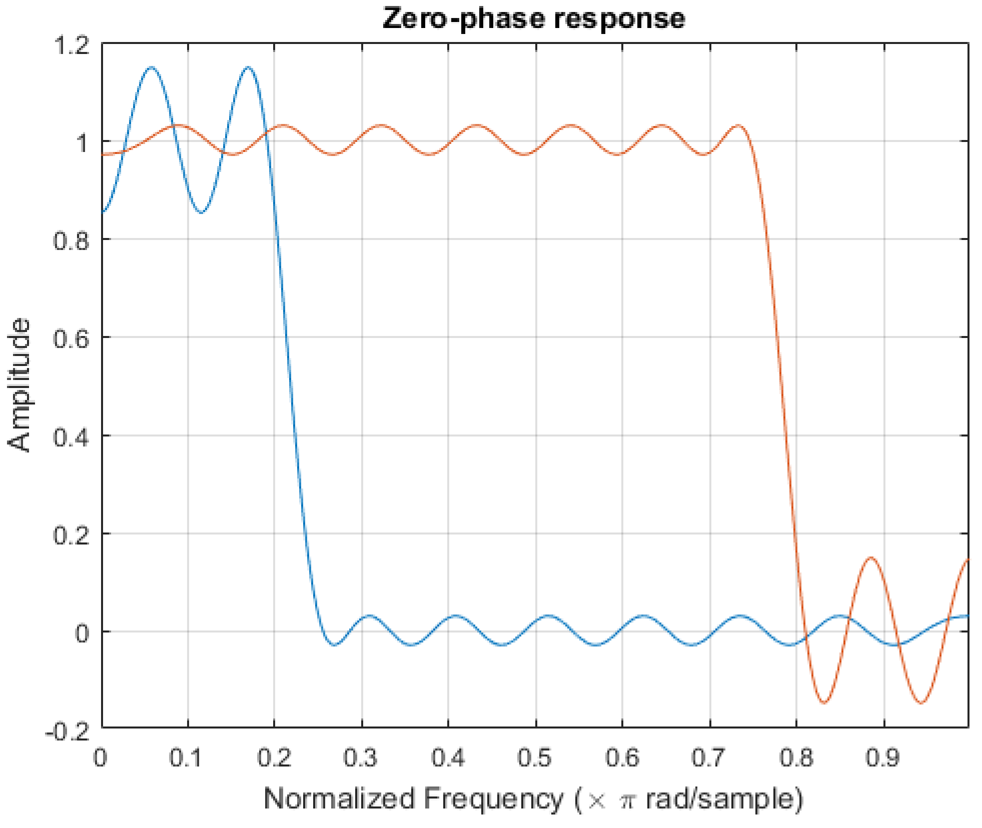
```
b = firgr(36,[0 0.2 0.25 1],[1 1 0 0],[1 5]);
```

```
zerophase(b)
```



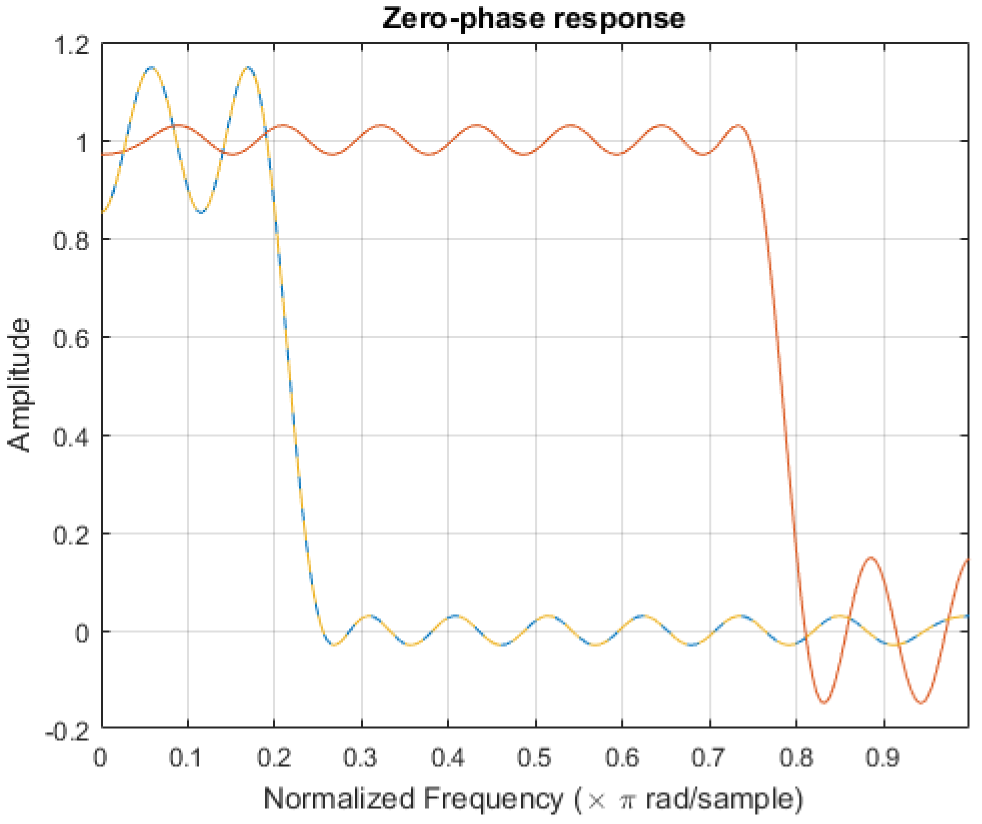
Convert the prototype filter to a wideband lowpass filter. Add to the plot the zero-phase response of the new filter.

```
h = firlp2lp(b);

hold on
zerophase(h)
```



Convert the previous filter back to a narrowband lowpass filter. Add to the plot the zero-phase response of the new filter.

```
g = firlp2lp(h);

[gr,w] = zerophase(g);
plot(w/pi,gr,'--')
hold off
```



References

Saramaki, T, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*. S.K. Mitra and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

See Also

`firlp2hp` | `zerophase`

Introduced in R2011a

firlp2hp

Convert FIR lowpass filter to Type I FIR highpass filter

Syntax

```
g = firlp2hp(b)
g = firlp2hp(b,'narrow')
g = firlp2hp(b,'wide')
```

Description

`g = firlp2hp(b)` transforms the lowpass FIR filter `b` into a Type I highpass FIR filter `g` with zero-phase response Hr(π-w). Filter `b` can be any FIR filter, including a nonlinear-phase filter.

The passband and stopband ripples of `g` will be equal to the passband and stopband ripples of `b`.

`g = firlp2hp(b,'narrow')` transforms the lowpass FIR filter `b` into a Type I narrow band highpass FIR filter `g` with zero-phase response Hr(π-w). `b` can be any FIR filter, including a nonlinear-phase filter.

`g = firlp2hp(b,'wide')` transforms the Type I lowpass FIR filter `b` with zero-phase response Hr(w) into a Type I wide band highpass FIR filter `g` with zero-phase response 1 - Hr(w). Note the restriction that b must be a Type I linear-phase filter.

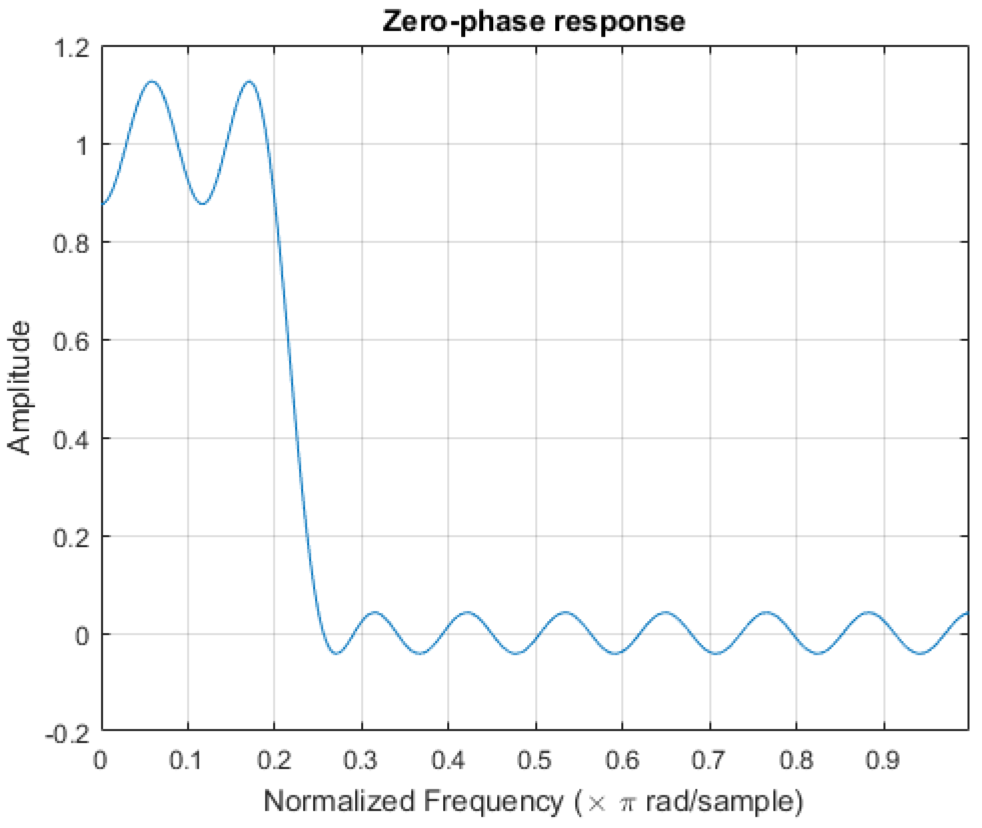
For this case, the passband and stopband ripples of `g` will be equal to the stopband and passband ripples of `b`.

Examples

Convert Narrowband Lowpass Filter to Highpass Filter

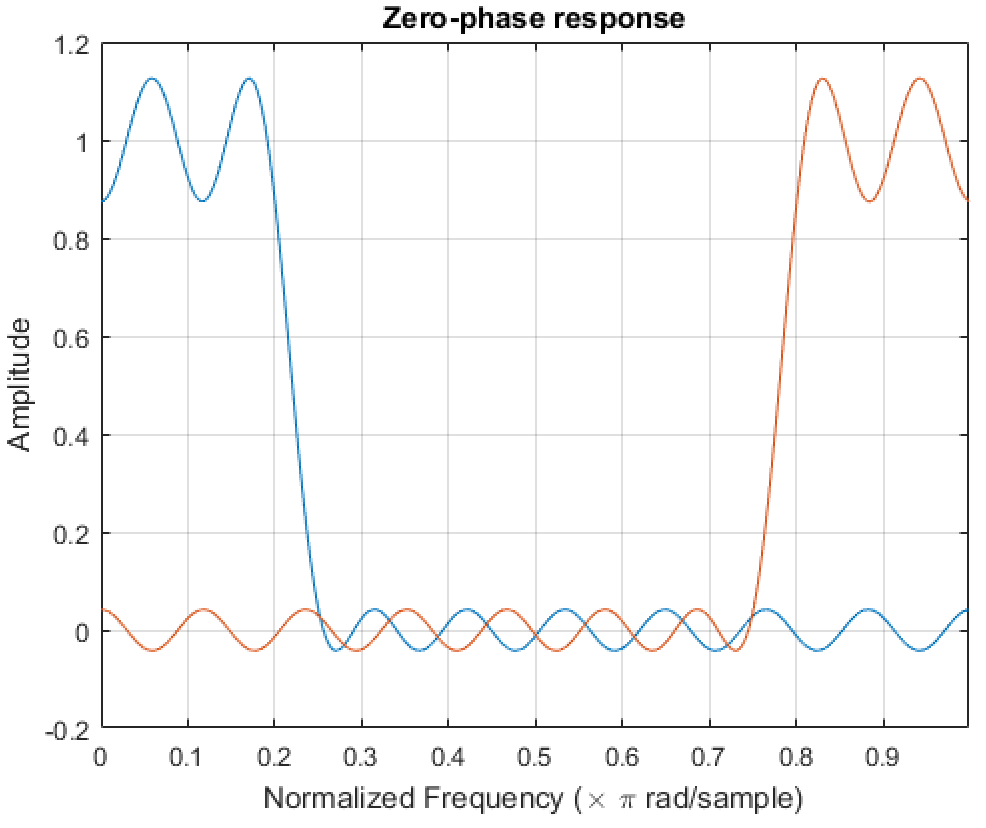
Create a narrowband lowpass filter to use as prototype. Display its zero-phase response.

```
b = firgr(36,[0 0.2 0.25 1],[1 1 0 0],[1 3]);

zerophase(b)
```



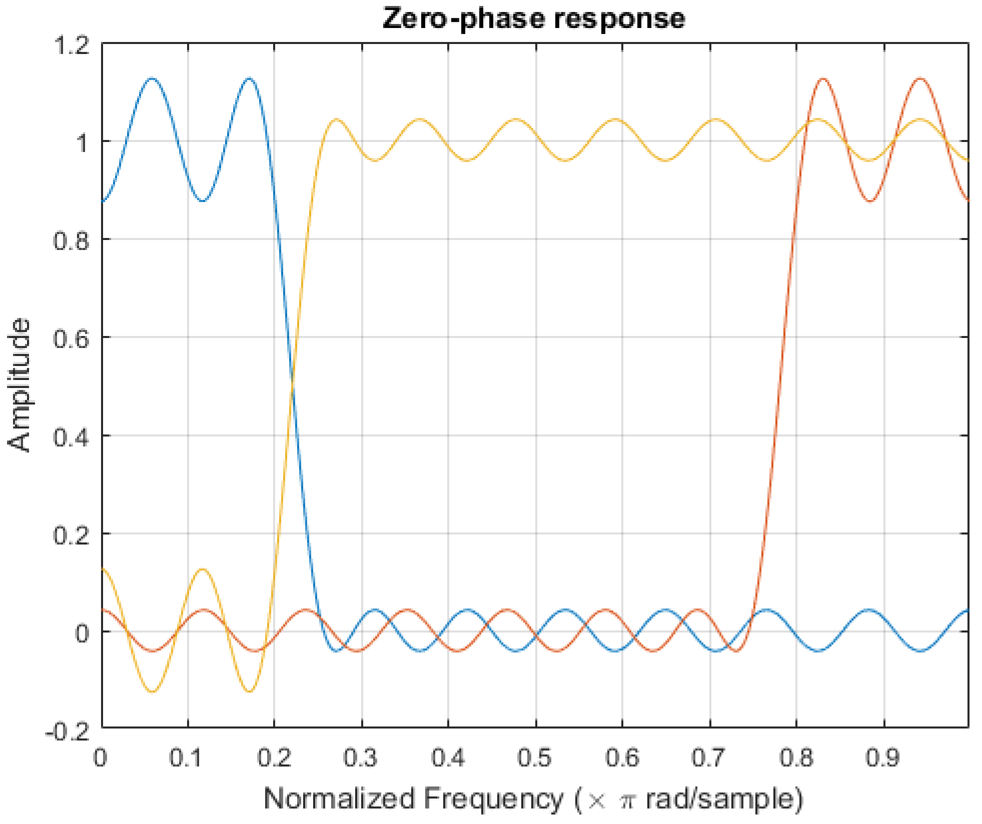
Convert the prototype filter to a narrowband highpass filter. Add to the plot the zero-phase response of the new filter.

```
h = firlp2hp(b);

hold on
zerophase(h)
```



Convert the prototype filter to a wideband highpass filter. Add to the plot the zero-phase response of the new filter.

```
g = firlp2hp(b,'wide');

zerophase(g)
hold off
```



References

Saramaki, T, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*Mitra, S.K. and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

See Also

`firlp2lp` | `zerophase`

Introduced in R2011a

firlpnorm

Least P-norm optimal FIR filter

Syntax

```
b = firlpnorm(n,f,edges,a)
b = firlpnorm(n,f,edges,a,w)
b = firlpnorm(n,f,edges,a,w,p)
b = firlpnorm(n,f,edges,a,w,p,dens)
b = firlpnorm(n,f,edges,a,w,p,dens,initnum)
b = firlpnorm(...,'minphase')
[b,err] = firlpnorm(...)
```

Description

`b = firlpnorm(n,f,edges,a)` returns a filter of numerator order `n` which represents the best approximation to the frequency response described by `f` and `a` in the least-Pth norm sense. P is set to 128 by default, which essentially equivalent to the infinity norm. Vector `edges` specifies the band-edge frequencies for multiband designs. `firlpnorm` uses an unconstrained quasi-Newton algorithm to design the specified filter.

`f` and `a` must have the same number of elements, which can exceed the number of elements in `edges`. This lets you specify filters with any gain contour within each band. However, the frequencies in `edges` must also be in vector `f`. Always use `freqz` to check the resulting filter.

Note `firlpnorm` uses a nonlinear optimization routine that may not converge in some filter design cases. Furthermore the algorithm is not well-suited for certain large-order (order > 100) filter designs.

`b = firlpnorm(n,f,edges,a,w)` uses the weights in `w` to weight the error. `w` has one entry per frequency point (the same length as `f` and `a`) which tells `firlpnorm` how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points. For example,

```
b = firlpnorm(20,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])
```

designs a lowpass filter with a peak of 1.6 within the passband, and with emphasis placed on minimizing the error in the stopband.

`b = firlpnorm(n,f,edges,a,w,p)` where `p` is a two-element vector [`pmin pmax`] lets you specify the minimum and maximum values of `p` used in the least-pth algorithm. Default is `[2 128]` which essentially yields the L-infinity, or Chebyshev, norm. `pmin` and `pmax` should be even numbers. The design algorithm starts optimizing the filter with `pmin` and moves toward an optimal filter in the `pmax` sense. When `p` is set to '**`inspect`**', `firlpnorm` does not optimize the resulting filter. You might use this feature to inspect the initial zero placement.

`b = firlpnorm(n,f,edges,a,w,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is [`dens*(n+1)`]. The default is 20. You can specify `dens` as a single-element cell array. The grid is equally spaced.

`b = firlpnorm(n,f,edges,a,w,p,dens,initnum)` lets you determine the initial estimate of the filter numerator coefficients in vector `initnum`. This can prove helpful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox software can be used for generating `initnum`.

`b = firlpnorm(...,'minphase')` where '`minphase`' is the last argument in the argument list generates a minimum-phase FIR filter. By default, `firlpnorm` design mixed-phase filters. Specifying input option '`minphase`' causes `firlpnorm` to use a different optimization method to design the minimum-phase filter. As a result of the different optimization used, the minimum-phase filter can yield slightly different results.

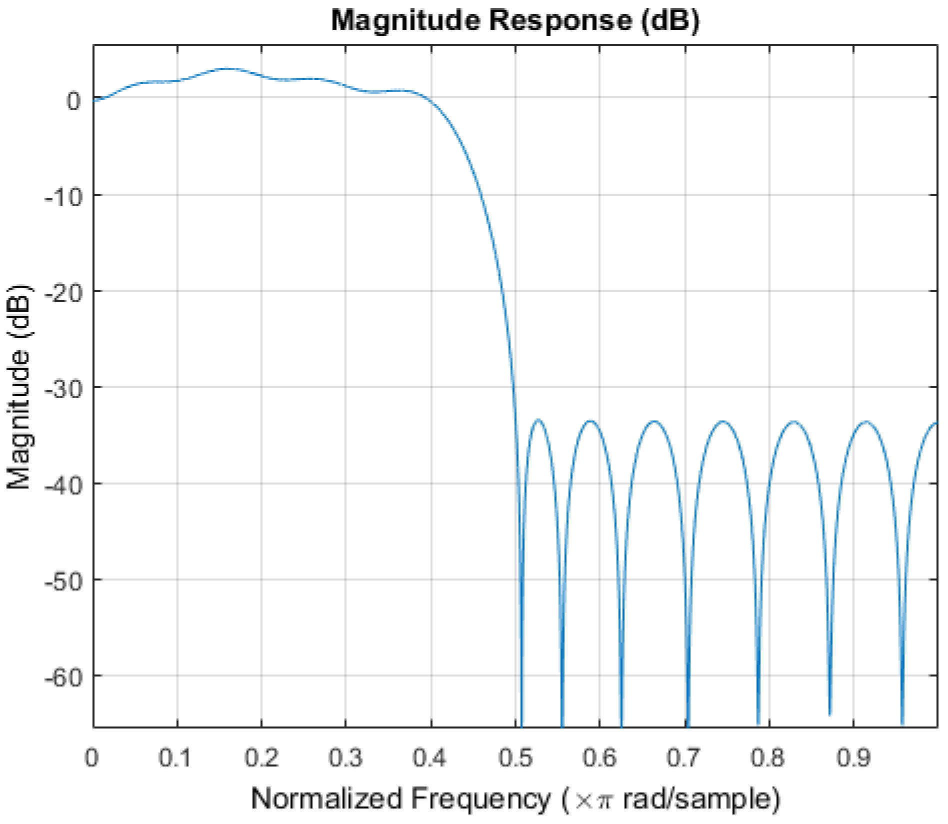
`[b,err] = firlpnorm(...)` returns the least-pth approximation error `err`.

Examples

Design a Lowpass and Highpass Filter Using firlpnorm

Lowpass filter with a peak of 1.4 in the passband.

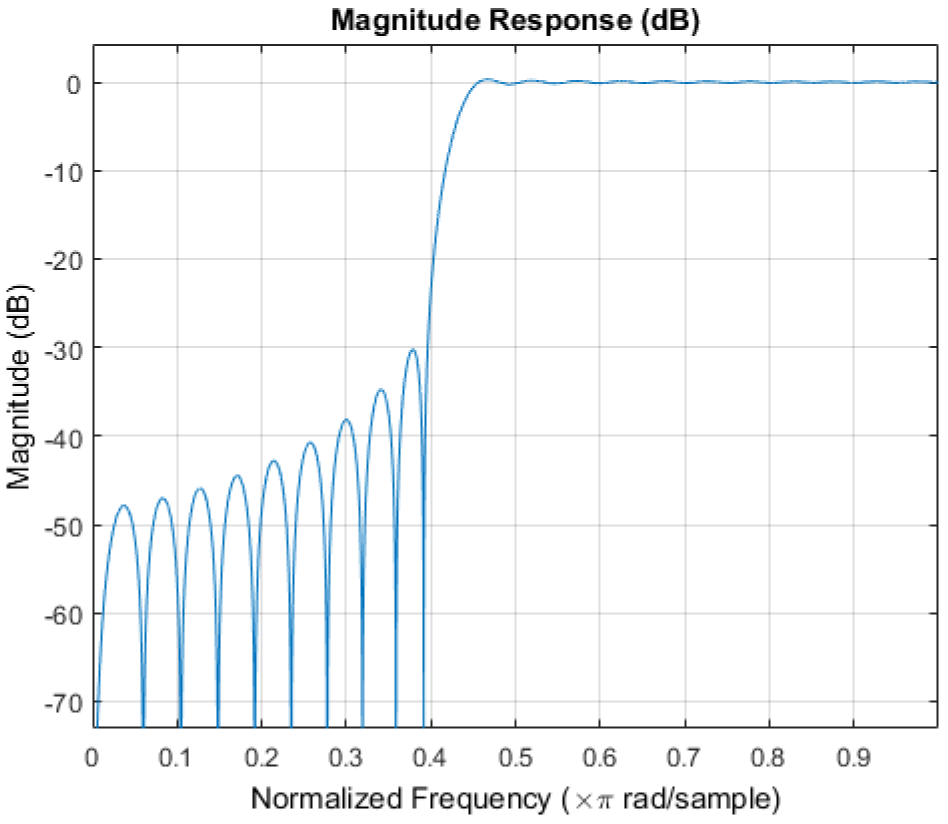
```
b = firlpnorm(22,[0 .15 .4 .5 1],[0 .4 .5 1],[1 1.4 1 0 0],...
[1 1 1 2 2]);
fvtool(b)
```



The resulting filter is lowpass, with the desired 1.4 peak in the passband (notice the 1.4 specified in vector a).

Highpass minimum-phase filter optimized for the 4-norm.

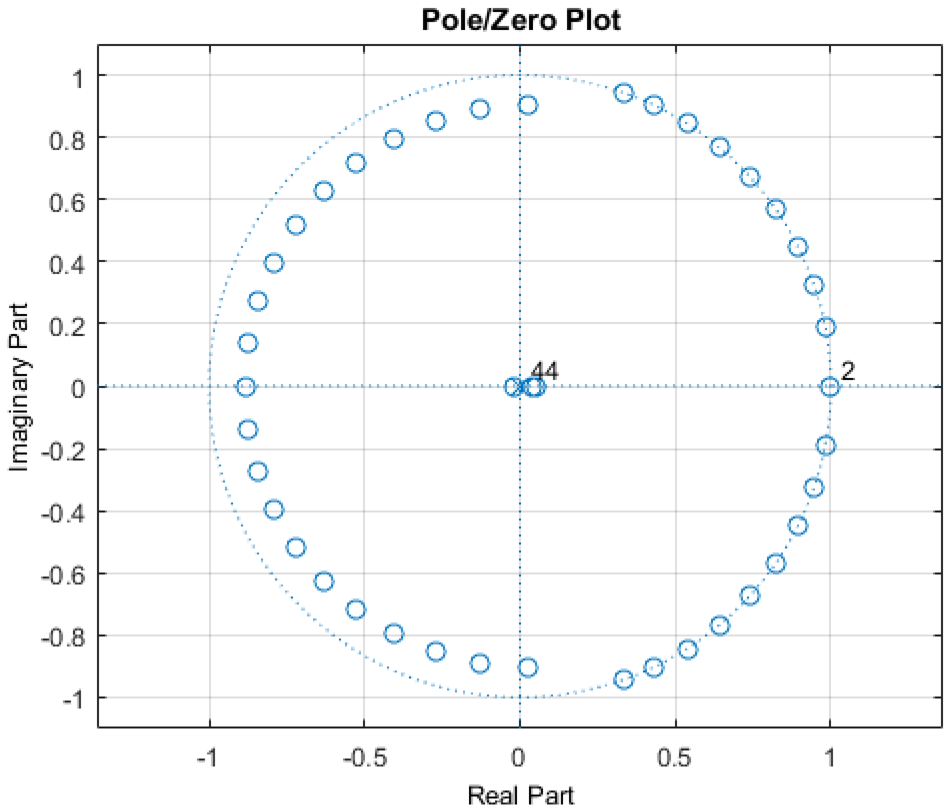
```
b = firlpnorm(44,[0 .4 .45 1],[0 .4 .45 1],[0 0 1 1],[5 1 1 1],...
[2 4],'minphase');
fvtool(b)
```



This is a minimum-phase, highpass filter.

The zero-pole plot shows the minimum phase nature more clearly.

```
fvtool(b,'polezero')
```



References

Saramaki, T, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*Mitra, S.K. and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All inputs must be constant. Expressions or variables are allowed if their values do not change.
- Does not support syntaxes that have cell array input.

See Also

`firgr` | `iirgrpdelay` | `filter` | `fvtool` | `freqz` | `zplane` | `iirlpnorm` | `iirlpnormc`

Introduced in R2011a

firls

Least-square linear-phase FIR filter design

Syntax

```
b = firls(n,f,a)
b = firls(n,f,a,w)
b = firls(n,f,a,ftype)
b = firls(n,f,a,w,ftype)
```

Description

`b = firls(n,f,a)` returns row vector `b` containing the `n+1` coefficients of the order `n` FIR filter. This filter has frequency-amplitude characteristics approximately matching those given by vectors, `f` and `a`.

`b = firls(n,f,a,w)` uses the weights in vector `w`, to weigh the error.

`b = firls(n,f,a,ftype)` specifies a filter type where `ftype` is:

- `'hilbert'`
- `'differentiator'`

`b = firls(n,f,a,w,ftype)` uses the weights in vector `w` to weigh the error. It also specifies a filter type where `ftype` is:

- `'hilbert'`
- `'differentiator'`

Examples

Design a Lowpass Filter with Transition Band

The following illustrates how to design a lowpass filter of order 225 with transition band.

Create the frequency and amplitude vectors, `f` and `a`.

```
f = [0 0.25 0.3 1]
a = [1 1 0 0]
```

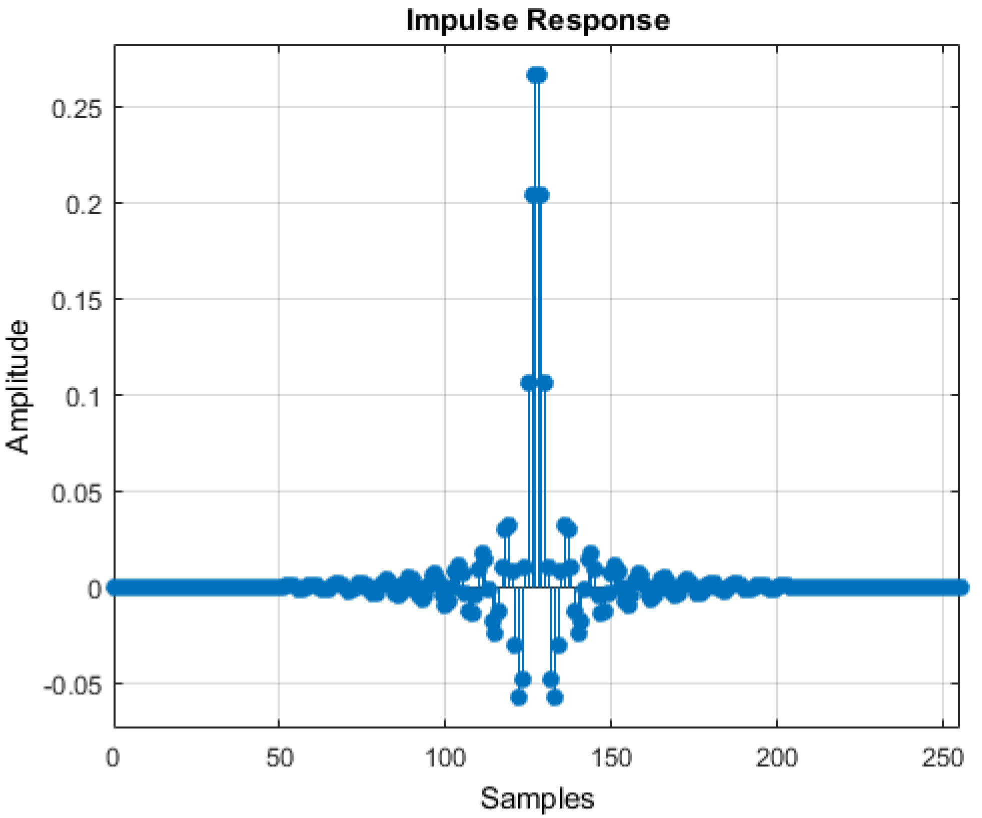
```
f =

         0    0.2500    0.3000    1.0000


a =

     1     1     0     0
```

Use `firls` to obtain the `n+1` coefficients of the order `n` lowpass FIR filter.

```
b = firls(255,f,a);
```

Show the impulse response of the filter

```
fvtool(b,'impulse')
```



Design an Antisymmetric Filter with Piecewise Linear Passbands

The following shows how to design a 24th-order anti-symmetric filter with piecewise linear passbands, and plot the desired and actual amplitude responses.

Create the frequency and amplitude vectors, `f` and `a`.

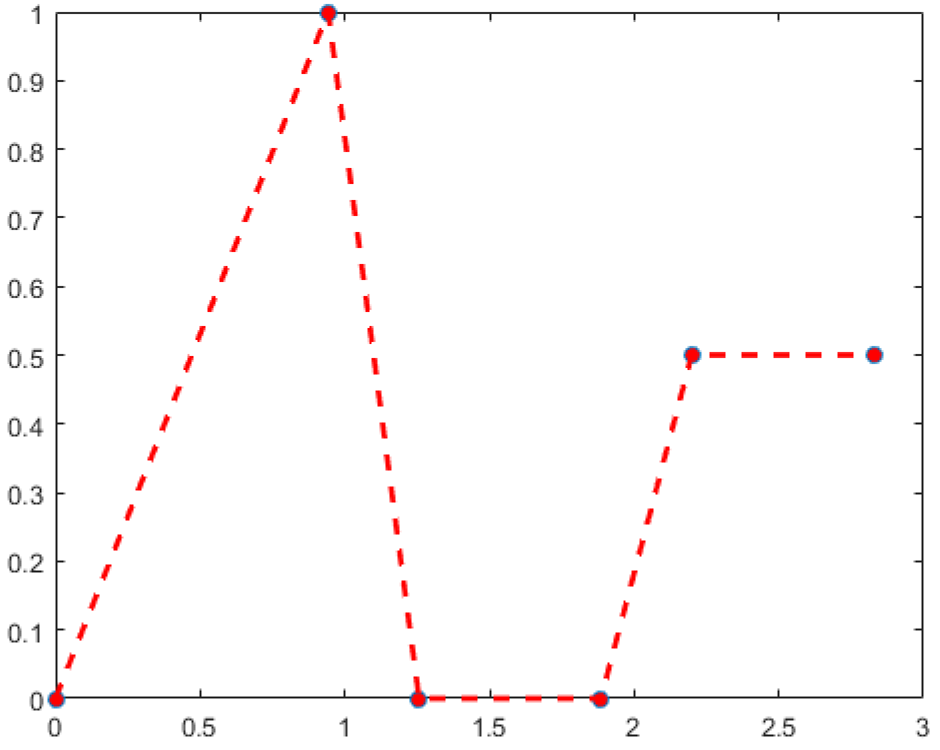
```
f = [0 0.3 0.4 0.6 0.7 0.9];
a = [0 1 0 0 0.5 0.5];
```

Use `firls` to obtain the 25 coefficients of the filter.

```
b = firls(24,f,a,'hilbert');
```

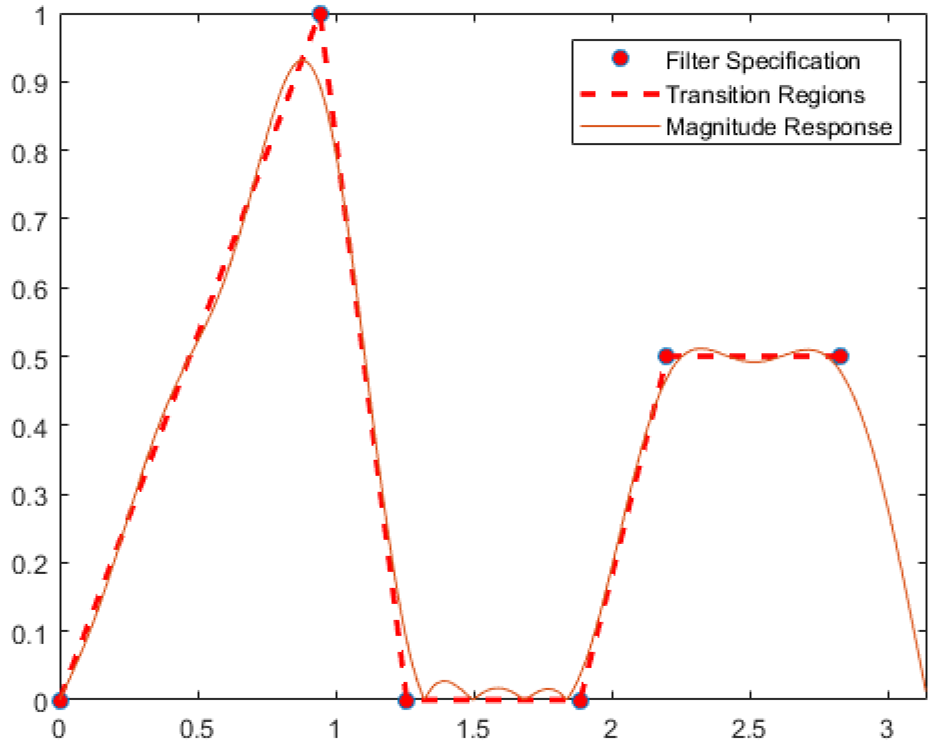
Plot the ideal amplitude response along with the transition regions.

```
plot(f.*pi,a,'o','markerfacecolor',[1 0 0]);
hold on;
plot(f.*pi,a,'r--','linewidth',2);
```



Use `freqz` to obtain the frequency response of the designed filter and plot the magnitude response of the filter.

```
[H,F] = freqz(b,1);
plot(F,abs(H));
set(gca,'xlim',[0 pi])
legend('Filter Specification','Transition Regions','Magnitude Response')
```



Input Arguments

n — Filter order

(default) | integer scalar

Order of the filter, specified as an integer scalar. For odd orders, the frequency response at the Nyquist frequency is necessarily 0. For this reason, `firls` always uses an even filter order for configurations with a passband at the Nyquist frequency. If you specify an odd-valued `n`, `firls` increments it by 1.

Example: 8

Data Types: `int8` | `int16` | `int32` | `int64`

`f` — Pairs of frequency points

vector of numeric values

Pairs of frequency points, specified as a vector of values ranging between 0 and 1, where 1 corresponds to the Nyquist frequency. The frequencies must be in increasing order, and duplicate frequency points are allowed. You can use duplicate frequency points to design filters exactly like those returned by the `fir1` and `fir2` functions with a rectangular (`rectwin`) window.

`f` and `a` are the same length. This length must be an even number.

Example: `[0 0.3 0.4 1]`

Data Types: `double` | `single`

`a` — Amplitude values

vector of numeric values

Amplitude values of the function at each frequency point, specified as a vector of the same length as `f`. This length must be an even number.

The desired amplitude at frequencies between pairs of points ($f(k)$, $f(k+1)$) for k odd, is the line segment connecting the points ($f(k)$, $a(k)$) and ($f(k+1)$, $a(k+1)$).

The desired amplitude at frequencies between pairs of points ($f(k)$, $f(k+1)$) for k even is unspecified. These are transition or “don't care” regions.

Example: `[1 1 0 0]`

Data Types: `double` | `single`

`w` — Weights

vector of numeric values

Weights to weigh the fit for each frequency band, specified as a vector of length half the length of `f` and `a`, so there is exactly one weight per band. `w` indicates how much emphasis to put on minimizing the integral squared error in each band, relative to the other bands.

Example: `[0.5 1]`

Data Types: `double` | `single`

ftype — Filter type
'hilbert' and 'differentiator'

Filter type, specified as either 'hilbert' or 'differentiator'.

Example: 'hilbert'

Data Types: char

Output Arguments

b — Filter coefficients
vector of numeric values

Filter coefficients, returned as a numeric vector of n+1 values, where n is the filter order.

b = firls(n,f,a) designs a linear-phase filter of type I (n odd) and type II (n) . The output coefficients, or "taps," in b obey the relation:
b(k) = b(n+2-k), k = 1, ... , n + 1

b = firls(n,f,a,'hilbert') designs a linear-phase filter with odd symmetry (type III and type IV). The output coefficients, or "taps," in b obey the relation:
b(k) = –b(n+2-k), k = 1, ... , n + 1

b = firls(n,f,a,'differentiator') designs type III and type IV filters, using a special weighting technique. For nonzero amplitude bands, the integrated squared error has a weight of $(1/f)^2$. This weighting causes the error at low frequencies to be much smaller than at high frequencies. For FIR differentiators, which have an amplitude characteristic proportional to frequency, the filters minimize the relative integrated squared error. This value is the integral of the square of the ratio of the error to the desired amplitude.

Definitions

Diagnostics

Error and warning messages

One of the following diagnostic messages is displayed when an incorrect argument is used:

```
F must be even length.
F and A must be equal lengths.
Requires symmetry to be 'hilbert' or 'differentiator'.
Requires one weight per band.
Frequencies in F must be nondecreasing.
Frequencies in F must be in range [0,1].
```

A more serious warning message is

```
Warning: Matrix is close to singular or badly scaled.
```

This tends to happen when the product of the filter length and transition width grows large. In this case, the filter coefficients b might not represent the desired filter. You can check the filter by looking at its frequency response.

Algorithms

`firls` designs a linear-phase FIR filter. This filter minimizes the weighted, integrated squared error between an ideal piecewise linear function and the magnitude response of the filter over a set of desired frequency bands.

Reference [1] describes the theoretical approach behind `firls`. The function solves a system of linear equations involving an inner product matrix of size roughly `n/2` using the MATLAB \ operator.

This function designs type I, II, III, and IV linear-phase filters. Type I and II are the defaults for n even and odd respectively. The `'hilbert'` and `'differentiator'` flags produce type III (n even) and IV (n odd) filters. The various filter types have different symmetries and constraints on their frequency responses (see [2] for details).

Linear Phase Filter Type	Filter Order	Symmetry of Coefficients	Response H(f), f = 0	Response H(f), f = 1 (Nyquist)
Type I	Even	b(k) = b(n+2-k), k=1,..., n+1	No restriction	No restriction
Type II	Even	b(k) = b(n+2-k), k=1,..., n+1	No restriction	$H(1) = 0$
Type III	Odd	b(k) = –b(n+2-k), k=1,..., n+1	$H(0) = 0$	$H(1) = 0$
Type IV	Odd	b(k) = –b(n+2-k), k=1,..., n+1	$H(0) = 0$	No restriction

References

[1] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, 1987, pp. 54-83.

[2] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 256-266.

See Also

See Also

`fir1` | `fir2` | `firpm` | `rcosdesign`

Introduced in R2011a

firminphase

Minimum-phase FIR spectral factor

Syntax

```
h = firminphase(b)
h = firminphase(b,nz)
```

Description

`h = firminphase(b)` computes the minimum-phase FIR spectral factor `h` of a linear-phase FIR filter `b`. Filter `b` must be real, have even order, and have nonnegative zero-phase response.

`h = firminphase(b,nz)` specifies the number of zeros, `nz`, of `b` that lie on the unit circle. You must specify `nz` as an even number to compute the minimum-phase spectral factor because every root on the unit circle must have even multiplicity. Including `nz` can help `firminphase` calculate the required FIR spectral factor. Zeros with multiplicity greater than two on the unit circle cause problems in the spectral factor determination.

Note You can find the maximum-phase spectral factor, `g`, by reversing `h`, such that *g = fliplr(h)*, and *b = conv(h, g)*.

Examples

Design a Constrained Least Squares Filter using firminphase

This example designs a constrained least squares filter with a nonnegative zero-phase response, and then uses `firminphase` to compute the minimum-phase spectral factor.

```
f   = [0 0.4 0.8 1];
a   = [0 1 0];
up  = [0.02 1.02  0.01];
lo  = [0 0.98 0]; % The zeros insure nonnegative zero-phase resp.
n   = 32;
```

```
b   = fircls(n,f,a,up,lo);
h   = firminphase(b)
```

```
h =

  Columns 1 through 7

    0.2397   -0.1556   -0.2834    0.3866    0.0415   -0.2529    0.0584

  Columns 8 through 14

   -0.0028    0.0868    0.0079   -0.0978    0.0309    0.0095    0.0669

  Columns 15 through 17

    0.0171   -0.0111   -0.0019
```

References

Saramaki, T, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*Mitra, S.K. and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

`firgr` | `fircls` | `zerophase`

Introduced in R2011a

firnyquist

Lowpass Nyquist (Lth-band) FIR filter

Syntax

```
b = firnyquist(n,l,r)
b = firnyquist('minorder',l,r,dev)
b = firnyquist(n,l,r,decay)
b = firnyquist(n,l,r,'nonnegative')
b = firnyquist(n,l,r,'minphase')
```

Description

`b = firnyquist(n,l,r)` designs an Nth order, Lth band, Nyquist FIR filter with a rolloff factor `r` and an equiripple characteristic.

The rolloff factor `r` is related to the normalized transition width `tw` by $tw = 2\pi(r/l)$ (rad/sample). The order, `n`, must be even. `l` must be an integer greater than one. If `l` is not specified, it defaults to 4. `r` must satisfy `0< r < 1`. If `r` is not specified, it defaults to `0.5`.

`b = firnyquist('`**`minorder`**`',l,r,dev)` designs a minimum-order, Lth band Nyquist FIR filter with a rolloff factor `r` using the Kaiser window. The peak ripple is constrained by the scalar `dev`.

`b = firnyquist(n,l,r,decay)` designs an Nth order (`n`), Lth band (`l`), Nyquist FIR filter where the scalar `decay`, specifies the rate of decay in the stopband. `decay` must be nonnegative. If you omit or leave it empty, `decay` defaults to `0` which yields an equiripple stopband. A nonequiripple stopband (`decay` $\neq 0$) may be desirable for decimation purposes.

`b = firnyquist(n,l,r,'`**`nonnegative`**`')` returns an FIR filter with nonnegative zero-phase response. This filter can be spectrally factored into minimum-phase and maximum-phase “square-root” filters. This allows you to use the spectral factors in applications such as matched-filtering.

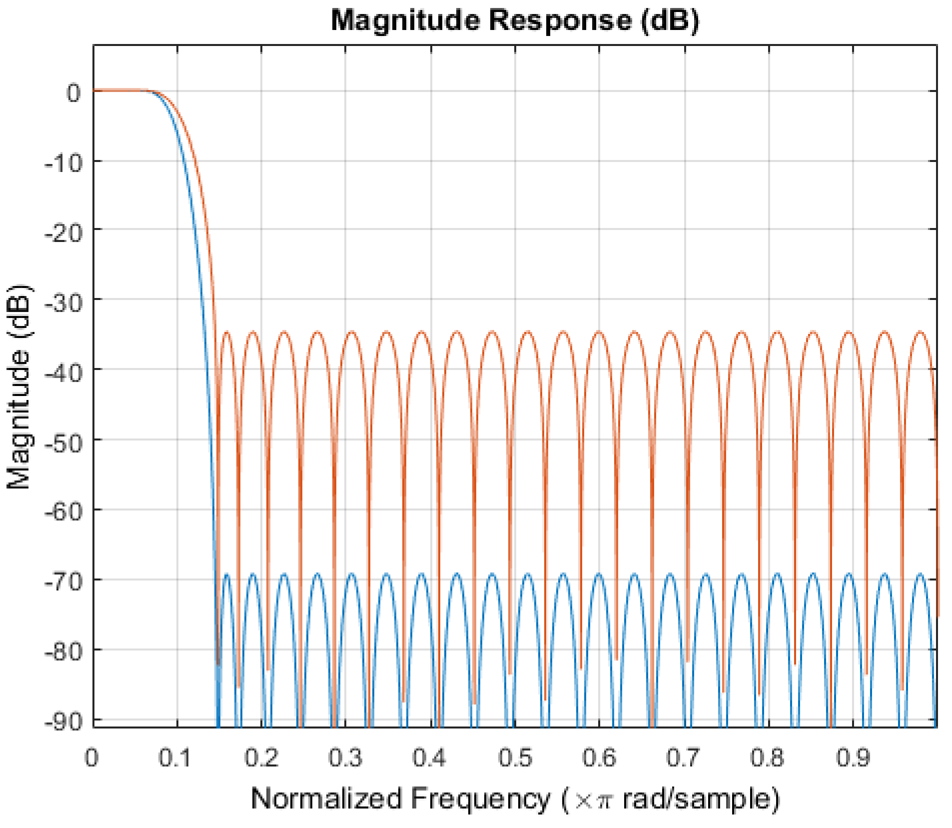
`b = firnyquist(n,l,r,'minphase')` returns the minimum-phase spectral factor `bmin` of order `n`. `bmin` meets the condition `b=conv(bmin,bmax)` so that `b` is an Lth band FIR Nyquist filter of order 2n with filter rolloff factor `r`. Obtain `bmax`, the maximum phase spectral factor by reversing the coefficients of `bmin`. For example, `bmax = bmin(end:-1:1)`.

Examples

Design a Nyquist Filter Using firnyquist

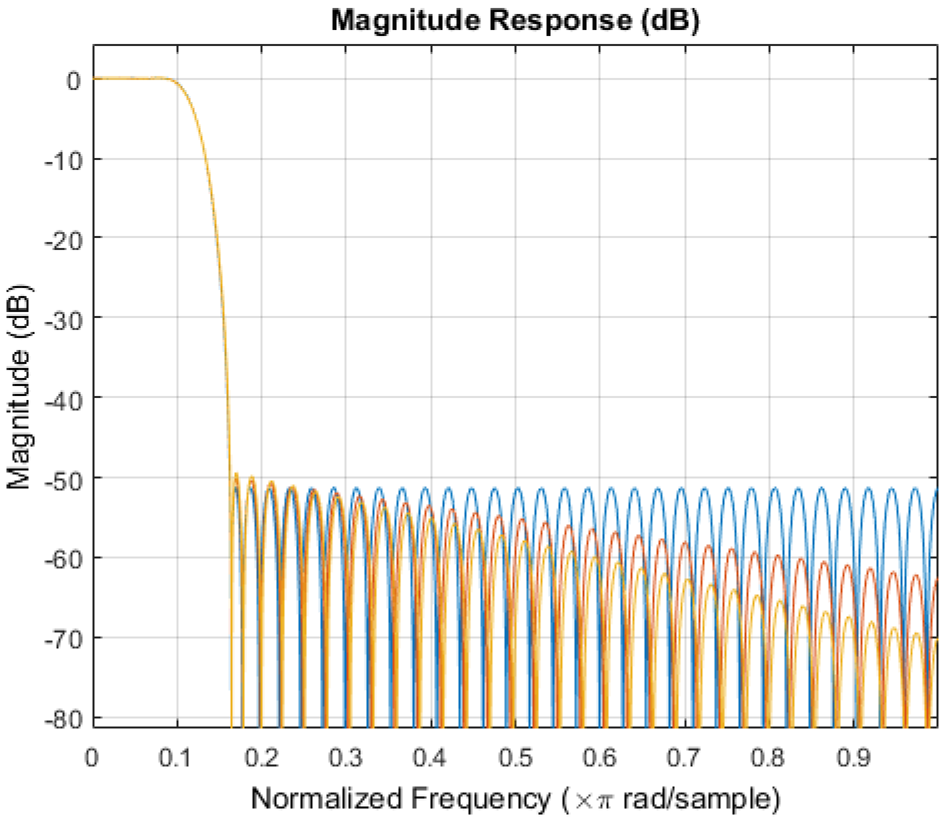
This example designs a minimum phase factor of a Nyquist filter.

```
bmin = firnyquist(47,10,.45,'minphase');
b = firnyquist(2*47,10,.45,'nonnegative');
[h,w,s] = freqz(b); hmin = freqz(bmin);
fvtool(b,1,bmin,1);
```



This example compares filters with different decay rates.

```
b1 = firnyquist(72,8,.3,0); % Equiripple
b2 = firnyquist(72,8,.3,15);
b3 = firnyquist(72,8,.3,25);
fvtool(b1,1,b2,1,b3,1);
```



References

T. Saramaki, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*, Mitra, S.K. and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also
`firhalfband` | `firgr` | `firls` | `firminphase` | `rcosdesign` | `firls`

Introduced in R2011a

firpr2chfb

Two-channel FIR filter bank for perfect reconstruction

Syntax

```
[h0,h1,g0,g1] = firpr2chfb(n,fp)
[h0,h1,g0,g1] = firpr2chfb(n,dev,'dev')
[h0,h1,g0,g1] = firpr2chfb('minorder',fp,dev)
```

Description

`[h0,h1,g0,g1] = firpr2chfb(n,fp)` designs four FIR filters for the analysis sections (`h0` and `h1`) and synthesis section is (`g0` and `g1`) of a two-channel perfect reconstruction filter bank. The design corresponds to the orthogonal filter banks also known as power-symmetric filter banks.

`n` is the order of all four filters. It must be an odd integer. `fp` is the passband-edge for the lowpass filters `h0` and `g0`. The passband-edge argument `fp` must be less than 0.5. `h1` and `g1` are highpass filters with the passband-edge given by (1-`fp`).

`[h0,h1,g0,g1] = firpr2chfb(n,dev,'dev')` designs the four filters such that the maximum stopband ripple of `h0` is given by the scalar `dev`. Specify `dev` in linear units, not decibels. The stopband-ripple of `h1` is also be given by `dev`, while the maximum stopband-ripple for both `g0` and `g1` is (2*`dev`).

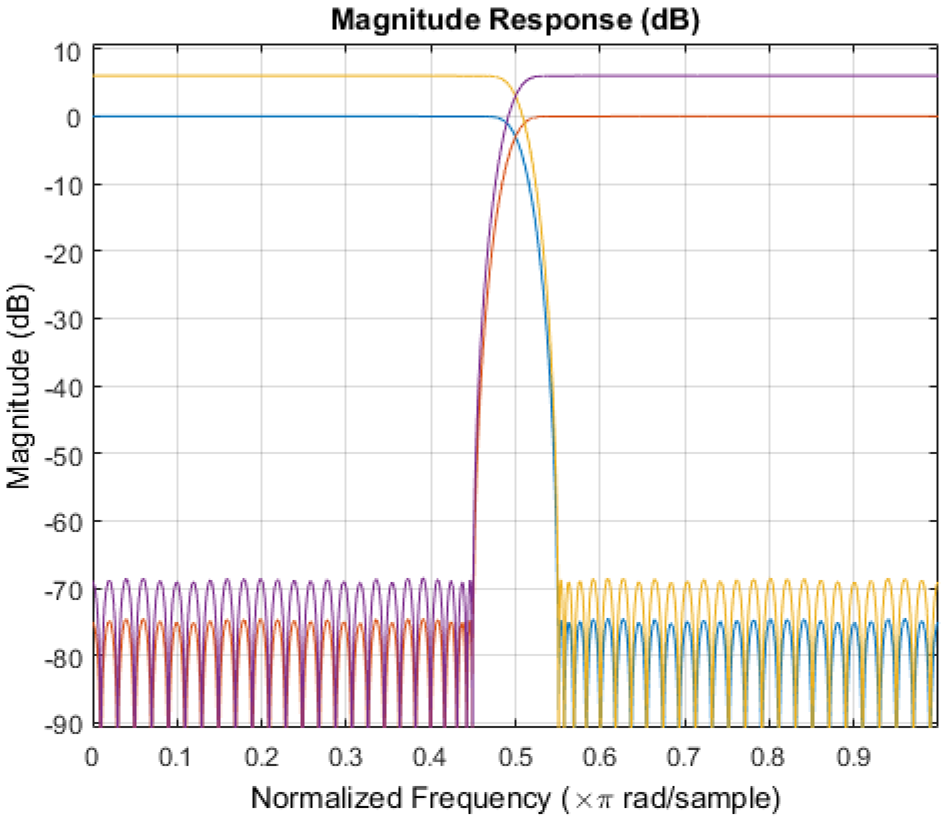
`[h0,h1,g0,g1] = firpr2chfb('minorder',fp,dev)` designs the four filters such that `h0` meets the passband-edge specification `fp` and the stopband-ripple `dev` using minimum order filters to meet the specification.

Examples

Design a Filter Bank Using firpr2chfb

Design a filter bank with filters of order n equal to 99 and passband edges of 0.45 and 0.55.

```
n = 99;
[h0,h1,g0,g1] = firpr2chfb(n,.45);
fvtool(h0,1,h1,1,g0,1,g1,1);
```

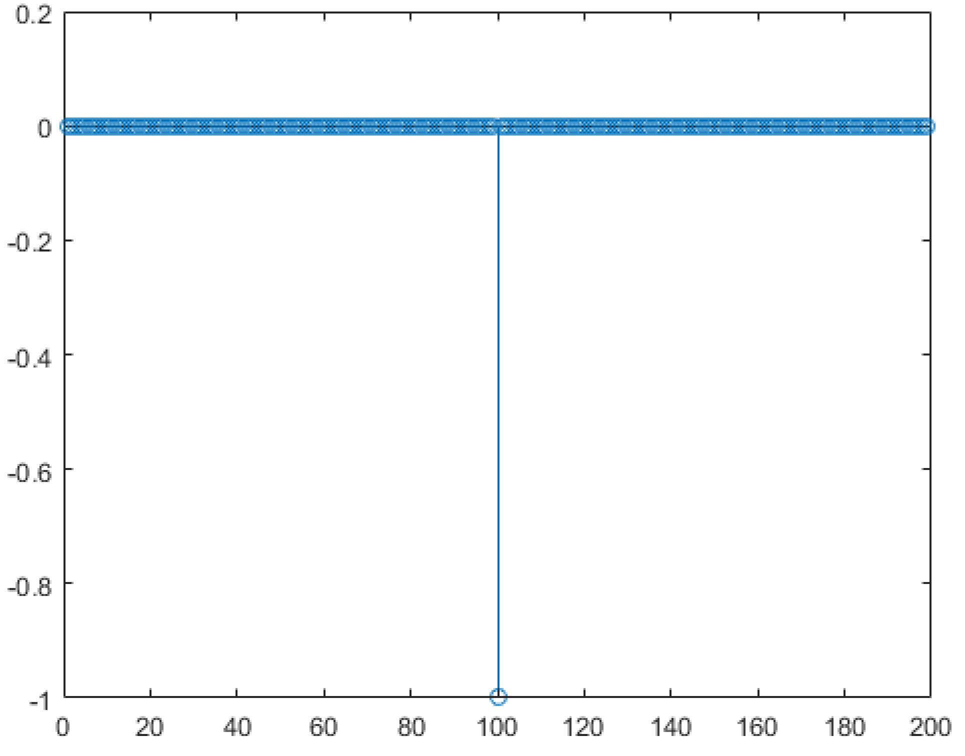


Here are the filters, showing clearly the passband edges.

Use the following stem plots to verify perfect reconstruction using the filter bank created by firpr2chfb.

```
stem(1/2*conv(g0,h0)+1/2*conv(g1,h1))
n=0:n;
stem(1/2*conv((-1).^n.*h0,g0)+1/2*conv((-1).^n.*h1,g1))
stem(1/2*conv((-1).^n.*g0,h0)+1/2*conv((-1).^n.*g1,h1))
stem(1/2*conv((-1).^n.*g0,(-1).^n.*h0)+...
```

```
1/2*conv((-1).^n.*g1,(-1).^n.*h1))
stem(conv((-1).^n.*h1,h0)-conv((-1).^n.*h0,h1))
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

`firceqrip` | `firgr` | `firhalfband` | `firnyquist`

Introduced in R2011a

firtype

Type of linear phase FIR filter

Syntax

```
t = firtype(b)
t = firtype(d)
```

Description

`t = firtype(b)` determines the type, `t`, of an FIR filter with coefficients `b`. `t` can be 1, 2, 3, or 4. The filter must be real and have linear phase.

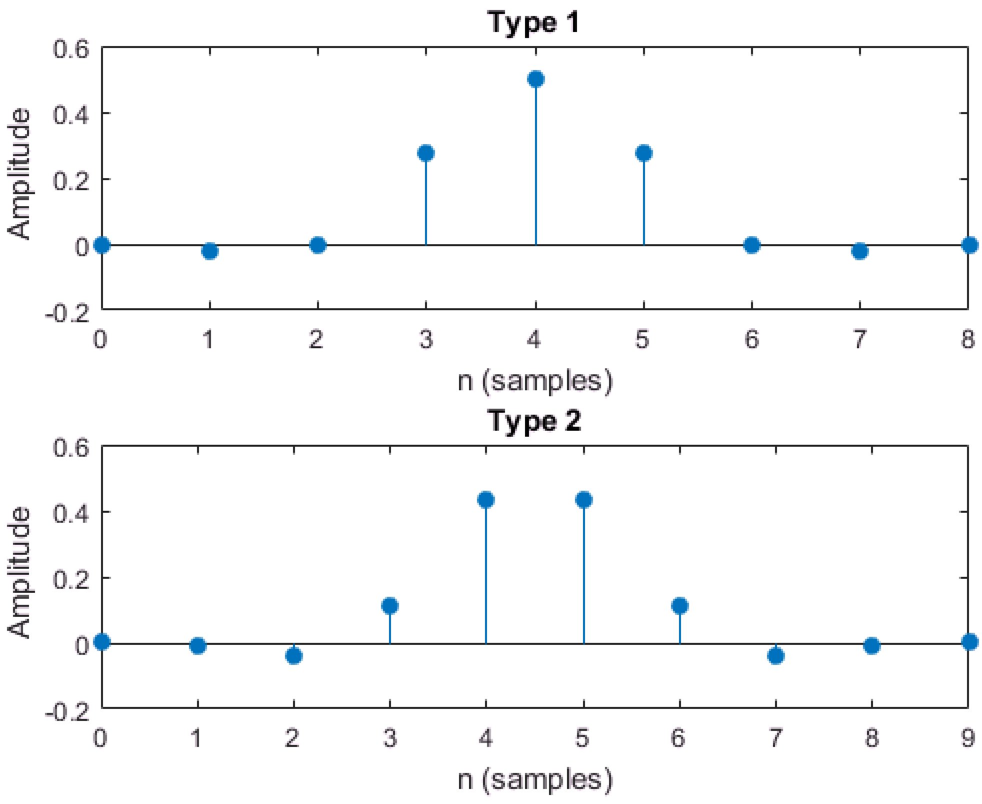
`t = firtype(d)` determines the type, `t`, of an FIR filter, `d`. `t` can be 1, 2, 3, or 4. The filter must be real and have linear phase.

Examples

Types of Linear Phase Filters

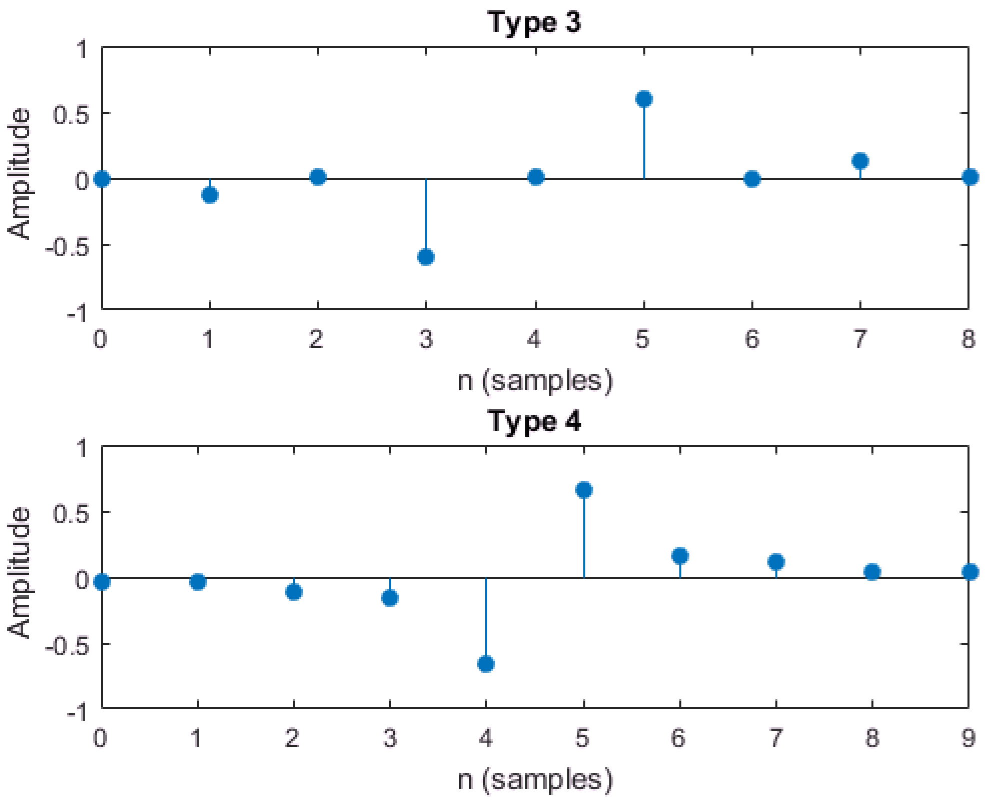
Design two FIR filters using the window method, one of even order and the other of odd order. Determine their types and plot their impulse responses.

```
subplot(2,1,1)
b = fir1(8,0.5);
impz(b), title(['Type ' int2str(firtype(b))])

subplot(2,1,2)
b = fir1(9,0.5);
impz(b), title(['Type ' int2str(firtype(b))])
```



Design two equiripple Hilbert transformers, one of even order and the other of odd order. Determine their types and plot their impulse responses.

```
subplot(2,1,1)
b = firpm(8,[0.2 0.8],[1 1],'hilbert');
impz(b), title(['Type ' int2str(firtype(b))])


subplot(2,1,2)
b = firpm(9,[0.2 0.8],[1 1],'hilbert');
impz(b), title(['Type ' int2str(firtype(b))])
```



Types of FIR digitalFilter Objects

Use `designfilt` to design the filters from the previous example. Display their types.

```
d1 = designfilt('lowpassfir','DesignMethod','window', ...
                'FilterOrder',8,'CutoffFrequency',0.5);
disp(['d1 is of type ' int2str(firtype(d1))])
```

```
d1 is of type 1
```

```
d2 = designfilt('lowpassfir','DesignMethod','window', ...
                'FilterOrder',9,'CutoffFrequency',0.5);
disp(['d2 is of type ' int2str(firtype(d2))])
```

```
d2 is of type 2
```

```
d3 = designfilt('hilbertfir','DesignMethod','equiripple', ...
                'FilterOrder',8,'TransitionWidth',0.4);
disp(['d3 is of type ' int2str(firtype(d3))])
```

```
d3 is of type 3
```

```
d4 = designfilt('hilbertfir','DesignMethod','equiripple', ...
                'FilterOrder',9,'TransitionWidth',0.4);
disp(['d4 is of type ' int2str(firtype(d4))])
```

```
d4 is of type 4
```

Input Arguments

b — Filter coefficients
vector

Filter coefficients of the FIR filter, specified as a double- or single-precision real-valued row or column vector.

Data Types: `double` | `single`

d — FIR filter
`digitalFilter` object | filter System object

FIR filter, specified as any of the following:

- A digitalFilter object. Use `designfilt` to generate a digital filter based on frequency-response specifications.
- A Filter System object. You can use this input if you have a license for DSP System Toolbox software. `firtype` supports the following Filter System objects:

dsp.LowpassFilter
dsp.HighpassFilter
dsp.FIRFilter
dsp.FIRRateConverter
dsp.FIRDecimator

dsp.FIRInterpolator
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.CICDecimator
dsp.CICInterpolator
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.VariableBandwidthFIRFilter

Output Arguments

t — Filter type

1 | 2 | 3 | 4

Filter type, returned as either 1, 2, 3, or 4. Filter types are defined as follows:

- Type 1 — Even-order symmetric coefficients
- Type 2 — Odd-order symmetric coefficients
- Type 3 — Even-order antisymmetric coefficients
- Type 4 — Odd-order antisymmetric coefficients

See Also

See Also

`designfilt` | digitalFilter | `islinphase`

Introduced in R2013a

freqrespest

Frequency response estimate via filtering

Syntax

```
[h,w] = freqrespest(sysobj,L)
[h,w] = freqrespest(sysobj,L,param1,value1,param2,value2,...)
[h,w] = freqrespest(sysobj,L,opts)
freqrespest(sysobj,...)
```

Description

`[h,w] = freqrespest(sysobj,L)` estimates the frequency response of a filter System object, `sysobj`. A set of input data is filtered and then forming the ratio between output data and input data. The test input data comprises sinusoids with uniformly distributed random frequencies.

Use this technique for comparing the performance of fixed-point filters to that of another filter type. You can, for example, compare fixed—point frequency response estimate to that of a similar filter that uses quantized coefficients, but applies floating-point arithmetic internally. Such comparison determines whether the fixed-point filter performance closely matches the floating-point, quantized coefficients version of the filter.

`L` is the number of trials to use to compute the estimate. If you do not specify this value, `L` defaults to 10. More trials generates a more accurate estimate of the response, but require more time to compute the estimate.

`h` is the estimate of the complex frequency response. `w` contains the vector of frequencies at which `h` is estimated.

`[h,w] = freqrespest(sysobj,L,param1,value1,param2,value2,...)` uses parameter value (PV) pairs as input arguments to specify optional parameters for the test. These parameters are the valid PV pairs. Enter the parameter names as input arguments in single quotation marks. The following table provides valid parameters for `[h, w]`.

Parameter Name	Default Value	Description
NFFT	512	Number of FFT points to use.
NormalizedFrequency	true	Indicate whether to use normalized frequency or linear frequency. Values are true (use normalized frequency), or false (use linear frequency). When you specify false, you must supply the sampling frequency Fs.
Fs	normalized	Specify the sampling frequency when NormalizedFrequency is false. No default value. You must set NormalizedFrequency to false before setting a value for Fs.
SpectrumRange	half	Specify the spectrum range to be used as whole or half (default).
CenterDC	false	Specify whether to set the center of the spectrum to the DC value in the output plot. If you select true, both the negative and positive values appear in the plot. If you select false, DC appears at the origin of the axes.
Arithmetic	ARITH	Analyze the filter System object, based on the arithmetic specified in the ARITH input. ARITH can be set to double, single, or fixed. The analysis tool assumes a double-precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state.

freqrespest requires knowledge of the input data type. Analysis cannot be performed if the input data type is not available. If you do not specify the Arithmetic parameter, i.e., use the syntax [h,w] = freqrespest(sysobj) , then the following rules apply for this method:

- The System object state is Unlocked — freqrespest performs double precision analysis.

- The System object state is `Locked` — `freqrespest` performs analysis based on the locked input data type.

When you do specify the `Arithmetic` parameter, i.e., use the syntax `[h,w] = freqrespest(sysobj,'Arithmetic', ARITH)`, review the following rules for this method. Which rule applies depends on the value you set for the `Arithmetic` parameter.

Value	**System Object State**	**Rule**
`ARITH = ’double’`	`Unlocked`	`freqrespest` performs double-precision analysis.
	`Locked`	`freqrespest` performs double-precision analysis.
`ARITH = ‘single’`	`Unlocked`	`freqrespest` performs single-precision analysis.
	`Locked`	`freqrespest` performs single-precision analysis.
`ARITH = ‘fixed’`	`Unlocked`	`freqrespest` produces an error because the fixed-point input data type is unknown.
	`Locked`	If the input data type is double or single, then `freqrespest` produces an error because the fixed-point input data type is unknown.
		When the input data is of fixed-point type, `freqrespest` performs analysis based on the locked input data type.

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpoleFilter
dsp.AllpassFilter
dsp.BiquadFilter

Filter System objects
dsp.CoupledAllpassFilter
dsp.FIRFilter
dsp.HighpassFilter
dsp.IIRFilter
dsp.LowpassFilter
dsp.NotchPeakFilter
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

`[h,w] = freqrespest(sysobj,L,opts)` uses an object, `opts`, to specify the optional input parameters. This specification is not done directly by specifying PV pairs as input arguments. Create `opts` with

```
opts = freqrespopts(sysobj);
```

Because `opts` is an object, you use `set` to change the parameter values in `opts` before you use it with `freqrespest`. For example, you could specify a new sample rate with

```
set(opts,'fs',48e3); % Same as opts.fs=48e3
```

`freqrespest(sysobj,...)` with no output argument launches FVTool.

`freqrespest` can also compute the frequency response of double-precision floating filters. Such filters cannot be converted to transfer-function form without introducing significant round off errors which affect the `freqz` frequency response computation. Examples of these kinds of filters include state-space or lattice filters, especially if they are high-order filters.

Examples

Estimate the Frequency Response of Fixed-Point FIR Filter

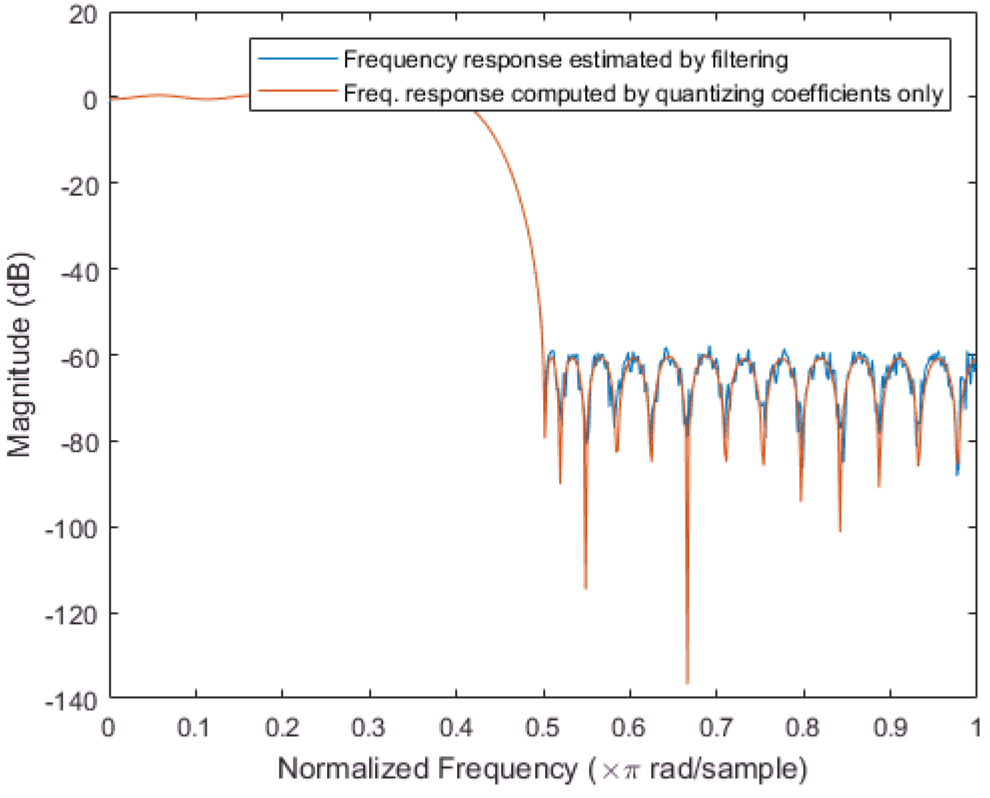
This example shows to how to estimate the frequency response of a fixed-point FIR filter.

```
firFilt = design(fdesign.lowpass(.4,.5,1,60),'equiripple','Systemobject',true);
```

```
dataIn = fi(randn(16,15),1,16,15);
dataOut = firFilt(dataIn);
[h,w] = freqrespest(firFilt); % This should be about the same as freqz.
release(firFilt);
```

Continuing with filter `firFilt`, change `FullPrecisionOverride` to `false` and then specify the word lengths and precision (the fraction lengths) applied to the output from internal addition and multiplication operations. After you set the word and fraction lengths, use `freqrespest` to compute the frequency response estimate for the fixed-point filter.

```
firFilt.FullPrecisionOverride = false;
firFilt.ProductDataType = 'Custom';
firFilt.CustomProductDataType = numerictype(1,16,15);
firFilt.AccumulatorDataType = 'Custom';
firFilt.CustomAccumulatorDataType = numerictype(1,16,15);
firFilt.OutputDataType = 'Same as accumulator';
dataOut = firFilt(dataIn);
[h,w] = freqrespest(firFilt,2);
[h2,w2] = freqz(firFilt,512);
plot(w/pi,20*log10(abs([h,h2])))
legend('Frequency response estimated by filtering',...
'Freq. response computed by quantizing coefficients only');
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Magnitude (dB)')
```



See Also

See Also

Functions
`freqrespopts` | `freqz` | `limitcycle` | `noisepsd` | `scale`

Introduced in R2011a

freqrespopts

Options for filter frequency response analysis

Syntax

```
opts = freqrespopts(H)
```

Description

`opts = freqrespopts(H)` uses the settings in the filter System object, `H`, to create an object, `opts`. This object contains parameters and values for estimating the filter frequency response. You pass `opts` as an input argument to `freqrespest` to specify values for the input parameters.

`freqrespopts` allows you to use the same settings for `freqrespest` with multiple filters without specifying all of the parameters as input arguments to `freqrespest`.

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpoleFilter
dsp.AllpassFilter
dsp.BiquadFilter
dsp.CoupledAllpassFilter
dsp.FIRFilter
dsp.HighpassFilter
dsp.IIRFilter
dsp.LowpassFilter
dsp.NotchPeakFilter
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

In addition, dsp.FarrowRateConverter and dsp.FilterCascade support this analysis method.

Examples

Set the Frequency Response Options

This example shows freqrespopts in use for setting options for freqrespest. hd and hd2 are bandpass filters that use different design methods. The opts object makes it easier to set the same conditions for the frequency response estimate in freqrespest.

```
d = fdesign.bandpass('fst1,fp1,fp2,fst2,ast1,ap,ast2',...
0.25,0.3,0.45,0.5,60,0.1,60);
hd = design(d,'butter','SystemObject',true)
```

```
hd = 

  dsp.BiquadFilter with properties:

                 Structure: 'Direct form II'
           SOSMatrixSource: 'Property'
                 SOSMatrix: [18×6 double]
               ScaleValues: [19×1 double]
         InitialConditions: 0
    OptimizeUnityScaleValues: true

  Use get to show all properties
```

```
hd2 = design(d,'cheby2','SystemObject',true)
```

```
hd2 = 

  dsp.BiquadFilter with properties:

                 Structure: 'Direct form II'
           SOSMatrixSource: 'Property'
                 SOSMatrix: [9×6 double]
               ScaleValues: [10×1 double]
         InitialConditions: 0
    OptimizeUnityScaleValues: true
```

```
  Use get to show all properties

opts = freqrespopts(hd)

opts =

  <a href="matlab:helpPopup struct" style="font-weight:bold">struct</a> with fields:

             FreqPoints: 'All'
                   NFFT: 512
    NormalizedFrequency: true
                     Fs: 'Normalized'
          SpectrumRange: 'Half'
               CenterDC: false

opts.NFFT=256; % Same as set(opts,'nfft',256).
opts.NormalizedFrequency=false;
opts.fs=1.5e3;
opts.CenterDC=true

opts =

  <a href="matlab:helpPopup struct" style="font-weight:bold">struct</a> with fields:

             FreqPoints: 'All'
                   NFFT: 256
    NormalizedFrequency: false
                     Fs: 1500
          SpectrumRange: 'Whole'
               CenterDC: true
```

With opts configured as needed, use it as an input argument for freqrespest.

```
[h2,w2]=freqrespest(hd2,20,opts);
[h1,w1]=freqrespest(hd,20,opts);
```

See Also

freqrespest | norm | scale | noisepsd | noisepsdopts

Introduced in R2011a

freqsamp

Real or complex frequency-sampled FIR filter from specification object

Syntax

```
hd = design(d,'freqsamp','SystemObject',true)
hd = design(...,'filterstructure',structure,'SystemObject',true)
hd = design(...,'window',window,'SystemObject',true)
```

Description

`hd = design(d,'freqsamp','SystemObject',true)` designs a frequency-sampled filter specified by the filter specifications object `d`.

`hd = design(...,'filterstructure',structure,'SystemObject',true)` returns a filter with the filter structure you specify by the `structure` input argument. `structure` is `dffir` by default and can be any one of the following filter structures.

Structure	Description of Resulting Filter Structure
`dffir`	Direct-form FIR filter
`dffirt`	Transposed direct-form FIR filter
`dfsymfir`	Symmetrical direct-form FIR filter
`dfasymfir`	Asymmetrical direct-form FIR filter

`hd = design(...,'window',window,'SystemObject',true)` designs filters using the window specified by `window`. Provide the input argument `window` as

- A character vector for the window type. For example, use `'bartlett'`, or `'hamming'`. See `window` for the full list of windows available or refer to `window` in the *Signal Processing Toolbox User's Guide*.
- A function handle that references the `window` function. When the `window` function requires more than one input, use a cell array to hold the required arguments. The first example shows a cell array input argument.

- The window vector itself.

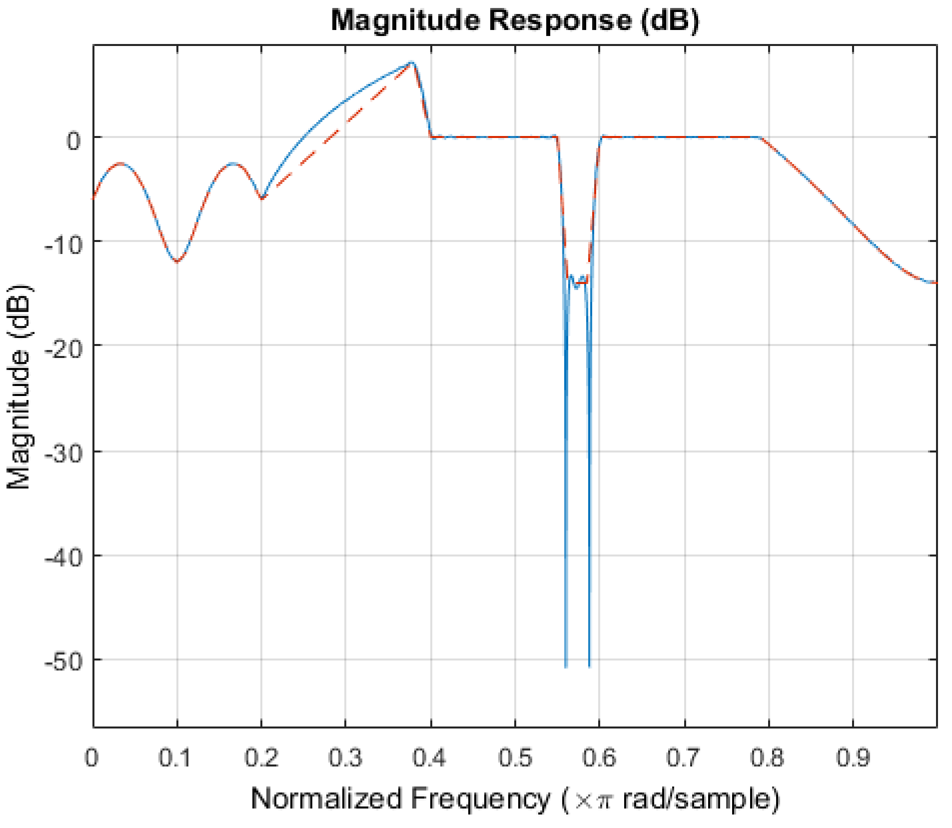
Examples

Design a Frequency Sampled FIR Filter

These examples design FIR filters that have arbitrary magnitude responses. In the first filter, the response has three distinct sections and the resulting filter is real.

The second example creates a complex filter.

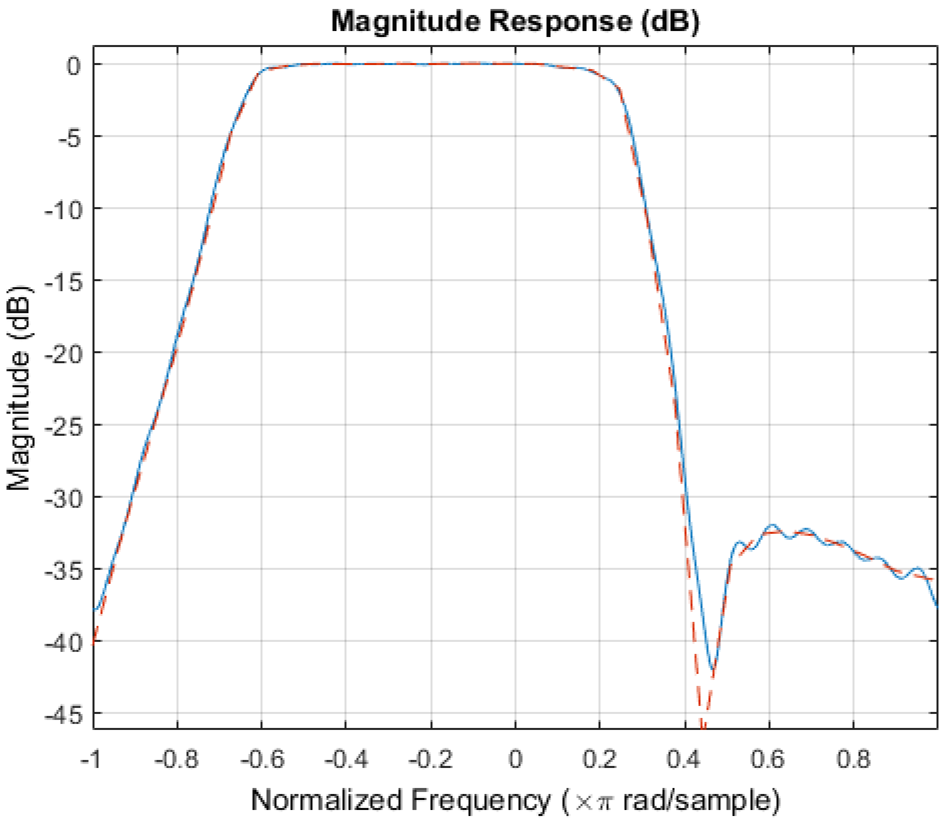
```
b1 = 0:0.01:0.18;
b2 = [.2 .38 .4 .55 .562 .585 .6 .78];
b3 = [0.79:0.01:1];
a1 = .5+sin(2*pi*7.5*b1)/4;    % Sinusoidal response section.
a2 = [.5 2.3 1 1 -.2 -.2 1 1]; % Piecewise linear response section.
a3 = .2+18*(1-b3).^2;          % Quadratic response section.
f = [b1 b2 b3];
a = [a1 a2 a3];
n = 300;
d = fdesign.arbmag('n,f,a',n,f,a); % First specifications object.
hd = design(d,'freqsamp','window',{@kaiser,.5},...
    'SystemObject',true); % Filter.
fvtool(hd)
```



Now design the arbitrary-magnitude complex FIR filter. Recall that vector f contains frequency locations and vector a contains the desired filter response values at the locations specified in f.

```
f = [-1 -.93443 -.86885 -.80328 -.7377 -.67213 -.60656 -.54098 ...
-.47541,-.40984 -.34426 -.27869 -.21311 -.14754 -.081967 ...
-.016393 .04918 .11475,.18033 .2459 .31148 .37705 .44262 ...
 .5082 .57377 .63934 .70492 .77049,.83607 .90164 1];
a = [.0095848 .021972 .047249 .099869 .23119 .57569 .94032 ...
.98084 .99707,.99565 .9958 .99899 .99402 .99978 .99995 .99733 ...
.99731 .96979 .94936,.8196 .28502 .065469 .0044517 .018164 ...
.023305 .02397 .023141 .021341,.019364 .017379 .016061];
n = 48;
d = fdesign.arbmag('n,f,a',n,f,a); % Second spec. object.
```

```
hdc = design(d,'freqsamp','window','rectwin','SystemObject',true); % Filter.
fvtool(hdc)
```



fvtool shows you the response for hdc from -1 to 1 in normalized frequency. design(d,...) returns a complex filter for hdc because the frequency vector includes negative frequency values.

See Also

design | designmethods | help | window | fdesign.arbmag

Introduced in R2011a

freqz

Frequency response of filter

Syntax

```
[h,w] = freqz(sysobj)
[h,w] = freqz(sysobj,n)
[h,w] = freqz(sysobj,Name,Value)
freqz(sysobj)
```

Description

`[h,w] = freqz(sysobj)` returns the complex frequency response `h` of the filter System object, `sysobj`. The vector `w` contains the frequencies (in radians) at which the function evaluates the frequency response. The frequency response is evaluated at 8192 points equally spaced around the upper half of the unit circle.

`[h,w] = freqz(sysobj,n)` returns the complex frequency response of the filter System object and the corresponding frequencies at `n` points equally spaced around the upper half of the unit circle.

`freqz` uses the transfer function associated with the filter to calculate the frequency response of the filter with the current coefficient values.

`[h,w] = freqz(sysobj,Name,Value)` returns a frequency response with additional options specified by one or more `Name,Value` pair arguments.

`freqz(sysobj)` uses `fvtool` to plot the magnitude and unwrapped phase of the frequency response of the filter System object `sysobj`.

For more information about optional input arguments for `freqz`, refer to `freqz` in Signal Processing Toolbox documentation.

Input Arguments

`sysobj`

Filter System object.

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpassFilter
dsp.AllpoleFilter
dsp.BiquadFilter
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.CICDecimator
dsp.CICInterpolator
dsp.CoupledAllpassFilter
dsp.FarrowRateConverter
dsp.FilterCascade
dsp.FIRDecimator
dsp.FIRFilter
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.FIRInterpolator
dsp.FIRRateConverter
dsp.HighpassFilter
dsp.IIRFilter
dsp.IIRHalfbandDecimator
dsp.IIRHalfbandInterpolator
dsp.LowpassFilter
dsp.NotchPeakFilter
dsp.VariableBandwidthFIRFilter

Filter System objects
dsp.VariableBandwidthIIRFilter

dsp.SampleRateConverter is also supported by `freqz`.

n

Number of samples. For an FIR filter where `n` is a power of two, the computation is done faster using FFTs.

Default: 8192

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Arithmetic'` — Value types:
`'double'` | `'single'` | `'fixed'`

Specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

System Object State	Coefficient Data Type	Rule
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

h

Complex `n`-element frequency response vector. If `n` is not specified, the function uses a default value of 8192.

w

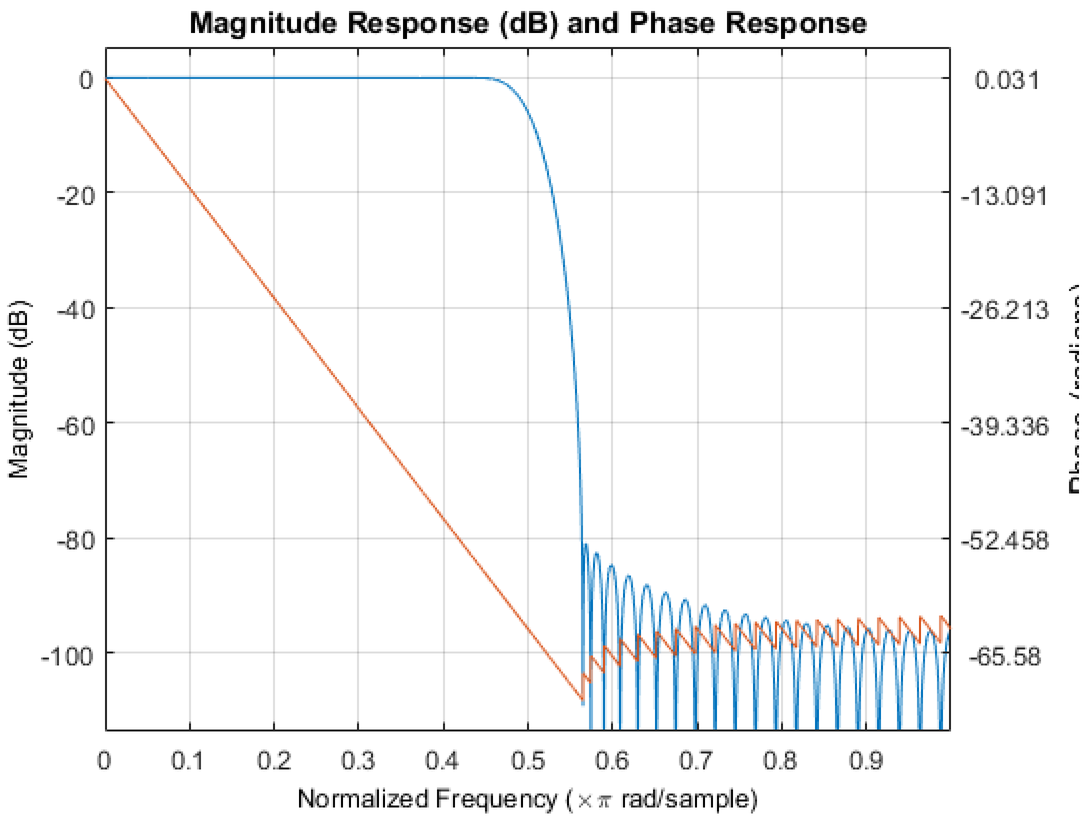
Frequency vector of length `n`, in radians/sample. `w` consists of `n` points equally spaced around the upper half of the unit circle (from 0 to π radians/sample). If `n` is not specified, the function uses a default value of 8192.

Examples

Frequency Response of the Filter

This examples plot the frequency response of the lowpass FIR filter using `freqz`.

```
b = fir1(80,0.5,kaiser(81,8));
firFilt = dsp.FIRFilter('Numerator',b);
freqz(firFilt);
```



Tips

There are several ways of analyzing the frequency response of filters. `freqz` accounts for quantization effects in the filter coefficients, but does not account for quantization effects in filtering arithmetic. To account for the quantization effects in filtering arithmetic, refer to function `noisepsd`.

Algorithms

freqz calculates the frequency response for a filter from the filter transfer function $Hq(z)$. The complex-valued frequency response is calculated by evaluating $Hq(e^{j\omega})$ at discrete values of w specified by the syntax you use. The integer input argument n determines the number of equally-spaced points around the upper half of the unit circle at which freqz evaluates the frequency response. The frequency ranges from 0 to π radians per sample when you do not supply a sampling frequency as an input argument. When you supply the scalar sampling frequency fs as an input argument to freqz, the frequency ranges from 0 to fs/2 Hz.

See Also

See Also

Functions
fvtool

Introduced in R2011a

gain

CIC filter gain

Syntax

```
gain(hs)
gain(hs,j)
```

Description

`gain(hs)` returns the gain of the filter System object `hs`. The function supports the `dsp.CICDecimator` and `dsp.CICInterpolator` filter structures.

When `hs` is a decimator, `gain` returns the gain for the overall CIC decimator.

When `hs` is an interpolator, the CIC interpolator inserts zeros into the input data stream, reducing the filter overall gain by $1/R$, where R is the interpolation factor, to account for the added zero valued samples. Therefore, the gain of a CIC interpolator is $(RM)^N/R$, where N is the number of filter sections and M is the filter differential delay. `gain(hs)` returns this value. The next example presents this case.

`gain(hs,j)` returns the gain of the `j`th section of a CIC interpolation filter. When you omit `j`, `gain` assumes that `j` is 2*N, where N is the number of sections, and returns the gain of the last section of the filter. This syntax does not apply when `hs` is a decimator.

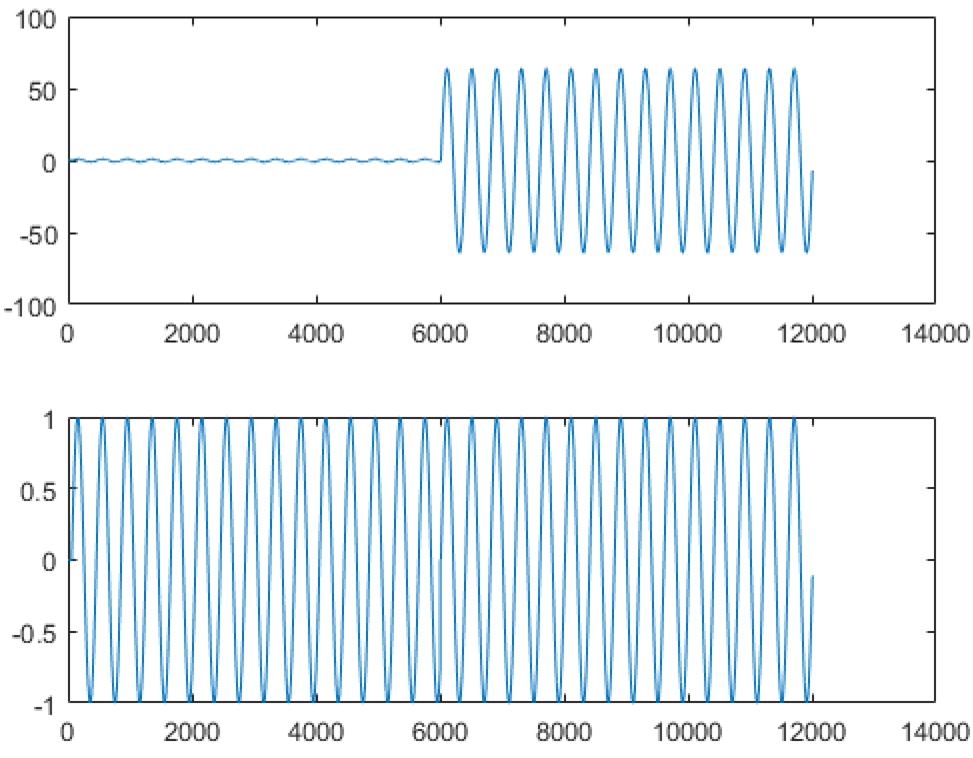
Examples

Compare the Performance of Interpolation Filters

To compare the performance of two interpolators, one a CIC filter and the other an FIR filter, use gain to adjust the CIC filter output amplitude to match the FIR filter output amplitude. Start by creating an input data set, a sinusoidal signal x.

```
fs = 1000;          % Input sampling frequency.
t = 0:1/fs:1.5;     % Signal length = 1501 samples.
x = sin(2*pi*10*t); % Amplitude = 1 sinusoid.
x = x';
```

```
l = 4; % Interpolation factor for FIR filter.
d = fdesign.interpolator(l);
firInterp = design(d,'multistage','SystemObject',true);
yfir = firInterp(x);

r = 4; % Interpolation factor for the CIC filter.
d = fdesign.interpolator(r,'cic');
cicInterp = design(d,'multisection','SystemObject',true);
ycic = cicInterp(x);
gaincic = gain(cicInterp);
subplot(211);
plot([yfir; double(ycic)]);
subplot(212)
plot([yfir; double(ycic)/gain(cicInterp)]);
```



After correcting for the gain induced by the CIC interpolator, the second subplot shows that the FIR filter and the CIC filter provide nearly identical interpolation.

See Also

`scale`

Introduced in R2011a

grpdelay

Group delay response of discrete-time filter System object

Syntax

```
[gd,w] = grpdelay(sysobj)
[gd,w] = grpdelay(sysobj,n)
[gd,w] = grpdelay(sysobj,Name,Value)
grpdelay(sysobj)
```

Description

`[gd,w] = grpdelay(sysobj)` returns the group delay `gd` of the filter System object, `sysobj`, based on the current filter coefficients. The vector `w` contains the frequencies (in radians) at which the group delay is evaluated. The group delay is defined as:

$$-\frac{d}{dw}(angle(w))$$

The group delay is evaluated at 8192 points equally spaced around the upper half of the unit circle.

`[gd,w] = grpdelay(sysobj,n)` returns the group delay of the filter System object and the corresponding frequencies at `n` points equally spaced around the upper half of the unit circle.

`[gd,w] = grpdelay(sysobj,Name,Value)` returns the group delay with additional options specified by one or more `Name,Value` pair arguments.

`grpdelay(sysobj)` plots the group delay of the filter System object `sysobj` in the `fvtool`.

For information on more input options, refer to `grpdelay` in Signal Processing Toolbox documentation.

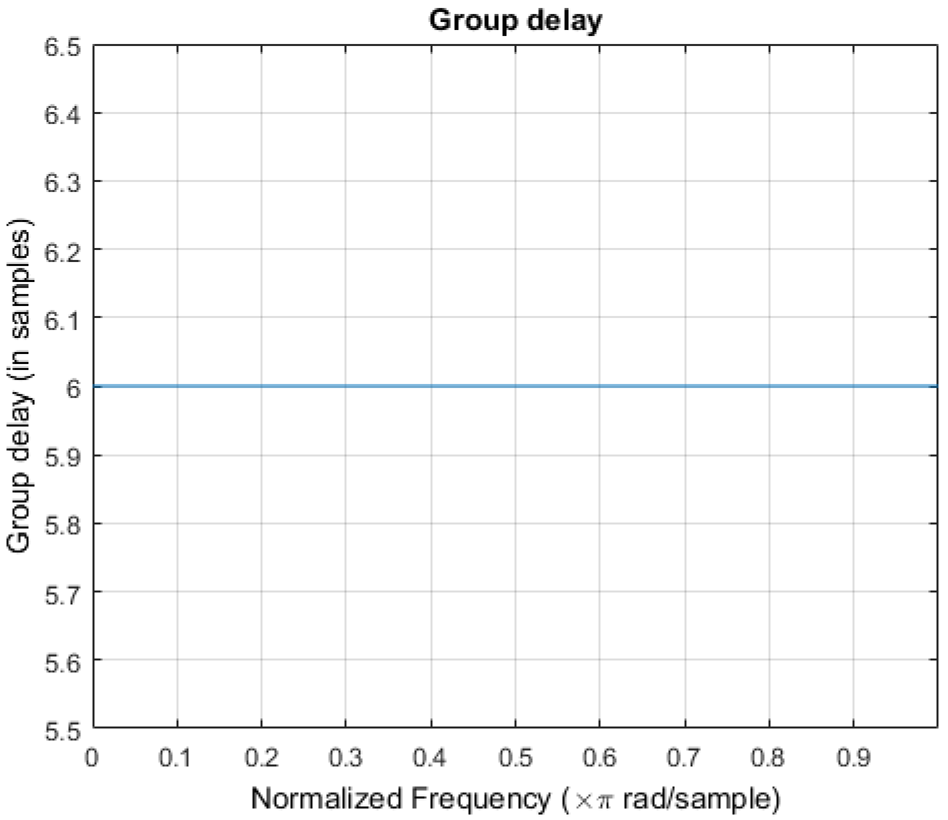
Examples

Group Delay of a Discrete-Time Multirate Filter

```
CICComp = dsp.CICCompensationDecimator;
```

`grpdelay` computes the group delay of the filter and displays it using fvtool

```
grpdelay(CICComp);
```



Input Arguments

sysobj

Filter System object.

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpassFilter

Filter System objects
dsp.AllpoleFilter
dsp.BiquadFilter
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.CICDecimator
dsp.CICInterpolator
dsp.CoupledAllpassFilter
dsp.FarrowRateConverter
dsp.FilterCascade
dsp.FIRDecimator
dsp.FIRFilter
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.FIRInterpolator
dsp.FIRRateConverter
dsp.HighpassFilter
dsp.IIRFilter
dsp.IIRHalfbandDecimator
dsp.IIRHalfbandInterpolator
dsp.LowpassFilter
dsp.NotchPeakFilter
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

n

Number of samples. For an FIR filter where `n` is a power of two, the computation is done faster using FFTs.

Default: 8192

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Arithmetic'` — Value types:

`‘double’ | 'single' | 'fixed'`

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depend on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.

System Object State	Coefficient Data Type	Rule
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

gd

Group delay vector of length n. If n is not specified, the function uses a default value of 8192.

w

Frequency vector of length n, in radians/sample. w consists of n points equally spaced around the upper half of the unit circle (from 0 to π radians/sample). If n is not specified, the function uses a default value of 8192.

See Also

See Also

Functions
phasez | zerophase

Introduced in R2011a

help

Help for design method with filter specification

Syntax

`help(d,'designmethod')`

Description

`help(d,'designmethod')` displays help in the command window for the design algorithm `designmethod` for the current specifications of the filter specification object `d`. `designmethod` must be one of the design algorithms returned by `designmethods` for `d`, the design object.

Examples

Get Help for Designing Butterworth Filters

Get specific help for designing lowpass Butterworth filters.

The first lowpass filter uses the default specification 'Fp,Fst,Ap,Ast' and returns help text specific to the specification.

```
d = fdesign.lowpass;
designmethods(d,'Systemobject',true)
```

```
Design Methods that support System objects for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

```
help(d,'butter')

 DESIGN Design a Butterworth IIR filter.
    HD = DESIGN(D, 'butter') designs a Butterworth filter specified by the
    FDESIGN object D, and returns the DFILT/MFILT object HD.

    HD = DESIGN(D, ..., 'SystemObject', true) implements the filter, HD,
    using a System object instead of a DFILT/MFILT object.

    HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a filter with the
    structure STRUCTURE.  STRUCTURE is 'df2sos' by default and can be any of
    the following:

    'df1sos'
    'df2sos'
    'df1tsos'
    'df2tsos'
    'cascadeallpass'
    'cascadewdfallpass'

    Some of the listed structures may not be supported by System object
    filters. Type validstructures(D, 'butter', 'SystemObject', true) to
    get a list of structures supported by System objects.

    HD = DESIGN(..., 'MatchExactly', MATCH) designs a Butterworth filter
    and matches the frequency and magnitude specification for the band
    MATCH exactly.  The other band will exceed the specification.  MATCH
    can be 'stopband' or 'passband' and is 'stopband' by default.

    HD = DESIGN(..., 'SOSScaleNorm', NORM) designs an SOS filter and scales
    the coefficients using the P-Norm NORM. NORM can be either a
    discrete-time-domain norm or a frequency-domain norm. Valid time-domain
    norms are 'l1','l2', and 'linf'. Valid frequency-domain norms are
    'L1','L2', and 'Linf'. Note that L2-norm is equal to l2-norm
    (Parseval's theorem) but the same is not true for other norms.

    The different norms can be ordered in terms of how stringent they are
    as follows: 'l1' >= 'Linf' >= 'L2' = 'l2' >= 'L1' >= 'linf'.
    Using the most stringent scaling, 'l1', the filter is the least prone
    to overflow, but also has the worst signal-to-noise ratio. Linf-scaling is
    the most commonly used scaling in practice.

    Scaling is turned off by default, which is equivalent to setting
```

```
SOSScaleNorm = ''.

HD = DESIGN(..., 'SOSScaleOpts', OPTS) designs an SOS filter and scales
the coefficients using an FDOPTS.SOSSCALING object OPTS.  Scaling options
are:
Property                Default     Description/Valid values
---------               -------     ------------------------
'sosReorder'            'auto'      Reorder section prior to scaling.
                                    {'auto','none','up','down','lowpass',
                                     'highpass','bandpass','bandstop'}
'MaxNumerator'          2           Maximum value for numerator coefficients
'NumeratorConstraint'   'none'      {'none', 'unit', 'normalize','po2'}
'OverflowMode'          'wrap'      {'wrap','saturate'}
'ScaleValueConstraint'  'unit'      {'unit','none','po2'}
'MaxScaleValue'         'Not used'  Maximum value for scale values

When sosReorder is set to 'auto', the sections will be automatically
reordered depending on the response type of the design (lowpass,
highpass, etc.).

Note that 'MaxScaleValue' will only be used when 'ScaleValueConstraint'
is set to something other than 'unit'. If 'MaxScaleValue' is set to a
number, the 'ScaleValueConstraint' will be changed to 'none'. Further,
if SOSScaleNorm is off (as it is by default), then all the SOSScaleOpts
will be ignored.

For more information about P-Norm and scaling options see help for DFILT\SCALE.

% Example #1 - Compare passband and stopband MatchExactly.
   h     = fdesign.lowpass('Fp,Fst,Ap,Ast', .1, .3, 1, 60);
   Hd    = design(h, 'butter', 'MatchExactly', 'passband');
   Hd(2) = design(h, 'butter', 'MatchExactly', 'stopband');

   % Compare the passband edges in FVTool.
   fvtool(Hd);
   axis([.09 .11 -2 0]);
```

Note the discussion of the `MatchExactly` input option. When you use a design object that uses a different specification, such as 'N,F3dB', the help content for the butter design method changes.

In this case, the MatchExactly option does not appear in the help because it is not an available input argument for the specification 'N,F3dB'.

```
d = fdesign.lowpass('N,F3dB')
```

```
d = 

  lowpass with properties:

               Response: 'Lowpass'
          Specification: 'N,F3dB'
            Description: {2×1 cell}
    NormalizedFrequency: 1
            FilterOrder: 10
                   F3dB: 0.5000
```

```
designmethods(d,'Systemobject',true)
```

```
Design Methods that support System objects for class fdesign.lowpass (N,F3dB):


butter
maxflat
```

```
help(d,'butter')
```

```
 DESIGN Design a Butterworth IIR filter.
    HD = DESIGN(D, 'butter') designs a Butterworth filter specified by the
    FDESIGN object D, and returns the DFILT/MFILT object HD.

    HD = DESIGN(D, ..., 'SystemObject', true) implements the filter, HD,
    using a System object instead of a DFILT/MFILT object.

    HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a filter with the
    structure STRUCTURE.  STRUCTURE is 'df2sos' by default and can be any of
    the following:

    'df1sos'
    'df2sos'
    'df1tsos'
    'df2tsos'
    'cascadeallpass'
    'cascadewdfallpass'
```

```
Some of the listed structures may not be supported by System object
filters. Type validstructures(D, 'butter', 'SystemObject', true) to
get a list of structures supported by System objects.

HD = DESIGN(..., 'SOSScaleNorm', NORM) designs an SOS filter and scales
the coefficients using the P-Norm NORM. NORM can be either a
discrete-time-domain norm or a frequency-domain norm. Valid time-domain
norms are 'l1','l2', and 'linf'. Valid frequency-domain norms are
'L1','L2', and 'Linf'. Note that L2-norm is equal to l2-norm
(Parseval's theorem) but the same is not true for other norms.

The different norms can be ordered in terms of how stringent they are
as follows: 'l1' >= 'Linf' >= 'L2' = 'l2' >= 'L1' >= 'linf'.
Using the most stringent scaling, 'l1', the filter is the least prone
to overflow, but also has the worst signal-to-noise ratio. Linf-scaling is
the most commonly used scaling in practice.

Scaling is turned off by default, which is equivalent to setting
SOSScaleNorm = ''.

HD = DESIGN(..., 'SOSScaleOpts', OPTS) designs an SOS filter and scales
the coefficients using an FDOPTS.SOSSCALING object OPTS.  Scaling options
are:
Property                Default     Description/Valid values
---------               -------     ------------------------
'sosReorder'            'auto'      Reorder section prior to scaling.
                                    {'auto','none','up','down','lowpass',
                                     'highpass','bandpass','bandstop'}
'MaxNumerator'          2           Maximum value for numerator coefficients
'NumeratorConstraint'   'none'      {'none', 'unit', 'normalize','po2'}
'OverflowMode'          'wrap'      {'wrap','saturate'}
'ScaleValueConstraint'  'unit'      {'unit','none','po2'}
'MaxScaleValue'         'Not used'  Maximum value for scale values

When sosReorder is set to 'auto', the sections will be automatically
reordered depending on the response type of the design (lowpass,
highpass, etc.).

Note that 'MaxScaleValue' will only be used when 'ScaleValueConstraint'
is set to something other than 'unit'. If 'MaxScaleValue' is set to a
number, the 'ScaleValueConstraint' will be changed to 'none'. Further,
if SOSScaleNorm is off (as it is by default), then all the SOSScaleOpts
will be ignored.
```

```
For more information about P-Norm and scaling options see help for DFILT\SCALE.

% Example #1 - Design a lowpass Butterworth filter in the DF2TSOS structure.
   h  = fdesign.lowpass('N,F3dB');
   Hd = design(h, 'butter', 'FilterStructure', 'df2tsos');
```

See Also

fdesign | design | designmethods | designopts

Introduced in R2011a

ifir

Interpolated FIR filter design

Syntax

```
[h,g] = ifir(l,type,f,dev)
[h,g,d] = ifir(l,type,f,dev)
[...] = ifir(...,str)
```

Description

`[h,g] = ifir(l,type,f,dev)` designs a periodic filter $h(z^{l})$, where `l` is the interpolation factor. It also finds an image-suppressor filter `g(z)`, such that the cascade of the two filters represents the optimal minimax FIR approximation of the desired response. This response is specified by `type`, with band edge frequencies contained in vector `f`. This is done while not exceeding the maximum deviations or ripples (linear) specified in vector `dev`.

When `type` is set to `'low'`, the filter design is a lowpass design. When `type` is set to `'high'`, the filter design is a highpass design. `f` is a two-element vector with passband and stopband edge frequency values. For narrowband lowpass filters and wideband highpass filters, `l×f(2)` is less than `1`. For wideband lowpass filters and narrowband highpass filters, specify `f` so that `l×(1–f(1))` is less than `1`.

`dev` is a two-element vector that contains the peak ripple or deviation (in linear units) allowed for both the passband and the stopband.

The `ifir` design algorithm achieves an efficient design in the sense that it reduces the total number of multipliers required. To do this, the design problem is broken into two stages. In the first stage, the filter is upsampled to achieve the stringent specifications without using many multipliers. In the second stage, the filter removes the images created when upsampling the previous filter.

`[h,g,d] = ifir(l,type,f,dev)` returns a delay `d` that is connected in parallel with the cascade of $h(z^{l})$ and `g(z)` for both wideband lowpass and highpass filters. This is necessary to obtain the desired response.

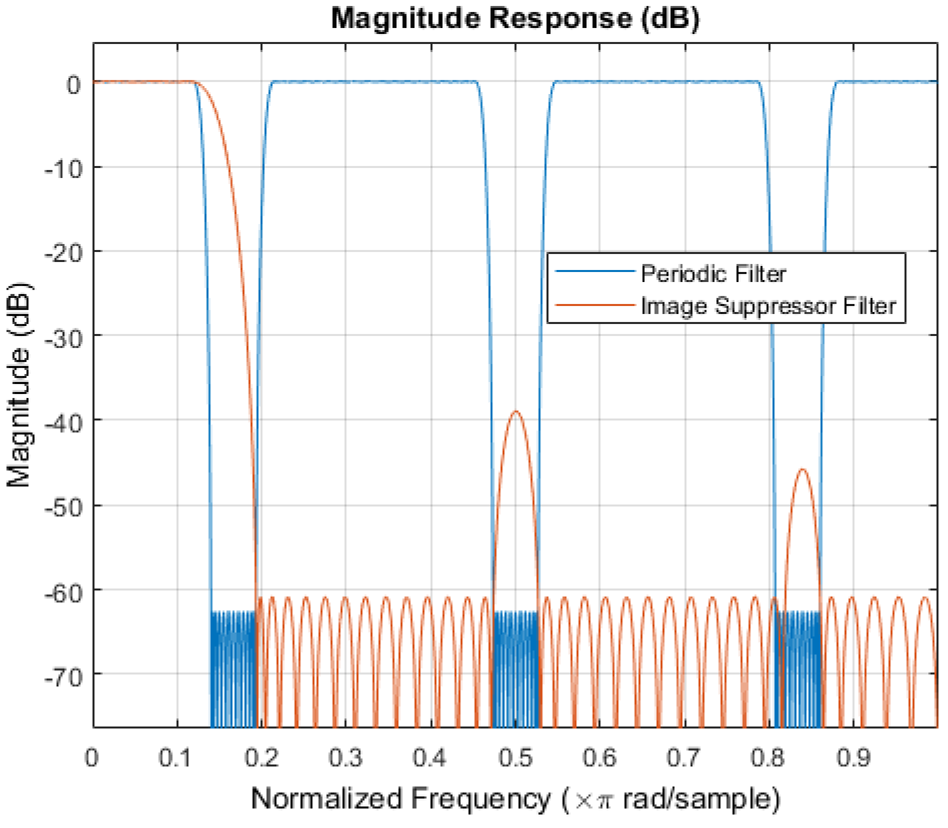
[...] = ifir(...,str) uses str to choose the algorithm level of optimization used. Possible values for str are 'simple', 'intermediate' (default) or 'advanced'. str provides for a tradeoff between design speed and filter order optimization. The 'advanced' option can result in substantial filter order reduction, especially for g(z).

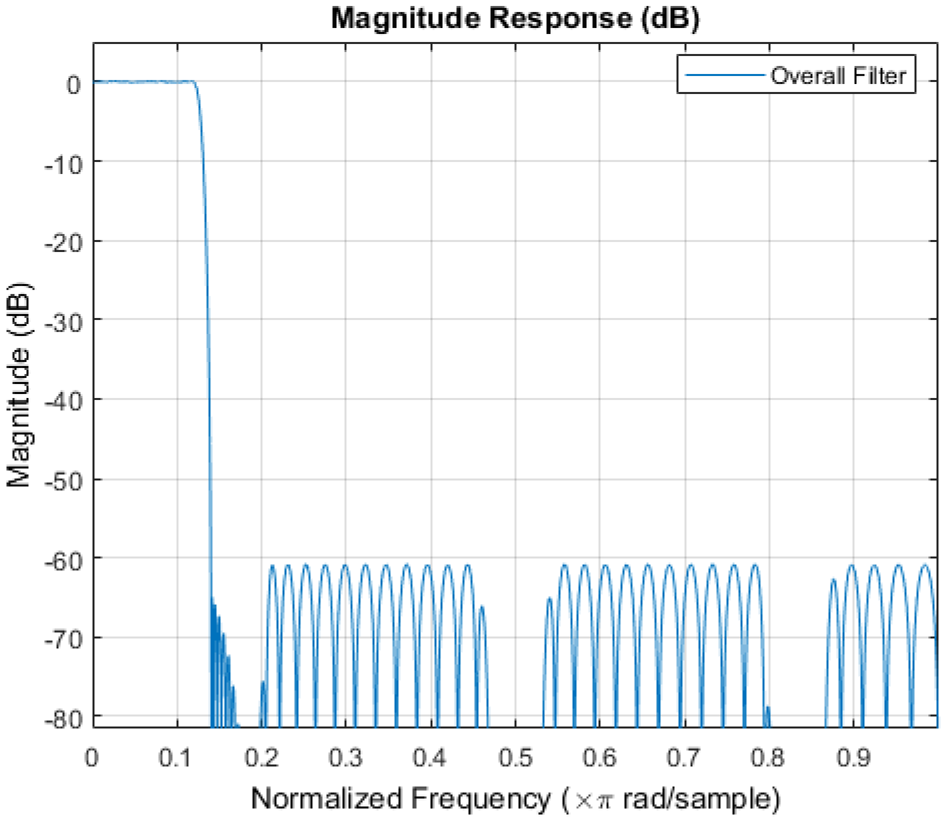
Examples

Narrowband lowpass design using an interpolation factor of 6

This example shows how to use the function ifir to design a narrowband lowpass filter.

```
[h,g]=ifir(6,'low',[.12 .14],[.01 .001]);
H = dsp.FIRFilter('Numerator',h);
G = dsp.FIRFilter('Numerator',g);
hfv = fvtool(H,G);
legend(hfv,'Periodic Filter','Image Suppressor Filter');
Hcas = cascade(H,G);
hfv2 = fvtool(Hcas);
legend(hfv2,'Overall Filter');
```

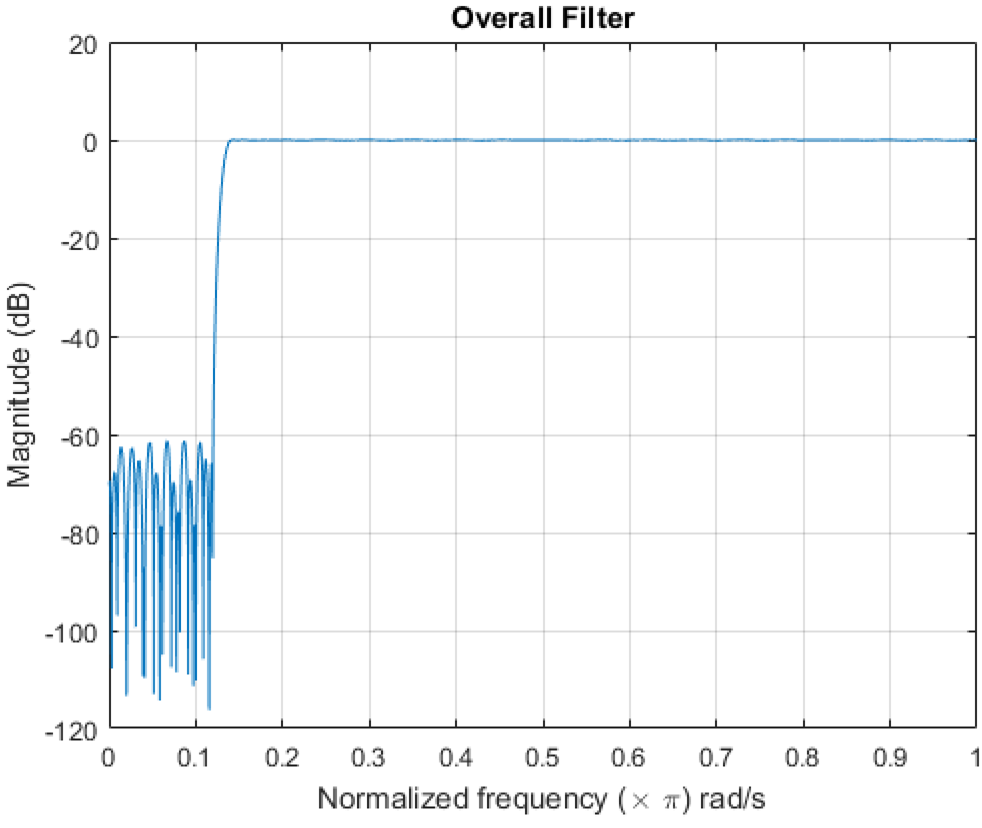

Magnitude Response (dB)
Periodic Filter
Image Suppressor Filter
Magnitude (dB)
0
-10
-20
-30
-40
-50
-60
-70
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Normalized Frequency (×π rad/sample)



Wideband highpass design using an interpolation factor of 6

This example shows how to use `ifir` to design a wideband highpass filter.

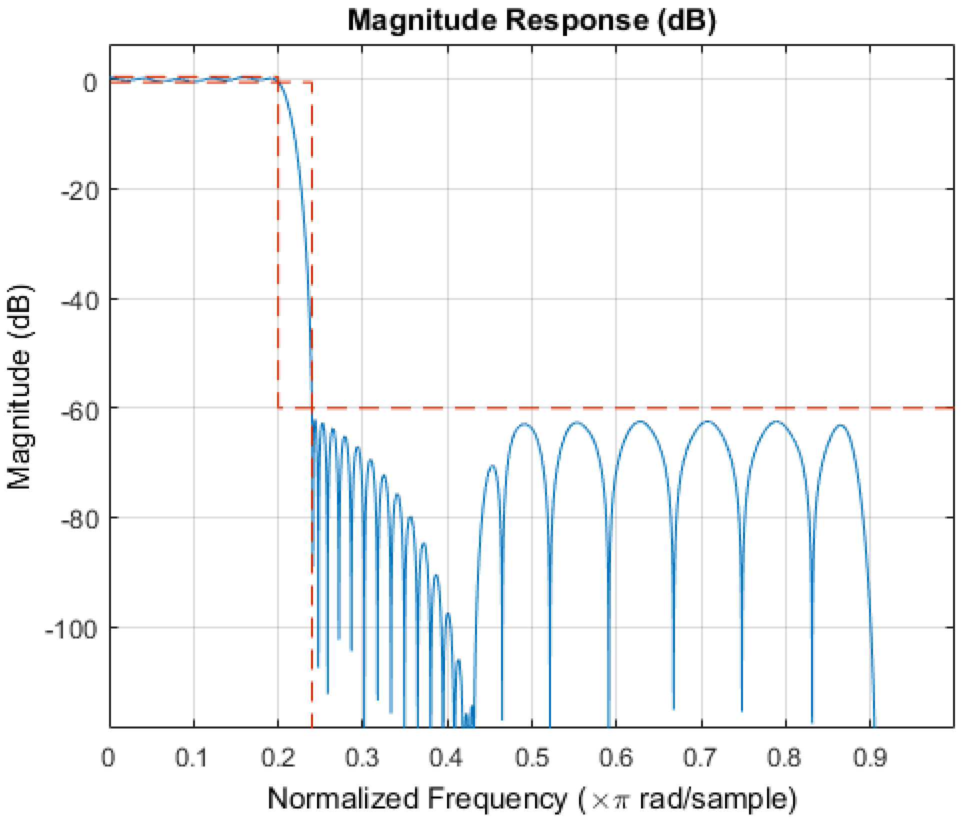
```
[h,g,d]=ifir(6,'high',[.12 .14],[.001 .01]);
H = dsp.FIRFilter('Numerator',h); G = dsp.FIRFilter('Numerator',g);
b1 = cascade(H,G);                % Branch 1
b2 = dsp.FIRFilter('Numerator',d); % Branch 2
Hoverall = freqz(b1) + freqz(b2); % Overall wideband highpass
plot(linspace(0,1,length(Hoverall)),20*log10(abs(Hoverall)))
xlabel('Normalized frequency (\times \pi) rad/s')
ylabel('Magnitude (dB)')
title('Overall Filter');
grid on
```



Design a cascade of lowpass filters

This example shows how to use `fdesign.lowpass` to design a cascade of lowpass filters. After designing the filter, use `fvtool` to plot the response curve.

```
fpass = 0.2;
fstop = 0.24;
d1 = fdesign.lowpass(fpass, fstop);
lowpassCascade = design(d1,'ifir','Systemobject',true);
fvtool(lowpassCascade)
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

See Also

Functions
`fdesign` | `fir1` | `firgr` | `firls` | `firpm`

Introduced in R2011a

iirbpc2bpc

Transform IIR complex bandpass filter to IIR complex bandpass filter with different characteristics

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirbpc2bpc(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirbpc2bpc(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the complex bandpass prototype by applying a first-order complex bandpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

This transformation effectively places two features of an original filter, located at frequencies W_{o1} and W_{o2}, at the required target frequency locations, W_{t1}, and W_{t2} respectively. It is assumed that W_{t2} is greater than W_{t1}. In most of the cases the features selected for the transformation are the band edges of the filter passbands. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

This transformation can also be used for transforming other types of filters; e.g., complex notch filters or resonators can be repositioned at two distinct desired frequencies at any place around the unit circle; e.g., in the adaptive system.

Examples

Shift Complex Bandpass

Design a prototype real IIR lowpass elliptic filter with a gain of about –3 dB at 0.5π rad/sample.

```
[b,a] = ellip(3,0.1,30,0.409);
```

Create a complex passband from 0.25π to 0.75π.

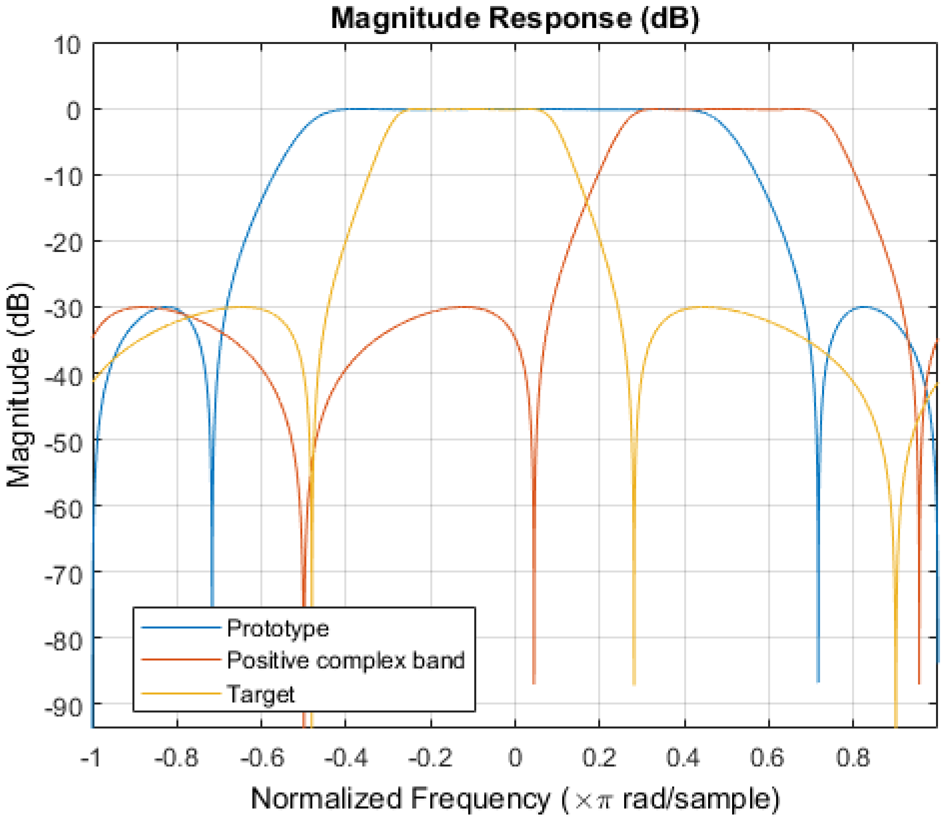
```
[bc,ac] = iirlp2bpc(b,a,0.5,[0.25 0.75]);
```

Move the bandpass to between –0.3π and 0.1π.

```
[num,den] = iirbpc2bpc(bc,ac,[0.25 0.75],[-0.3 0.1]);
```

Compare the three filters in FVTool.

```
hvft = fvtool(b,a,bc,ac,num,den);
legend(hvft,'Prototype','Positive complex band','Target')
```



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Num	Numerator of the target filter
Den	Denominator of the target filter

Variable	Description
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf` | `allpassbpc2bpc` | `zpkbpc2bpc`

Introduced in R2011a

iircomb

IIR comb notch or peak filter

Syntax

```
[num,den] = iircomb(n,bw)
[num,den] = iircomb(n,bw,ab)
[num,den] = iircomb(...,'type')
```

Description

`[num,den] = iircomb(n,bw)` returns a digital notching filter with order `n` and with the width of the filter notch at -3 dB set to `bw`, the filter bandwidth. The filter order must be a positive integer. `n` also defines the number of notches in the filter across the frequency range from 0 to 2π — the number of notches equals `n`+1.

For the notching filter, the transfer function takes the form

$$H(z) = b\frac{1-z^{-n}}{1-\alpha z^{-n}}$$

where a and b are the positive scalars and n is the filter order or the number of notches in the filter minus 1.

The quality factor (Q factor) q for the filter is related to the filter bandwidth by $q = \omega_0/bw$ where ω_0 is the frequency to remove from the signal.

`[num,den] = iircomb(n,bw,ab)` returns a digital notching filter whose bandwidth, `bw`, is specified at a level of -`ab` decibels. Including the optional input argument `ab` lets you specify the magnitude response bandwidth at a level that is not the default -3 dB point, such as -6 dB or 0 dB.

`[num,den] = iircomb(...,'type')` returns a digital filter of the specified type. The input argument `type` can be either

- `'notch'` to design an IIR notch filter. Notch filters attenuate the response at the specified frequencies. This is the default type. When you omit the `type` input argument, `iircomb` returns a notch filter.
- `'peak'` to design an IIR peaking filter. Peaking filters boost the signal at the specified frequencies.

The transfer function for peaking filters is

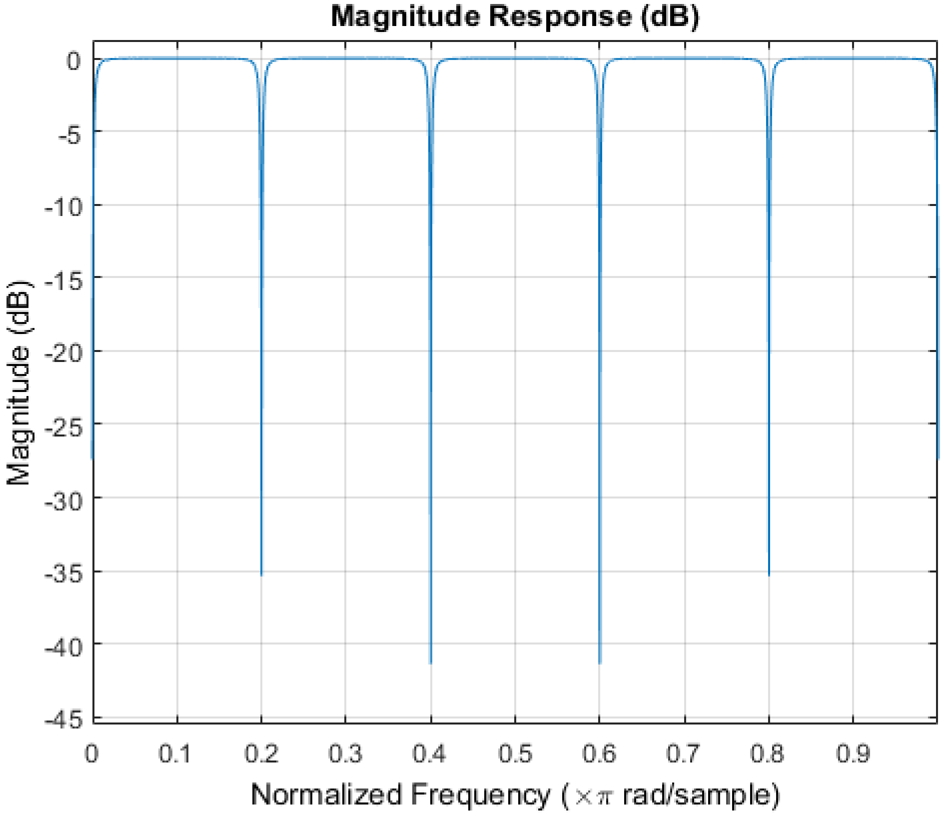
$$H(z) = b\frac{1 - z^{-n}}{1 + az^{-n}}$$

Examples

Design an IIR Notch Filter Using iircomb

Design and plot an IIR notch filter with 11 notches (equal to filter order plus 1) that removes a 60 Hz tone (f0) from a signal at 600 Hz(fs). For this example, set the Q factor for the filter to 35 and use it to specify the filter bandwidth.

```
fs = 600; fo = 60;  q = 35; bw = (fo/(fs/2))/q;
[b,a] = iircomb(fs/fo,bw,'notch'); % Note type flag 'notch'
fvtool(b,a);
```



Using the Filter Visualization Tool (FVTool) generates the following plot showing the filter notches. Note the notches are evenly spaced and one falls at exactly 60 Hz.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

`firgr` | `iirnotch` | `iirpeak` | `iirparameq`

Introduced in R2011a

iirftransf

IIR frequency transformation of filter

Syntax

```
[OutNum,OutDen] = iirftransf(OrigNum,OrigDen,FTFNum,FTFDen)
```

Description

`[OutNum,OutDen] = iirftransf(OrigNum,OrigDen,FTFNum,FTFDen)` returns the numerator and denominator vectors, `OutNum` and `OutDen`, of the target filter, which is the result of transforming the prototype filter specified by the numerator, `OrigNum`, and denominator, `OrigDen`, with the mapping filter given by the numerator, `FTFNum`, and the denominator, `FTFDen`. If the allpass mapping filter is not specified, then the function returns an original filter.

Examples

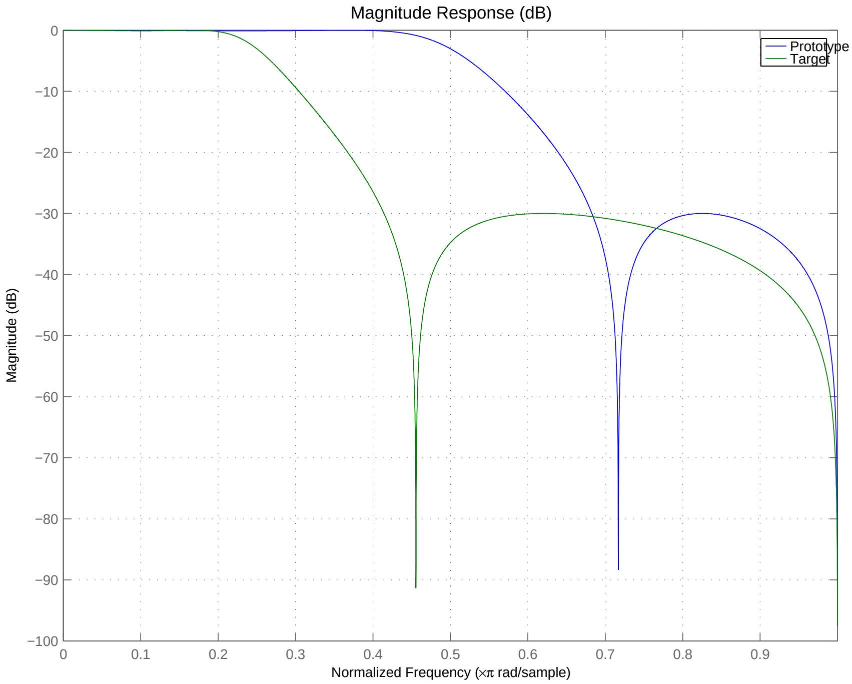
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
[AlpNum, AlpDen] = allpasslp2lp(0.5, 0.25);
[num, den] = iirftransf(b, a, AlpNum, AlpDen);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Here's the comparison between the filters.



Arguments

Variable	Description
OrigNum	Numerator of the prototype lowpass filter
OrigDen	Denominator of the prototype lowpass filter
FTFNum	Numerator of the mapping filter
FTFDen	Denominator of the mapping filter
OutNum	Numerator of the target filter
OutDen	Denominator of the target filter

See Also

zpkftransf

Introduced in R2011a

iirgrpdelay

Optimal IIR filter with prescribed group-delay

Syntax

```
[num,den] = iirgrpdelay(n,f,edges,a)
[num,den] = iirgrpdelay(n,f,edges,a,w)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden,tau)
[num,den,tau] = iirgrpdelay(n,f,edges,a,w)
```

Description

`[num,den] = iirgrpdelay(n,f,edges,a)` returns an allpass IIR filter of order `n` (`n` must be even) which is the best approximation to the relative group-delay response described by `f` and `a` in the least-pth sense. `f` is a vector of frequencies between 0 and 1 and `a` is specified in samples. The vector `edges` specifies the band-edge frequencies for multi-band designs. `iirgrpdelay` uses a constrained Newton-type algorithm. Always check your resulting filter using `grpdelay` or `freqz`.

`[num,den] = iirgrpdelay(n,f,edges,a,w)` uses the weights in `w` to weight the error. `w` has one entry per frequency point and must be the same length as `f` and `a`). Entries in `w` tell `iirgrpdelay` how much emphasis to put on minimizing the error in the vicinity of each specified frequency point relative to the other points.

`f` and `a` must have the same number of elements. `f` and `a` can contains more elements than the vector `edges` contains. This lets you use `f` and `a` to specify a filter that has any group-delay contour within each band.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius)` returns a filter having a maximum pole radius equal to `radius`, where 0<`radius`<1. `radius` defaults to 0.999999. Filters whose pole radius you constrain to be less than 1.0 can better retain transfer function accuracy after quantization.

[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p), where p is a two-element vector [pmin pmax], lets you determine the minimum and maximum values of p used in the least-pth algorithm. p defaults to [2 128] which yields filters very similar to the L-infinity, or Chebyshev, norm. pmin and pmax should be even. If p is 'inspect', no optimization occurs. You might use this feature to inspect the initial pole/zero placement.

[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens) specifies the grid density dens used in the optimization process. The number of grid points is (dens*(n+1)). The default is 20. dens can be specified as a single-element cell array. The grid is not equally spaced.

[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden) allows you to specify the initial estimate of the denominator coefficients in vector initden. This can be useful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox software can be used for generating initden.

[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden,tau) allows the initial estimate of the group delay offset to be specified by the value of tau, in samples.

[num,den,tau] = iirgrpdelay(n,f,edges,a,w) returns the resulting group delay offset. In all cases, the resulting filter has a group delay that approximates [a + tau]. Allpass filters can have only positive group delay and a non-zero value of tau accounts for any additional group delay that is needed to meet the shape of the contour specified by (f,a). The default for tau is max(a).

Hint: If the zeros or poles cluster together, your filter order may be too low or the pole radius may be too small (overly constrained). Try increasing n or radius.

For group-delay equalization of an IIR filter, compute a by subtracting the filter's group delay from its maximum group delay. For example,

```
[be,ae] = ellip(4,1,40,0.2);
f = 0:0.001:0.2;
g = grpdelay(be,ae,f,2);   % Equalize only the passband.
a = max(g)-g;
[num,den]=iirgrpdelay(8, f, [0 0.2], a);
```

References

Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All inputs must be constant. Expressions or variables are allowed if their values do not change.
- Does not support syntaxes that have cell array input.

See Also

`freqz` | `filter` | `grpdelay` | `zplane` | `iirlpnorm` | `iirlpnormc`

Introduced in R2011a

iirlinphase

Quasi-linear phase IIR filter from halfband filter specification

Syntax

```
iirlinFilter= design(d,'iirlinphase','SystemObject',true)
hd = design(...,'filterstructure',structure,'SystemObject',true)
```

Description

`iirlinFilter= design(d,'iirlinphase','SystemObject',true)` designs a quasi-linear phase filter `iirlinFilter` specified by the filter specification object `d`.

`hd = design(...,'`**`filterstructure`**`',structure,'SystemObject',true)` returns a filter with the structure specified by `structure`. By default, the filter structure is a cascade structure. The following table lists all the structures `design` supports for the IIR linear phase response.

Structure	Filter Structure
`cascadeallpass`	Cascade of allpass filters
`cascadewdfallpass`	Cascade of allpass wave digital filters
`iirdecim`	IIR polyphase decimator
`iirwdfdecim`	IIR wave digital filter polyphase decimator
`iirinterp`	IIR polyphase interpolator
`iirwdfinterp`	IIR wave digital filter polyphase interpolator

To get a list of all the structures the specific `fdesign` method supports, type the following in the MATLAB command prompt.

```
d = fdesign.halfband;
strucs = validstructures(d,'SystemObject',true);
```

To get a list of structures `iirlinphase` supports, type the following in the MATLAB command prompt.

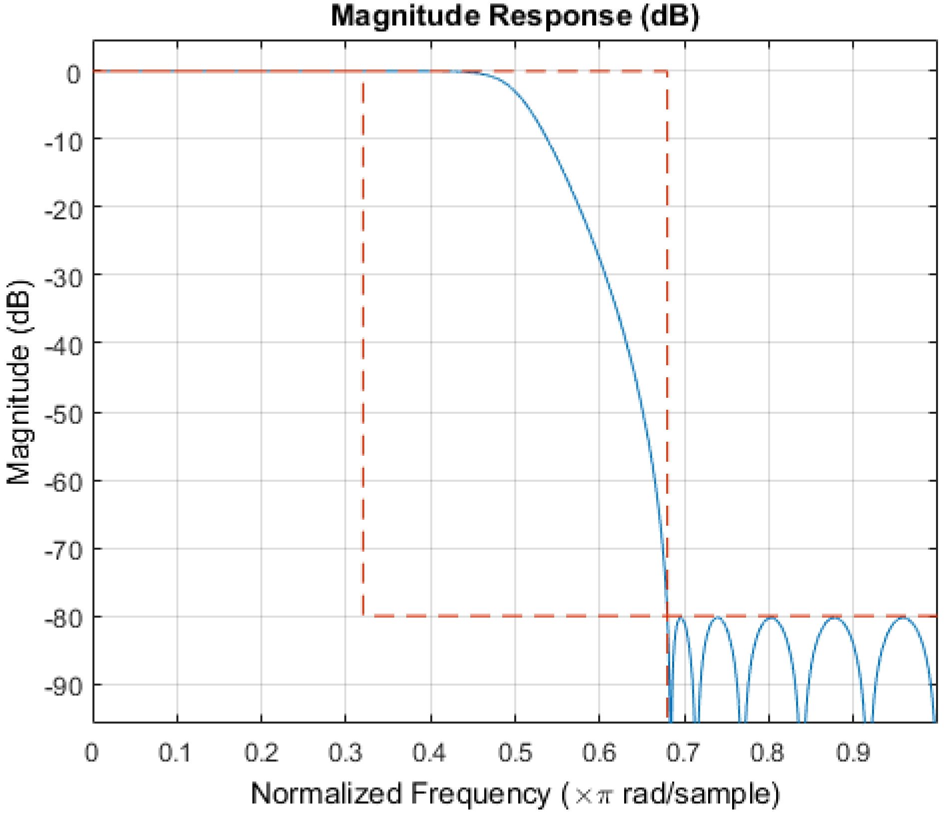
```
iirlinphaseStrucs = strucs.iirlinphase;
```

Examples

Design a Minimum-order Halfband IIR Filter

Design a quasi-linear phase, minimum-order halfband IIR filter with transition width of 0.36 and stopband attenuation of at least 80 dB.

```
tw = 0.36;
ast = 80;
d = fdesign.halfband('tw,ast',tw,ast); % Transition width,
                                       % stopband attenuation.
halfbandIIR = design(d,'iirlinphase','SystemObject',true);
fvtool(halfbandIIR)
```



Notice the characteristic halfband nature of the ripple in the stopband. If you measure the resulting filter, you see it meets the specifications.

```
measure(halfbandIIR)
```

```
ans =

Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.32
3-dB Point       : 0.5
6-dB Point       : 0.51911
Stopband Edge    : 0.68
Passband Ripple  : 4.0866e-08 dB
Stopband Atten.  : 80.2642 dB
Transition Width : 0.36
```

See Also

`fdesign.halfband`

Introduced in R2011a

iirlp2bp

Transform IIR lowpass filter to IIR bandpass filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2bp(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2bp(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandpass frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the “DC Mobility,” meaning that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_ts.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature: the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Real lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies.

Examples

Transform Lowpass Filter to Bandpass Filter

Design a prototype real IIR lowpass elliptic filter with a gain of about –3 dB at 0.5π rad/sample.

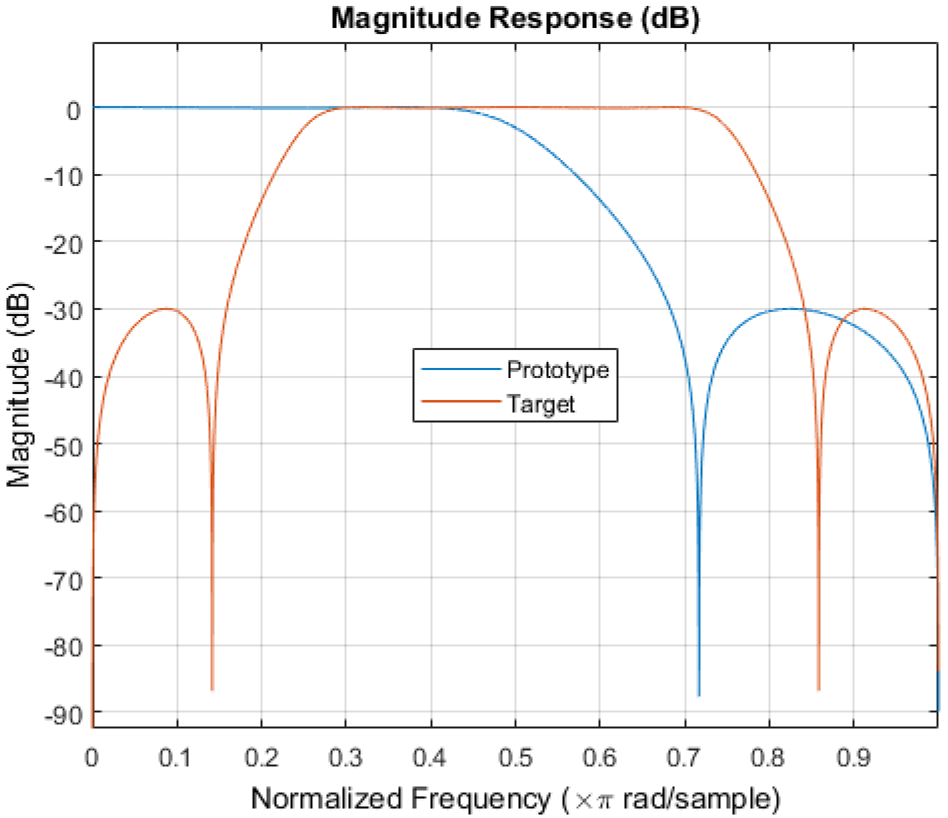
```
[b,a] = ellip(3,0.1,30,0.409);
```

Create a real bandpass filter by placing the cutoff frequencies of the prototype filter at 0.25π and 0.75π.

```
[num,den] = iirlp2bp(b,a,0.5,[0.25 0.75]);
```

Compare the magnitude responses of the filters using FVTool.

```
hvft = fvtool(b,a,num,den);
legend(hvft,'Prototype','Target')
```



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Num	Numerator of the target filter
Den	Denominator of the target filter

Variable	Description
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

[1] Constantinides, A.G., “Spectral transformations for digital filters,” *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

[2] Nowrouzian, B. and A.G. Constantinides, “Prototype reference transfer function parameters in the discrete-time frequency transformations,” *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

[3] Nowrouzian, B. and L.T. Bruton, “Closed-form solutions for discrete-time elliptic transfer functions,” *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

[4] Constantinides, A.G., “Design of bandpass digital filters,' *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

See Also

See Also

Functions

`allpasslp2bp` | `iirftransf` | `zpklp2bp`

Introduced in R2011a

iirlp2bpc

IIR lowpass to complex bandpass transformation

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2bpc(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2bpc(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency -W_o, at the required target frequency location, W_{t1}, and the second feature, originally at +W_o, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; for example real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandpass filters for radio receivers from the high-quality prototype lowpass filter.

Examples

Transform Lowpass Filter to Complex Bandpass Filter

Design a prototype real IIR lowpass elliptic filter with a gain of about –3 dB at 0.5π rad/sample.

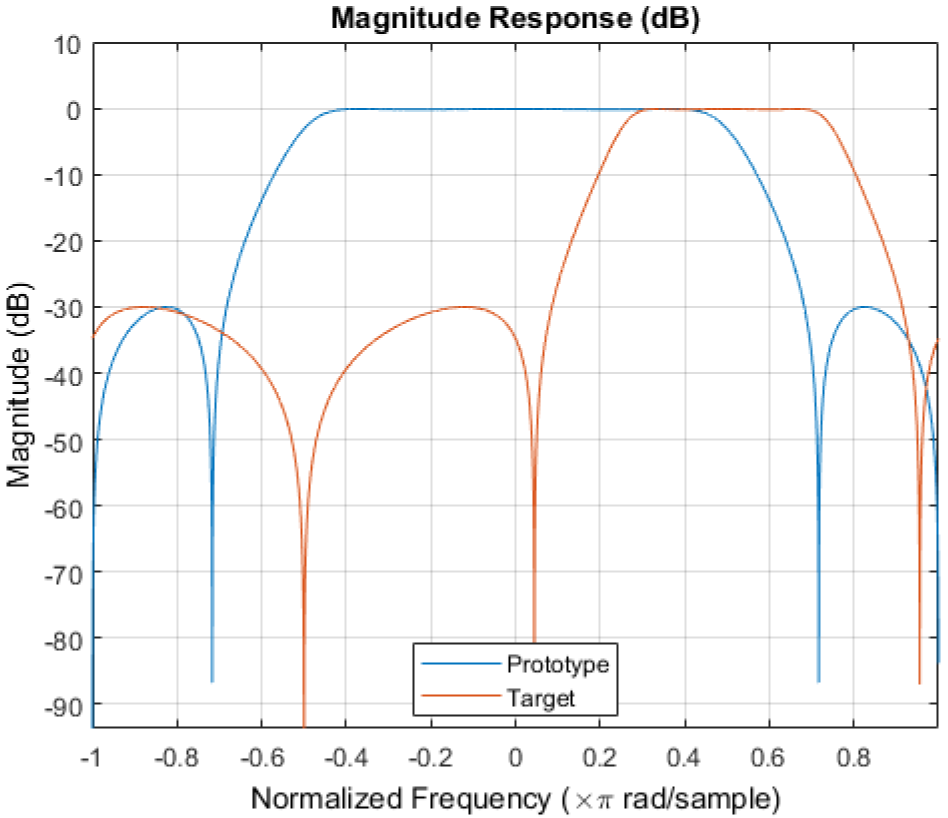
```
[b,a] = ellip(3,0.1,30,0.409);
```

Create a complex bandpass filter by placing the cutoff frequencies of the prototype filter at 0.25π and 0.75π.

```
[num,den] = iirlp2bpc(b,a,0.5,[0.25 0.75]);
```

Compare the magnitude responses of the filters using FVTool.

```
hvft = fvtool(b,a,num,den);
legend(hvft,'Prototype','Target')
```



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Variable	Description
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

See Also

Functions
allpasslp2bpc | iirftransf | zpklp2bpc

Introduced in R2011a

iirlp2bs

Transform IIR lowpass filter to IIR bandstop filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2bs(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2bs(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` of the bandstop digital filter. `AllpassNum` and `AllpassDen` are the vectors of numerator and denominator coefficients of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. It is assumed that W_{t2} is greater than W_{t1}. Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

This transformation implements the "Nyquist Mobility," which means that the DC feature stays at DC, but the Nyquist feature moves to a location dependent on the selection of W_o and W_ts.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

For more details on the lowpass to bandstop frequency transformation, see “Digital Frequency Transformations”.

Examples

Transform Lowpass Filter to Bandstop Filter

Design a prototype real IIR lowpass elliptic filter with a gain of about –3 dB at 0.5π rad/sample.

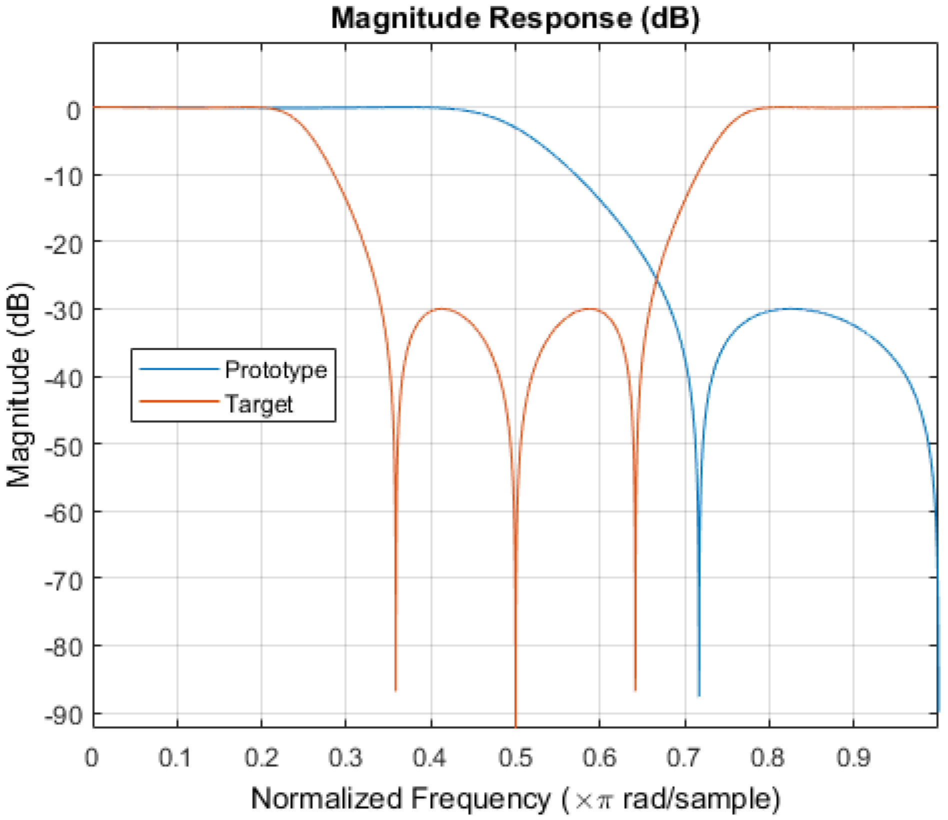
```
[b,a] = ellip(3,0.1,30,0.409);
```

Create a bandstop filter by placing the cutoff frequencies of the prototype filter at 0.25π and 0.75π.

```
[num,den] = iirlp2bs(b,a,0.5,[0.25 0.75]);
```

Compare the magnitude responses of the filters using FVTool.

```
fvt = fvtool(b,a,num,den);
legend(fvt,'Prototype','Target')
```



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Num	Numerator of the target filter
Den	Denominator of the target filter

Variable	Description
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

[1] Constantinides, A.G., "Spectral transformations for digital filters," *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

[2] Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

[3] Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

[4] Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

See Also

See Also

Functions

`allpasslp2bs` | `iirftransf` | `zpklp2bs`

Topics

"Digital Frequency Transformations"

Introduced in R2011a

iirlp2bsc

Transform IIR lowpass filter to IIR complex bandstop filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2bsc(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2bsc(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandstop frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and the denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency -W_o, at the required target frequency location, W_{t1}, and the second feature, originally at +W_o, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. Additionally the transformation swaps passbands with stopbands in the target filter.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandstop filters for band attenuation or frequency equalizers, from the high-quality prototype lowpass filter.

Examples

Transform Lowpass Filter to Complex Bandstop Filter

Design a prototype real IIR lowpass elliptic filter with a gain of about –3 dB at 0.5π rad/sample.

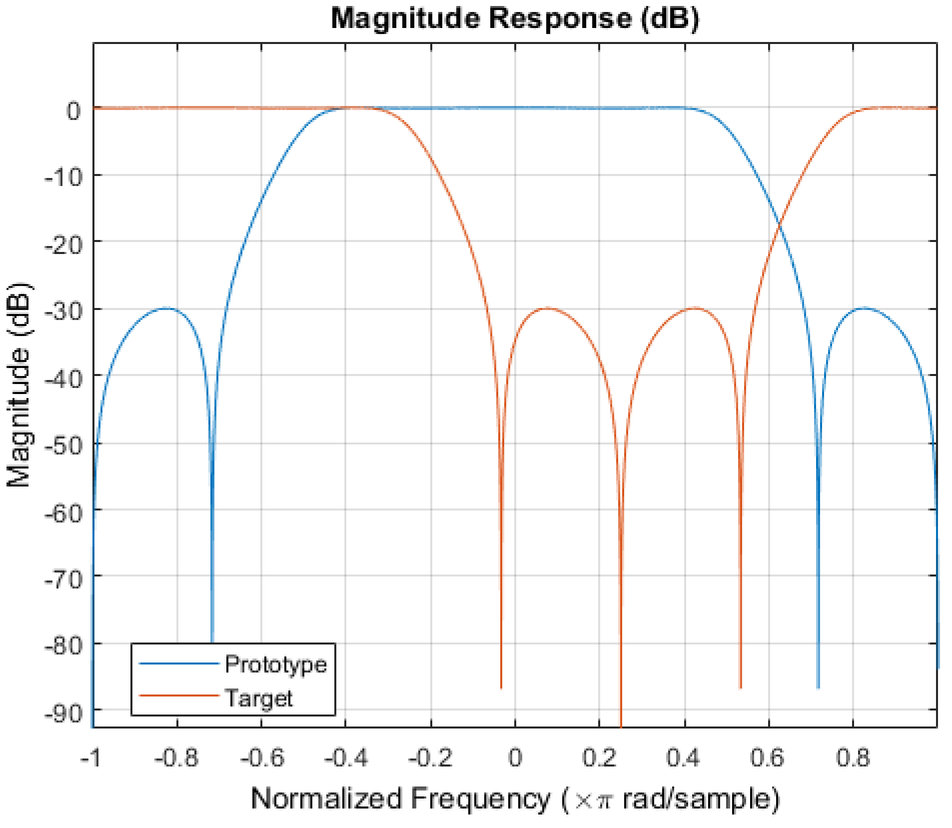
```
[b,a] = ellip(3,0.1,30,0.409);
```

Create a complex bandstop filter by placing the cutoff frequencies of the prototype filter at –0.25π and 0.75π.

```
[num,den] = iirlp2bsc(b,a,0.5,[-0.25 0.75]);
```

Compare the magnitude responses of the filters using FVTool.

```
fvt = fvtool(b,a,num,den);
legend(fvt,'Prototype','Target')
```



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Variable	Description
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

See Also

Functions
`allpasslp2bsc` | `iirftransf` | `zpklp2bsc`

Introduced in R2011a

iirlp2hp

Transform lowpass IIR filter to highpass filter

Syntax

```
[num,den] = iirlp2hp(b,a,wc,wd)
```

Description

`[num,den] = iirlp2hp(b,a,wc,wd)` with input arguments `b` and `a`, the numerator and denominator coefficients (zeros and poles) for a lowpass IIR filter, `iirlp2bp` transforms the magnitude response from lowpass to highpass. `num` and `den` return the coefficients for the transformed highpass filter. For `wc`, enter a selected frequency from your lowpass filter. You use the chosen frequency to define the magnitude response value you want in the highpass filter. Enter one frequency for the highpass filter — the value that defines the location of the transformed point — in `wd`. Note that all frequencies are normalized between zero and one. Notice also that the filter order does not change when you transform to a highpass filter.

When you select `wc` and designate `wd`, the transformation algorithm sets the magnitude response at the `wd` values of your bandstop filter to be the same as the magnitude response of your lowpass filter at `wc`. Filter performance between the values in `wd` is not specified, except that the stopband retains the ripple nature of your original lowpass filter and the magnitude response in the stopband is equal to the peak response of your lowpass filter. To accurately specify the filter magnitude response across the stopband of your bandpass filter, use a frequency value from within the stopband of your lowpass filter as `wc`. Then your bandstop filter response is the same magnitude and ripple as your lowpass filter stopband magnitude and ripple.

The fact that the transformation retains the shape of the original filter is what makes this function useful. If you have a lowpass filter whose characteristics, such as rolloff or passband ripple, particularly meet your needs, the transformation function lets you create a new filter with the same characteristic performance features, but in a highpass version. Without designing the highpass filter from the beginning.

In some cases transforming your filter may cause numerical problems, resulting in incorrect conversion to the highpass filter. Use `fvtool` to verify the response of your converted filter.

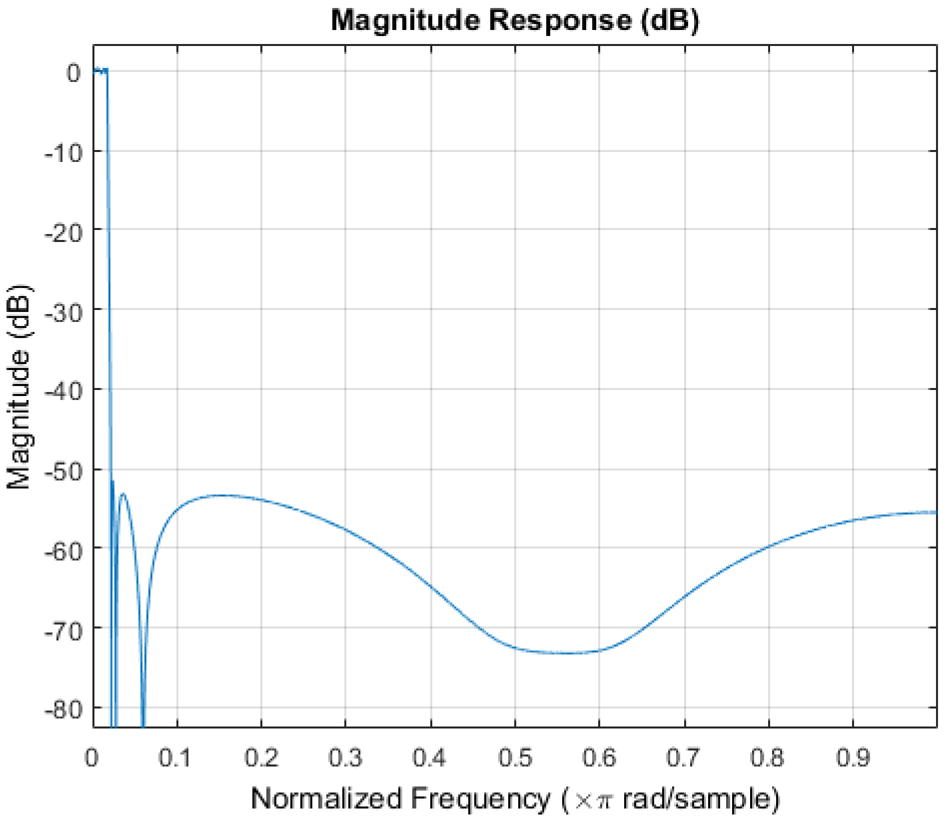
Examples

Transform Lowpass Filter to Highpass Filter

This example transforms an IIR filter from lowpass to highpass by moving the magnitude response at one frequency in the source filter to a new location in the transformed filter.

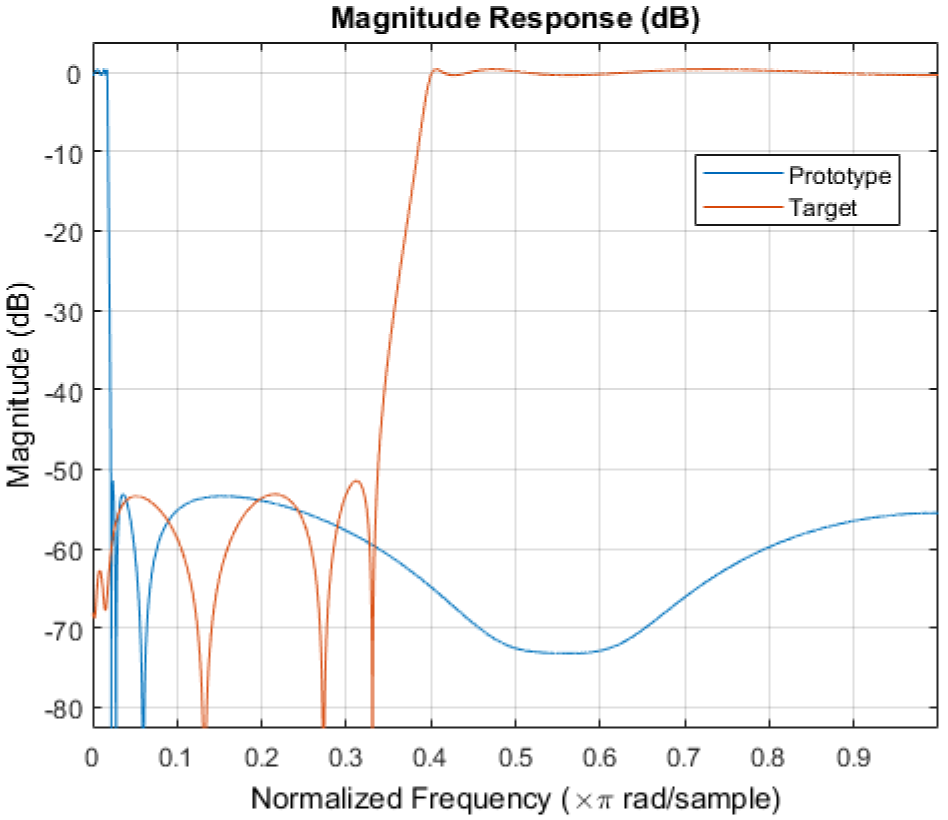
Generate a least P-norm optimal IIR lowpass filter with varying attenuation levels in the stopband. Specify a numerator order of 10 and a denominator order of 6. Visualize the magnitude response of the filter.

```
[b,a] = iirlpnorm(10,6,[0 0.0175 0.02 0.0215 0.025 1], ...
    [0 0.0175 0.02 0.0215 0.025 1],[1 1 0 0 0 0], ...
    [1 1 1 1 20 20]);

fvtool(b,a)
```



To generate a highpass filter whose passband flattens out at 0.4π rad/sample, select the frequency in the lowpass filter at 0.0175π, the frequency where the passband starts to roll off, and move it to the new location. Compare the magnitude responses of the filters using FVTool.

```
wc = 0.0175;
wd = 0.4;
[num,den] = iirlp2hp(b,a,wc,wd);

hvft = fvtool(b,a,num,den);
legend(hvft,'Prototype','Target')
```



The transition band for the highpass filter is essentially the mirror image of the transition for the lowpass filter from 0.0175π to 0.025π, stretched out over a wider frequency range. In the passbands, the filter share common ripple characteristics and magnitude.

References

[1] Mitra, Sanjit K., *Digital Signal Processing. A Computer-Based Approach*, Second Edition, McGraw-Hill, 2001.

See Also

See Also

Functions
`firlp2hp` | `firlp2lp` | `iirlp2bp` | `iirlp2bs` | `iirlp2lp`

Introduced in R2011a

iirlp2lp

Transform lowpass IIR filter to different lowpass filter

Syntax

```
[num,den] = iirlp2lp(b,a,wc,wd)
```

Description

`[num,den] = iirlp2lp(b,a,wc,wd)` with input arguments `b` and `a`, the numerator and denominator coefficients (zeros and poles) for a lowpass IIR filter, `iirlp2lp` transforms the magnitude response from lowpass to highpass. `num` and `den` return the coefficients for the transformed highpass filter. For `wc`, enter a selected frequency from your lowpass filter. You use the chosen frequency to define the magnitude response value you want in the highpass filter. Enter one frequency for the highpass filter — the value that defines the location of the transformed point — in `wd`. Note that all frequencies are normalized between zero and one. Notice also that the filter order does not change when you transform to a highpass filter.

When you select `wc` and designate `wd`, the transformation algorithm sets the magnitude response at the `wd` values of your bandstop filter to be the same as the magnitude response of your lowpass filter at `wc`. Filter performance between the values in `wd` is not specified, except that the stopband retains the ripple nature of your original lowpass filter and the magnitude response in the stopband is equal to the peak response of your lowpass filter. To accurately specify the filter magnitude response across the stopband of your bandpass filter, use a frequency value from within the stopband of your lowpass filter as `wc`. Then your bandstop filter response is the same magnitude and ripple as your lowpass filter stopband magnitude and ripple.

The fact that the transformation retains the shape of the original filter is what makes this function useful. If you have a lowpass filter whose characteristics, such as rolloff or passband ripple, particularly meet your needs, the transformation function lets you create a new filter with the same characteristic performance features, but in a highpass version. Without designing the highpass filter from the beginning.

In some cases transforming your filter may cause numerical problems, resulting in incorrect conversion to the highpass filter. Use `fvtool` to verify the response of your converted filter.

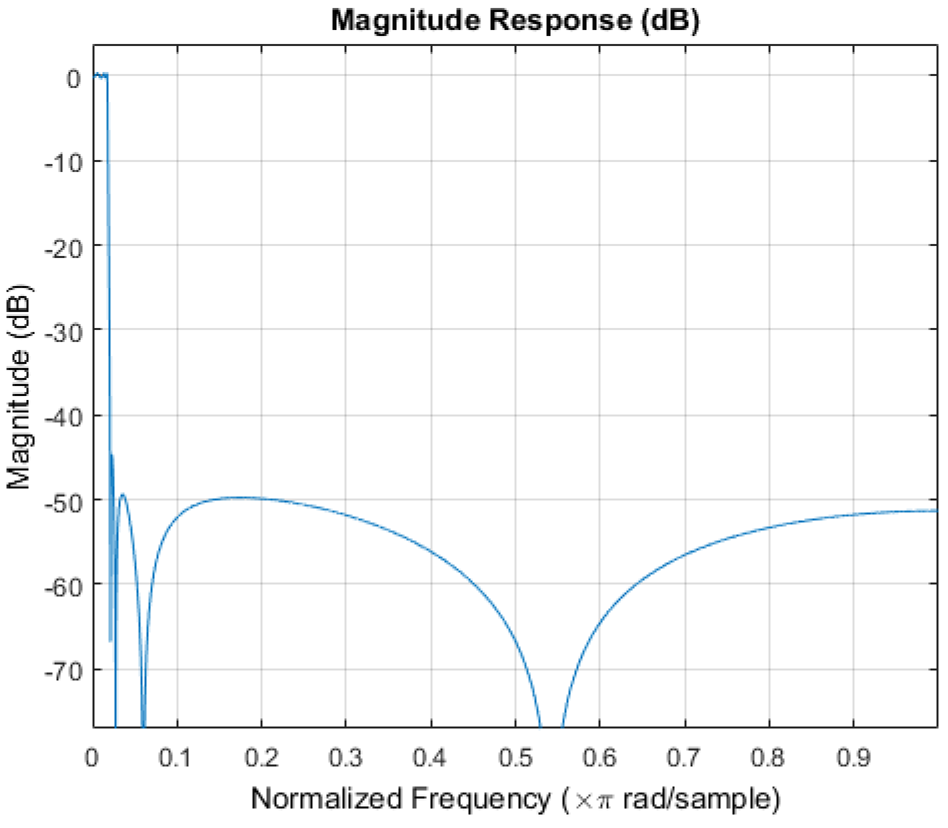
Examples

Extend Passband of Lowpass Filter

This example transforms the passband of a lowpass IIR filter by moving the magnitude response at one frequency in the source filter to a new location in the transformed filter.

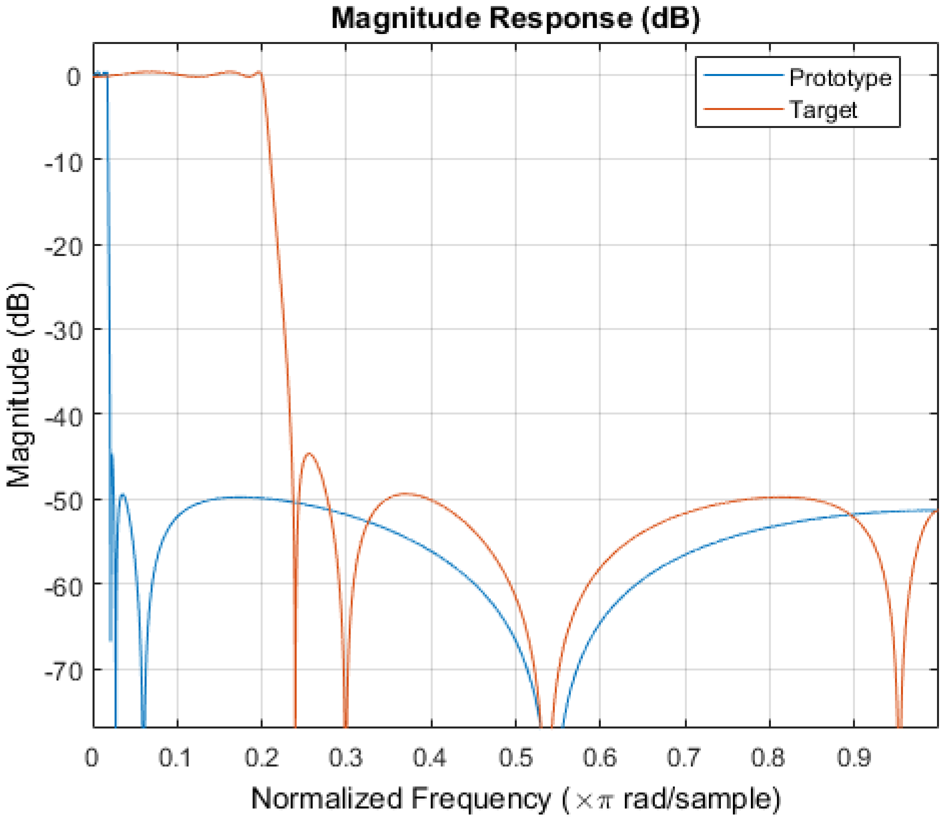
Generate a least P-norm optimal IIR lowpass filter with varying attenuation levels in the stopband. Specify a numerator order of 10 and a denominator order of 6. Visualize the magnitude response of the filter.

```
[b,a] = iirlpnorm(10,6,[0 0.0175 0.02 0.0215 0.025 1], ...
    [0 0.0175 0.02 0.0215 0.025 1],[1 1 0 0 0 0], ...
    [1 1 1 1 10 10]);

fvtool(b,a)
```



To generate a lowpass filter whose passband extends out to 0.2π rad/sample, select the frequency in the lowpass filter at 0.0175π, the frequency where the passband starts to roll off, and move it to the new location. Compare the magnitude responses of the filters using FVTool.

```
wc = 0.0175;
wd = 0.2;
[num,den] = iirlp2lp(b,a,wc,wd);

hvft = fvtool(b,a,num,den);
legend(hvft,'Prototype','Target')
```



Moving the edge of the passband from π to 0.2π results in a new lowpass filter whose peak response in-band is the same as in the original filter, with the same ripple and the same absolute magnitude. The rolloff is slightly less steep and the stopband profiles are the same for both filters. The new filter stopband is a "stretched" version of the original, as is the passband of the new filter.

References

[1] Mitra, Sanjit K, *Digital Signal Processing. A Computer-Based Approach*, Second Edition, McGraw-Hill, 2001.

See Also

See Also

Functions

`firlp2hp` | `firlp2lp` | `iirlp2bp` | `iirlp2bs` | `iirlp2hp`

Introduced in R2011a

iirlp2mb

Transform IIR lowpass filter to IIR M-band filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2mb(B,A,Wo,Wt)
[Num,Den,AllpassNum,AllpassDen]=iirlp2mb(B,A,Wo,Wt,Pass)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2mb(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying an Mth-order real lowpass to real multiple bandpass frequency mapping. By default the DC feature is kept at its original location.

`[Num,Den,AllpassNum,AllpassDen]=iirlp2mb(B,A,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility." In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is movable.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, W_{t1},...,W_{tM}.

Relative positions of other features of an original filter do not change in the target filter. It is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Transform Lowpass Filter to Multiband Filters

Design a prototype real IIR lowpass elliptic filter with a gain of about –3 dB at 0.5π rad/sample.

```
[b,a] = ellip(3,0.1,30,0.409);
```

Create a real multiband filter with two passbands.

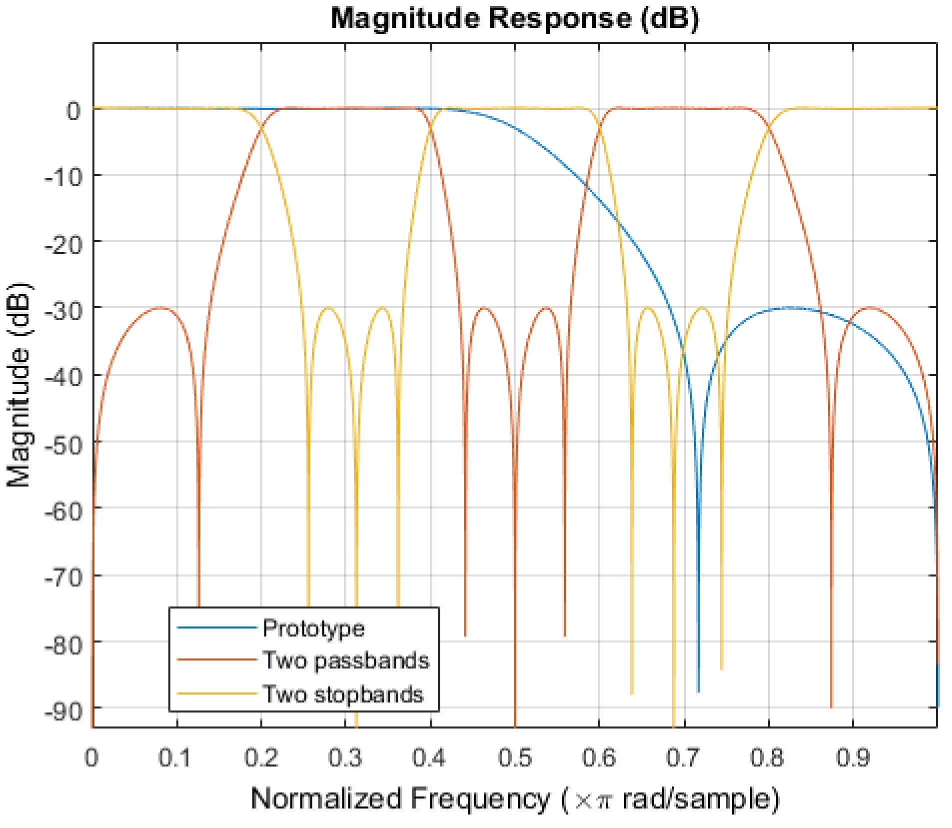
```
[num1,den1] = iirlp2mb(b,a,0.5,[2 4 6 8]/10);
```

Create a real multiband filter with two stopbands.

```
[num2,den2] = iirlp2mb(b,a,0.5,[2 4 6 8]/10, 'stop');
```

Compare the magnitude responses of the filters using FVTool.

```
hvft = fvtool(b,a,num1,den1,num2,den2);
legend(hvft,'Prototype','Two passbands','Two stopbands')
```



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Pass	Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `'pass'` being the default

Variable	Description
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

[1] Franchitti, J.C., "All-pass filter interpolation and frequency transformation problems," *MSc Thesis*, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.

[2] Feyh, G., J.C. Franchitti and C.T. Mullis, "All-pass filter interpolation and frequency transformation problem," *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

[3] Mullis, C.T. and R. A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Mass., Addison-Wesley, 1987.

[4] Feyh, G., W.B. Jones and C.T. Mullis, "An extension of the Schur Algorithm for frequency transformations," *Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

See Also

See Also

Functions

`allpasslp2mb` | `iirftransf` | `zpklp2mb`

Introduced in R2011a

iirlp2mbc

Transform IIR lowpass filter to IIR complex M-band filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2mbc(B,A,Wo,Wc)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2mbc(B,A,Wo,Wc)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying an `M`th-order real lowpass to complex multibandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, W_{t1},...,W_{tM}.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Transform Lowpass Filter to Complex Multiband Filter

Design a prototype real IIR lowpass elliptic filter with a gain of about –3 dB at 0.5π rad/sample.

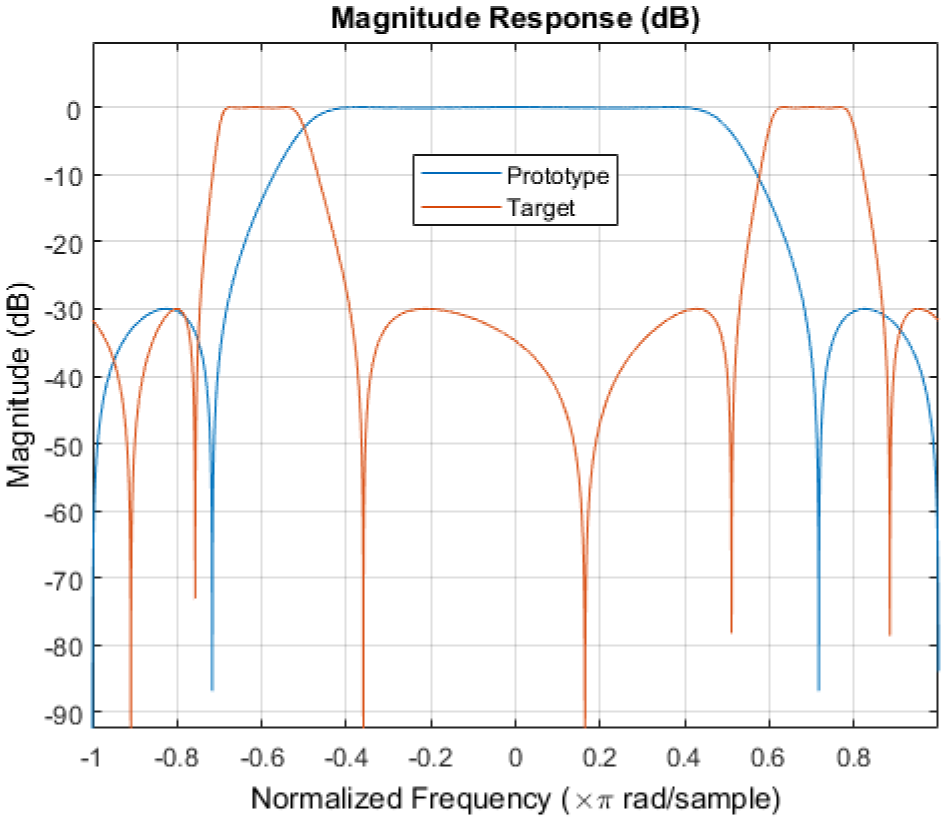
```
[b,a] = ellip(3,0.1,30,0.409);
```

Create a complex multiband filter with two passbands.

```
[num,den] = iirlp2mbc(b,a,0.5,[-7 -5 6 8]/10);
```

Compare the magnitude responses of the filters using FVTool. `iirlp2mbc` replicates the desired feature at 0.5 in the lowpass filter at four locations in the multiband filter.

```
hvft = fvtool(b,a,num,den);
legend(hvft,'Prototype','Target')
```



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter.
A	Denominator of the prototype lowpass filter.
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Variable	Description
Wc	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Num	Numerator of the target filter.
Den	Denominator of the target filter.
AllpassNum	Numerator of the mapping filter.
AllpassDen	Denominator of the mapping filter.

See Also

See Also

Functions
allpasslp2mbc | iirftransf | zpklp2mbc

Introduced in R2011a

iirlp2xc

Transform IIR lowpass filter to IIR complex N-point filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2xc(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2xc(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying an `N`th-order real lowpass to complex multipoint frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies $W_{o1},...,W_{oN}$, at the required target frequency locations, $W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., a stopband edge, DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Transform Lowpass Filter to IIR Complex N-Point Filter

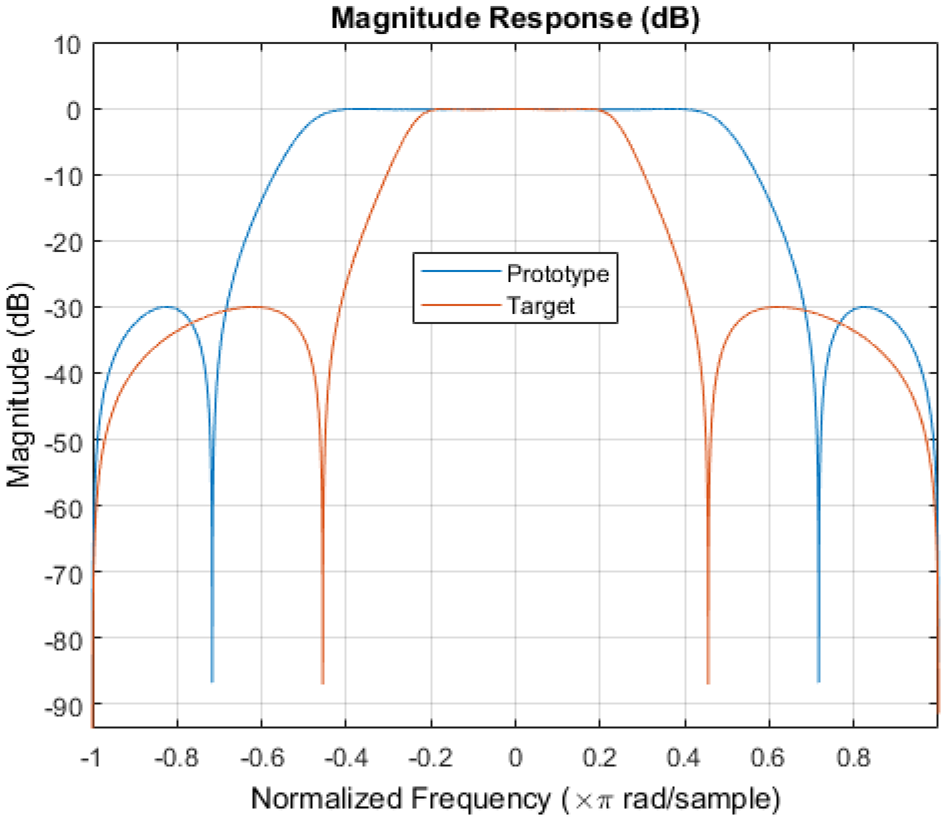
Design a prototype real IIR lowpass elliptic filter with a gain of about –3 dB at 0.5π rad/sample.

```
[b,a] = ellip(3,0.1,30,0.409);
```

Transform the lowpass filter to an IIR complex N-point filter. Compare the magnitude responses of the filters using FVTool.

```
[num,den] = iirlp2xc(b,a,[-0.5 0.5],[-0.25 0.25])

fvt = fvtool(b,a,num,den);
legend(fvt,'Prototype','Target')
```

```
num =

   0.0643 - 0.0000i   0.0464 + 0.0000i   0.0464 + 0.0000i   0.0643 + 0.0000i


den =

   1.0000 + 0.0000i  -1.6918 - 0.0000i   1.2340 + 0.0000i  -0.3207 - 0.0000i
```



The target filter has complex coefficients and is indeed a bandpass filter.

Arguments

Variable	Description
B	Numerator of the prototype lowpass filter.
A	Denominator of the prototype lowpass filter.
Wo	Frequency values to be transformed from the prototype filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

Variable	Description
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Num	Numerator of the target filter.
Den	Denominator of the target filter.
AllpassNum	Numerator of the mapping filter.
AllpassDen	Denominator of the mapping filter.

See Also

See Also

Functions
allpasslp2xc | iirftransf | zpklp2xc

Introduced in R2011a

iirlp2xn

Transform IIR lowpass filter to IIR real N-point filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2xn(B,A,Wo,Wt)
[Num,Den,AllpassNum,AllpassDen]= iirlp2xn(B,A,Wo,Wt,Pass)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2xn(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying an `N`th-order real lowpass to real multipoint frequency transformation, where `N` is the number of features being mapped. By default the DC feature is kept at its original location.

`[Num,Den,AllpassNum,AllpassDen]= iirlp2xn(B,A,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility." In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies $W_{o1},...,W_{oN}$, at the required target frequency locations, $W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

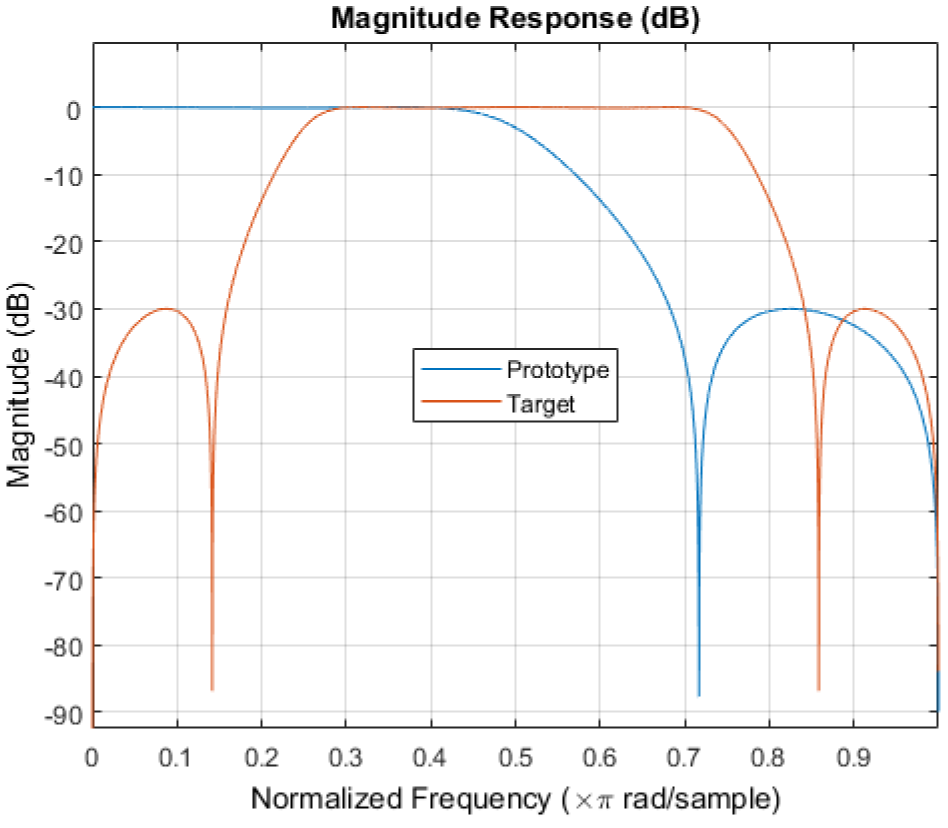
Transform Lowpass Filter to IIR Real N-Point Filter

Design a prototype real IIR lowpass elliptic filter with a gain of about –3 dB at 0.5π rad/sample.

```
[b,a] = ellip(3,0.1,30,0.409);
```

Transform the lowpass filter to an IIR real N-point filter. Compare the magnitude responses of the filters using FVTool.

```
[num,den] = iirlp2xn(b,a,[-0.5 0.5],[0.25 0.75]);

hvft = fvtool(b,a,num,den);
legend(hvft,'Prototype','Target')
```



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Pass	Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `'pass'` being the default

Variable	Description
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

[1] Cain, G.D., A. Krukowski and I. Kale, “High Order Transformations for Flexible IIR Filter Design,” *VII European Signal Processing Conference (EUSIPCO'94)*, vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

[2] Krukowski, A., G.D. Cain and I. Kale, “Custom designed high-order frequency transformations for IIR filters,” *38th Midwest Symposium on Circuits and Systems (MWSCAS'95)*, Rio de Janeiro, Brazil, August 1995.

See Also

See Also

Functions
`allpasslp2xn` | `iirftransf` | `zpklp2xn`

Introduced in R2011a

iirlpnorm

Least P-norm optimal IIR filter

Syntax

```
[num,den] = iirlpnorm(n,d,f,edges,a)
[num,den] = iirlpnorm(n,d,f,edges,a,w)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens,initnum,initden)
[num,den,err] = iirlpnorm(...)
[num,den,err,sos,g] = iirlpnorm(...)
```

Description

`[num,den] = iirlpnorm(n,d,f,edges,a)` returns a filter having a numerator order `n` and denominator order `d` which is the best approximation to the desired frequency response described by `f` and `a` in the least-pth sense. The vector `edges` specifies the band-edge frequencies for multi-band designs. An unconstrained quasi-Newton algorithm is employed and any poles or zeros that lie outside of the unit circle are reflected back inside. `n` and `d` should be chosen so that the zeros and poles are used effectively. See the "Hints" on page 5-983 section. Always use `freqz` to check the resulting filter.

`[num,den] = iirlpnorm(n,d,f,edges,a,w)` uses the weights in `w` to weight the error. `w` has one entry per frequency point (the same length as `f` and `a`) which tells `iirlpnorm` how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points. `f` and `a` must have the same number of elements, which may exceed the number of elements in `edges`. This allows for the specification of filters having any gain contour within each band. The frequencies specified in `edges` must also appear in the vector `f`. For example,

```
[num,den] = iirlpnorm(5,12,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])
```

is a lowpass filter with a peak of 1.6 within the passband.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p)` where `p` is a two-element vector `[pmin pmax]` allows for the specification of the minimum and maximum values of `p` used in the least-pth algorithm. Default is [2 128] which essentially yields the L-infinity, or Chebyshev, norm. `Pmin` and `pmax` should be even. If `p` is `'inspect'`, no optimization will occur. This can be used to inspect the initial pole/zero placement.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is `(dens*(n+d+1))`. The default is 20. `dens` can be specified as a single-element cell array. The grid is not equally spaced.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens,initnum,initden)` allows for the specification of the initial estimate of the filter numerator and denominator coefficients in vectors `initnum` and `initden`. `initnum` should be of length n+1, and `initden` should be of length d+1. This may be useful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox software can be used for generating `initnum` and `initden`.

`[num,den,err] = iirlpnorm(...)` returns the least-pth approximation error, `err`.

`[num,den,err,sos,g] = iirlpnorm(...)` returns the second-order section representation in the matrix `sos` and gain `g`. For numerical reasons it may be beneficial to use `sos` and `g` in some cases.

Examples

"Least Pth-Norm Optimal IIR Filter Design"

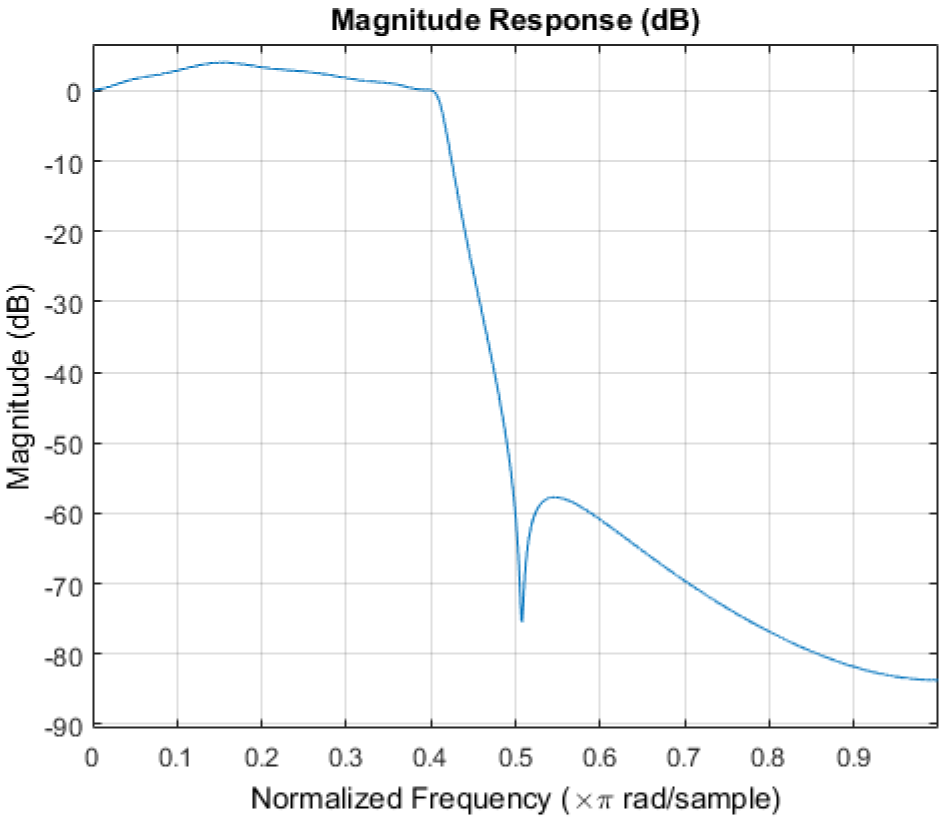
Design a Lowpass Filter with a Peak of 4.015 dB in Passband

```
[num,den] = iirlpnorm(5,12,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])


num =

   -0.0128    0.0041   -0.0068    0.0048    0.0056   -0.0001


den =
```

```
  Columns 1 through 7

    1.0000   -6.0264   18.6845  -38.7635   59.2365  -69.7345   64.5556

  Columns 8 through 13

  -47.2403   27.1100  -11.9198    3.8290   -0.8143    0.0882
```

Display the magnitude response in fvtool

```
fvtool(num,den)
```



Hints

- This is a weighted least-pth optimization.
- Check the radii and locations of the poles and zeros for your filter. If the zeros are on the unit circle and the poles are well inside the unit circle, try increasing the order of the numerator or reducing the error weighting in the stopband.
- Similarly, if several poles have a large radii and the zeros are well inside of the unit circle, try increasing the order of the denominator or reducing the error weighting in the passband.

References

Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All inputs must be constant. Expressions or variables are allowed if their values do not change.
- Does not support syntaxes that have cell array input.

See Also

`iirlpnormc` | `zplane` | `filter` | `freqz` | `iirgrpdelay`

Introduced in R2011a

iirlpnormc

Constrained least Pth-norm optimal IIR filter

Syntax

```
[num,den] = iirlpnormc(n,d,f,edges,a)
[num,den] = iirlpnormc(n,d,f,edges,a,w)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p)
[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens)
[num,den] =
iirlpnormc(n,d,f,edges,a,w,radius,p,dens,initnum,initden)
[num,den,err] = iirlpnormc(...)
[num,den,err,sos,g] = iirlpnormc(...)
```

Description

`[num,den] = iirlpnormc(n,d,f,edges,a)` returns a filter having numerator order `n` and denominator order `d` which is the best approximation to the desired frequency response described by `f` and `a` in the least-pth sense. The vector `edges` specifies the band-edge frequencies for multi-band designs. A constrained Newton-type algorithm is employed. `n` and `d` should be chosen so that the zeros and poles are used effectively. See the Hints section. Always check the resulting filter using `fvtool`.

`[num,den] = iirlpnormc(n,d,f,edges,a,w)` uses the weights in `w` to weight the error. `w` has one entry per frequency point (the same length as `f` and `a`) which tells `iirlpnormc` how much emphasis to put on minimizing the error in the vicinity of each frequency point relative to the other points. `f` and `a` must have the same number of elements, which can exceed the number of elements in `edges`. This allows for the specification of filters having any gain contour within each band. The frequencies specified in `edges` must also appear in the vector `f`. For example,

```
[num,den] = iirlpnormc(5,5,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])
```

designs a lowpass filter with a peak of 1.6 within the passband.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius)` returns a filter having a maximum pole radius of `radius` where 0<`radius`<1. `radius` defaults to 0.999999. Filters that have a reduced pole radius may retain better transfer function accuracy after you quantize them.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p)` where `p` is a two-element vector [`pmin pmax`] allows for the specification of the minimum and maximum values of `p` used in the least-pth algorithm. Default is [2 128] which essentially yields the L-infinity, or Chebyshev, norm. `pmin` and `pmax` should be even. If `p` is `'inspect'`, no optimization will occur. This can be used to inspect the initial pole/zero placement.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is `(dens*(n+d+1))`. The default is 20. `dens` can be specified as a single-element cell array. The grid is not equally spaced.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens,initnum,initden)` allows for the specification of the initial estimate of the filter numerator and denominator coefficients in vectors `initnum` and `initden`. This may be useful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox software can be used for generating `initnum` and `initden`.

`[num,den,err] = iirlpnormc(...)` returns the least-Pth approximation error `err`.

`[num,den,err,sos,g] = iirlpnormc(...)` returns the second-order section representation in the matrix SOS and gain G. For numerical reasons you may find SOS and G beneficial in some cases.

Hints

- This is a weighted least-pth optimization.
- Check the radii and location of the resulting poles and zeros.
- If the zeros are all on the unit circle and the poles are well inside of the unit circle, try increasing the order of the numerator or reducing the error weighting in the stopband.
- Similarly, if several poles have a large radius and the zeros are well inside of the unit circle, try increasing the order of the denominator or reducing the error weight in the passband.

- If you reduce the pole radius, you might need to increase the order of the denominator.

The message

```
Poorly conditioned matrix. See the "help" file.
```

indicates that `iirlpnormc` cannot accurately compute the optimization because either:

a The approximation error is extremely small (try reducing the number of poles or zeros — refer to the hints above).

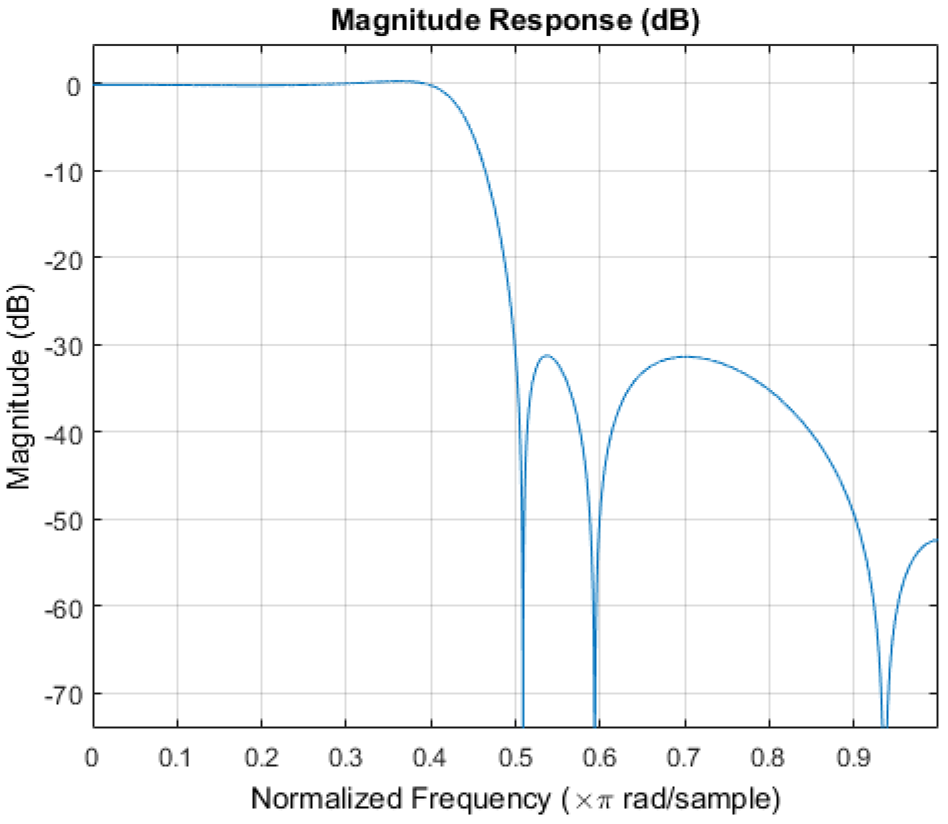
b The filter specifications have huge variation, such as `a=[1 1e9 0 0]`.

Examples

Magnitude Response of Constrained Least Pth-norm Optimal IIR Filter

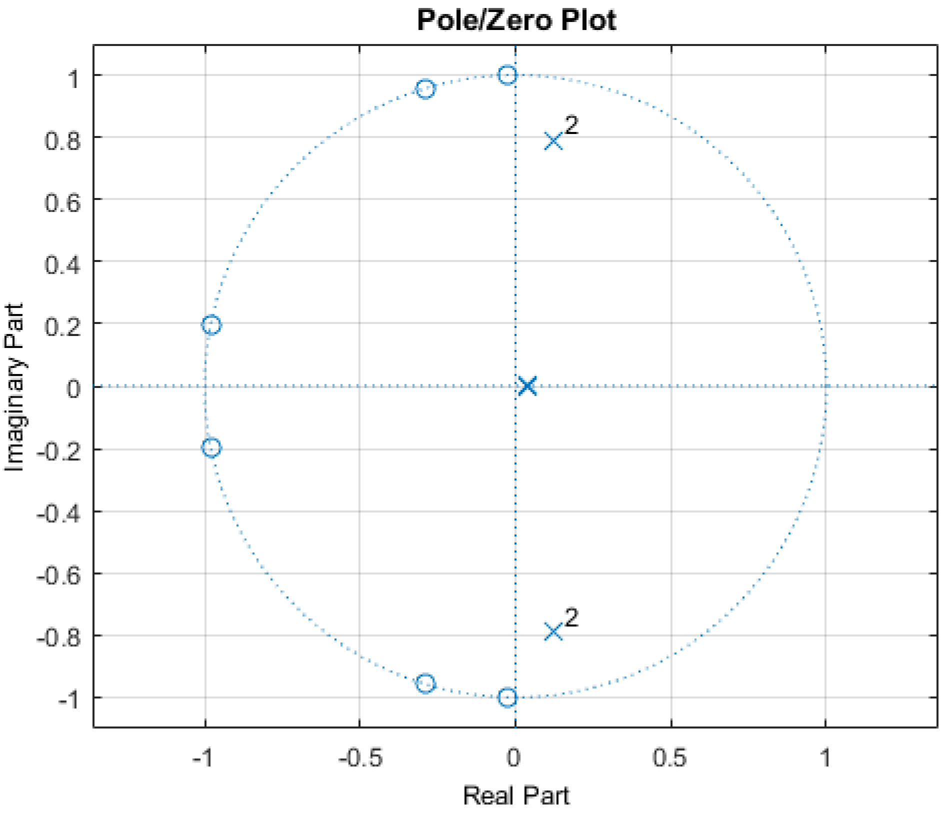
This example returns a lowpass filter whose pole radius is constrained to 0.8.

```
[b,a,err,s,g] = iirlpnormc(6,6,[0 .4 .5 1],[0 .4 .5 1],...
[1 1 0 0],[1 1 1 1],.8);
fvtool(b,a);
```



The magnitude response shows the lowpass nature of the filter. The pole/zero plot following shows that the poles are constrained to 0.8 as specified in the command.

```
fvtool(b,a,'polezero');
```



References

[1] Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All inputs must be constant. Expressions or variables are allowed if their values do not change.
- Does not support syntaxes that have cell array input.

See Also

See Also

Functions
`filter` | `freqz` | `iirgrpdelay` | `iirlpnorm` | `zplane`

Introduced in R2011a

iirls

Least-squares IIR filter from specification object

Syntax

```
hd = design(d,'iirls','SystemObject',true)
hd =
design(d,'iirls',designoption,value,designoption,value,...,'SystemObject',true
```

Description

`hd = design(d,'iirls','SystemObject',true)` designs a least-squares filter specified by the filter specification object `d`.

Note The `iirls` algorithm might not be well behaved in all cases. Experience is your best guide to determining if the resulting filter meets your needs. When you use `iirls` to design a filter, review the filter carefully to ensure that it is appropriate for your use.

`hd =`
`design(d,'iirls',designoption,value,designoption,value,...,'SystemObject',true`
returns a least-squares IIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `iirls`, refer to the command line help system. For example, to get specific information about using `iirls` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'iirls')
```

Examples

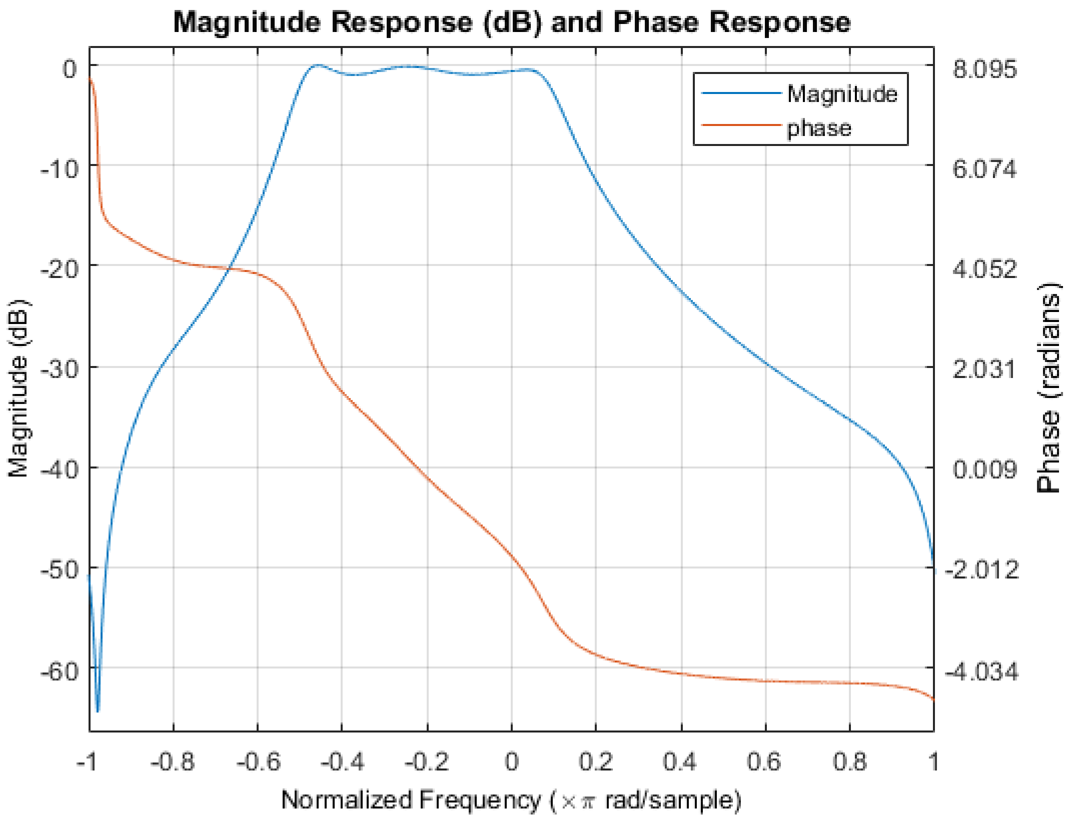
Design a Complex Bandpass Filter

Starting from an arbitrary magnitude and phase design object d, generate a complex bandpass filter of order = 5. To make the example a little easier to do, use the default values for F, and H, the frequency vector and the complex desired frequency response.

```
d = fdesign.arbmagnphase('N,F,H',5)
```

```
d = 

  arbmagnphase with properties:

               Response: 'Arbitrary Magnitude and Phase'
          Specification: 'N,F,H'
            Description: {3×1 cell}
    NormalizedFrequency: 1
            FilterOrder: 5
            Frequencies: [1×655 double]
           FreqResponse: [1×655 double]
```

```
fvt = design(d,'iirls','SystemObject',true);
fvtool(fvt,'freq');
legend('Magnitude','phase');
```



See Also

`fdesign.arbmag` | `firls` | `fdesign.arbmagnphase`

Introduced in R2011a

iirnotch

Second-order IIR notch filter

Syntax

```
[num,den] = iirnotch(w0,bw)
[num,den] = iirnotch(w0,bw,ab)
```

Description

`[num,den] = iirnotch(w0,bw)` turns a digital notching filter with the notch located at `w0`, and with the bandwidth at the -3 dB point set to `bw`. To design the filter, `w0` must meet the condition 0.0 `w0` 1.0, where 1.0 corresponds to π radians per sample in the frequency range.

The quality factor (Q factor) `q` for the filter is related to the filter bandwidth by `q` `w0/bw` where ω_0 is `w0`, the frequency to remove from the signal.

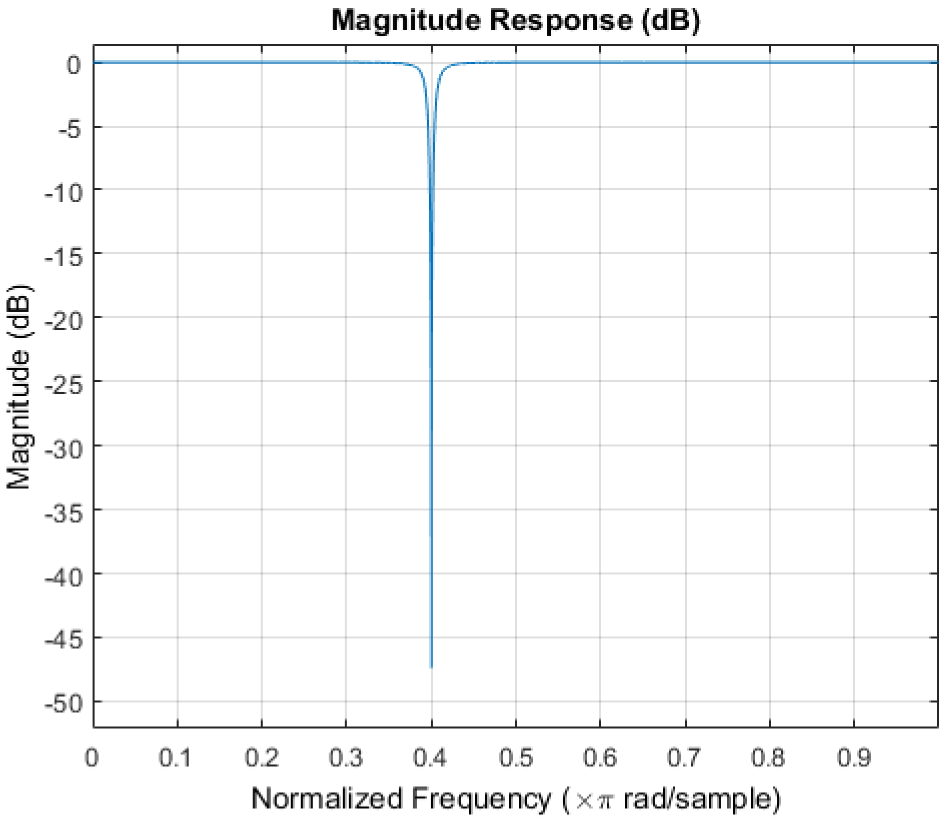
`[num,den] = iirnotch(w0,bw,ab)` returns a digital notching filter whose bandwidth, `bw`, is specified at a level of -`ab` decibels. Including the optional input argument `ab` lets you specify the magnitude response bandwidth at a level that is not the default -3 dB point, such as -6 dB or 0 dB.

Examples

Design an IIR Notch Filter Using iirnotch

Design and plot an IIR notch filter that removes a 60 Hz tone (f0) from a signal at 300 Hz (fs). For this example, set the Q factor for the filter to 35 and use it to specify the filter bandwidth:

```
wo = 60/(300/2);  bw = wo/35;
[b,a] = iirnotch(wo,bw);
fvtool(b,a);
```



The notch filter has the desired bandwidth with the notch located at 60 Hz, or 0.4? radians per sample. Compare this plot to the comb filter plot shown on the reference page for iircomb.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

`firgr` | `iircomb` | `iirpeak` | `iirparameq`

Introduced in R2011a

iirparameq

IIR biquad digital parametric equalizer filter

Compatibility

`iirparameq` function will be removed in a future release. Existing code using the function continues to run. For new code, use the `designParamEQ` function from Audio System Toolbox instead.

Syntax

```
[SOS,SV] = iirparameq(N,G,Wo,BW)
```

```
[SOS,SV,B,A] = iirparameq(N,G,Wo,BW)
```

Description

`[SOS,SV] = iirparameq(N,G,Wo,BW)` returns the coefficients of an `Nth` order IIR biquad digital parametric equalizer with gain `G`, center frequency `Wo`, and bandwidth `BW`. The coefficients are returned in a second-order section matrix, `SOS`, and a vector of scale values between each biquad stage, `SV`.

`[SOS,SV,B,A] = iirparameq(N,G,Wo,BW)` additionally returns a matrix of numerator fourth-order sections, B, and a matrix `A` of denominator fourth-order sections, A. These can be used in lieu of a biquad implementation and are useful for the case `Wo = 0.5`.

Examples

Second-order section matrices of the parametric equalizer

Compute the second-order section matrix and scale values of a parametric equalizer.

```
[SOS,SV] = iirparameq(6,5,0.0042,0.0028)
```

```
SOS =
```

```
    1.0000   -1.9892    0.9894    1.0000   -1.9911    0.9912
    1.0000   -1.9926    0.9929    1.0000   -1.9941    0.9944
    1.0000   -1.9964    0.9965    1.0000   -1.9967    0.9968

SV =

    1.0009
    1.0000
    1.0009
    1.0000
```

Fourth-order section matrices of the parametric equalizer

Compute the numerator and denominator coefficients of the fourth-order sections of the parametric equalizer.

```
[SOS,SV,B,A] = iirparameq(6,5,0.0042,0.0028)

SOS =

    1.0000   -1.9892    0.9894    1.0000   -1.9911    0.9912
    1.0000   -1.9926    0.9929    1.0000   -1.9941    0.9944
    1.0000   -1.9964    0.9965    1.0000   -1.9967    0.9968

SV =

    1.0009
    1.0000
    1.0009
    1.0000

B =

    1.0009   -1.9911    0.9903         0         0
    1.0009   -3.9927    5.9729   -3.9715    0.9903

A =

    1.0000   -1.9911    0.9912         0         0
    1.0000   -3.9908    5.9729   -3.9733    0.9912
```

Two Equalizers Centered at Different Frequencies

Design two equalizers centered at 100 Hz and 1000 Hz respectively, both with a gain of 5 dB and a Q-factor of 1.5, for a system running at 48 kHz.

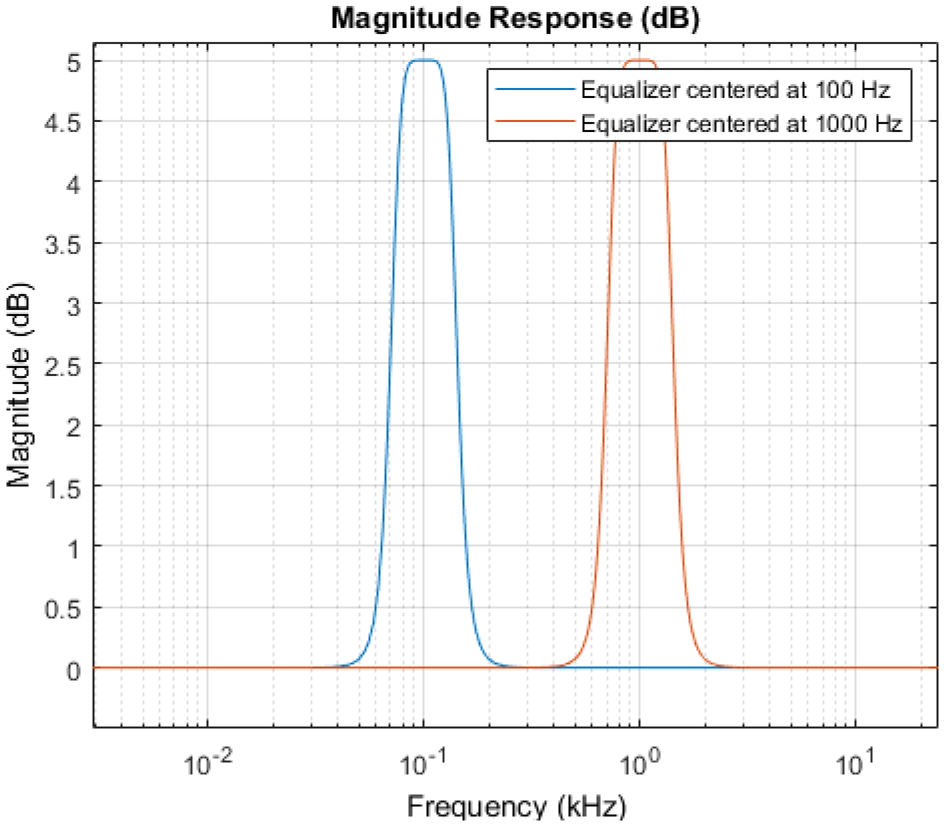
```
Fs  = 48e3;
N   = 6;
G   = 5;
Q   = 1.5;
Wo1 = 100/(Fs/2);
Wo2 = 1000/(Fs/2);
% Obtain the bandwidth of the equalizers from the center frequencies and
% Q-factors.
BW1 = Wo1/Q;
BW2 = Wo2/Q;
% Design the equalizers and obtain their SOS and SV values.
[SOS1,SV1] = iirparameq(N,G,Wo1,BW1);
[SOS2,SV2] = iirparameq(N,G,Wo2,BW2);
```

Design biquad filters using the obtained SOS and SV values.

```
BQ1 = dsp.BiquadFilter('SOSMatrix',SOS1,'ScaleValues',SV1);
BQ2 = dsp.BiquadFilter('SOSMatrix',SOS2,'ScaleValues',SV2);
```

Plot the magnitude response of both filters using a log scale.

```
fvtool(BQ1,BQ2,'Fs',Fs,'FrequencyScale','Log');
legend('Equalizer centered at 100 Hz','Equalizer centered at 1000 Hz');
```



Compare Notch Filters Designed with Different Orders

Design an eighth-order notch filter and compare it to a traditional second-order notch filter designed with `IIRNOTCH`.

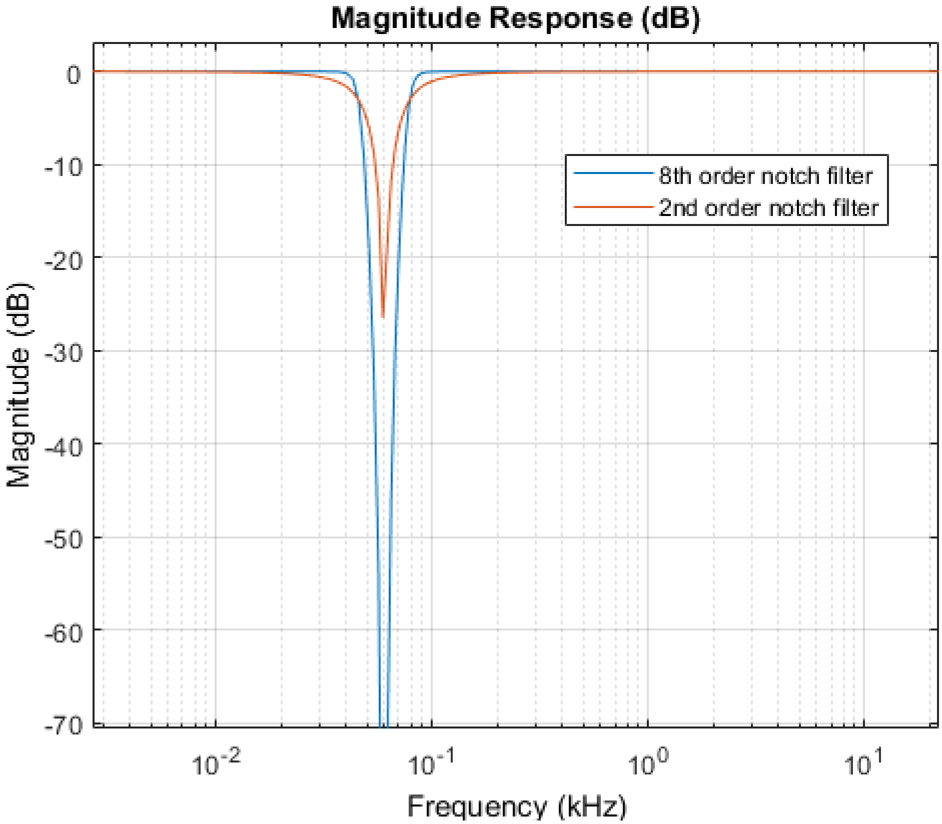
```
Fs  = 44.1e3;
N   = 8;
G   = -inf;
Q   = 1.8;
Wo  = 60/(Fs/2); % Notch at 60 Hz
BW  = Wo/Q; % Bandwidth occurs at -3 dB for this special case
[SOS1,SV1] = iirparameq(N,G,Wo,BW);
[NUM,DEN]  = iirnotch(Wo,BW);
SOS2 = [NUM,DEN];
```

Design the notch filters using the SOS and SV values.

```
BQ1 = dsp.BiquadFilter('SOSMatrix',SOS1,'ScaleValues',SV1);
BQ2 = dsp.BiquadFilter('SOSMatrix',SOS2);
```

Plot the magnitude response of both filters. The filters intersect at -3 dB point.

```
FVT = fvtool(BQ1,BQ2,'Fs',Fs,'FrequencyScale','Log');
legend(FVT,'8th order notch filter','2nd order notch filter');
```



Input Arguments

N — Order of the parametric equalizer

even positive integer

Order of the parametric equalizer, specified as an even positive integer.

Example: 6

Example: 10

Data Types: `single` | `double`

G — Gain of the parametric equalizer

real scalar

Gain of the parametric equalizer in dB, specified as a real scalar.

Example: 2

Example: -2.2

Data Types: single | double

Wo — Center frequency of the parametric equalizer

real scalar

Center frequency of the parametric equalizer, specified as a real scalar in the range [0.0 1.0]. A value of 1.0 corresponds to π radians/sample.

Example: 0.0

Example: 1.0

Data Types: single | double

BW — Bandwidth of the parametric equalizer

real scalar

Bandwidth of the parametric equalizer, specified as a real scalar in the range [0.0 1.0]. A value of 1.0 corresponds to π radians/sample.

Example: 0.0

Example: 1.0

Data Types: single | double

Output Arguments

SOS — Second-order section matrix

real-valued matrix

Second-order section matrix, returned as a real-valued L-by-6 matrix, where L is the number of second-order sections of the filter.

SV — Scale values between each biquad stage

real-valued vector

Scale values between each biquad stage, returned as a real-valued vector of length $L + 1$.

B — Numerator coefficients of the fourth-order sections of the parametric equalizer
real-valued matrix

Numerator coefficients of the fourth-order sections of the parametric equalizer, returned as a real-valued M-by-5 matrix. M is the number of fourth-order sections of the filter.

A — Denominator coefficients of the fourth-order sections of the parametric equalizer
real-valued matrix

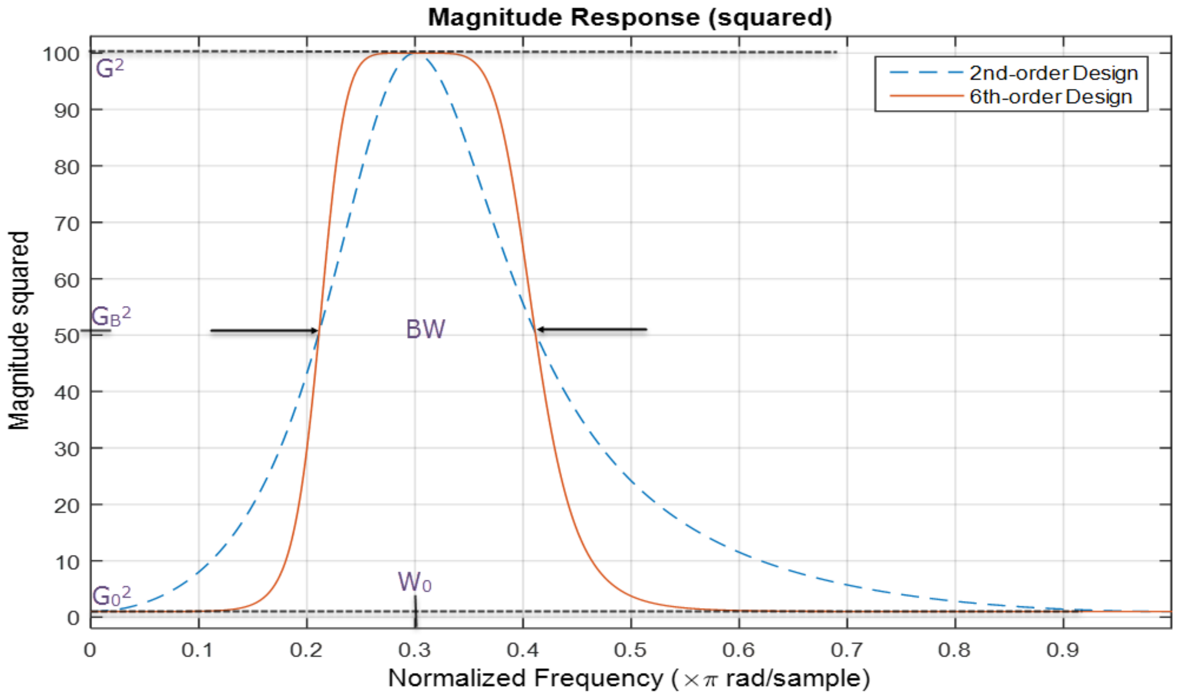
Denominator coefficients of the fourth-order sections of the parametric equalizer, returned as a real-valued M-by-5 matrix. M is the number of fourth-order sections of the filter.

More About

High-order Parametric Equalizer

Parametric equalizers are digital filters used in audio processing for adjusting the frequency content of a sound signal. Parametric equalizers provide capabilities beyond those of graphic equalizers by allowing the adjustment of gain, center frequency, and bandwidth of each filter. In contrast, graphic equalizers allow for the adjustment of the gain of each filter only. Digital parametric audio equalizers are commonly implemented as biquadratic (second-order IIR) filters. Due to low order, biquadratic filters can present relatively large ripple or transition regions. When several filters are connected in cascade, they can overlap with each other. In such circumstances, high-order filters are preferred. High-order filters provide flatter passbands, sharper band edges, and more control over the shape of each filter. In addition, if the order of the filter is set to two, the design changes to a special case: a traditional second-order parametric equalizer.

The figure shows the magnitude response of a second-order parametric equalizer compared with a high-order parametric equalizer.



$W_0 = 0.3 \times \pi$ radians/sample is the center frequency of the equalizer, $BW = 0.2$ radians/sample is the bandwidth of the equalizer, $G = 10$ is the peak gain of the equalizer, $G_0 = 1$, and $G_B = (G_0^2 + G^2) / 2$.

Algorithm

The first step in the filter design is to design a high-order analog lowpass shelving filter that meets the specified gain and bandwidth. A high-order Butterworth filter is used for this purpose. The analog Butterworth filter is then transformed into a digital lowpass shelving filter, and finally into a peaking equalizer that is centered at the specified peak frequency.

The design specifications for the digital equalizer are the order of the equalizer, N, the gain of the equalizer, G, the center frequency of the equalizer, W_0, and the bandwidth of the equalizer, BW.

The transfer function of the high-order parametric equalizer is given by:

$$H(z) = [\frac{b_{00} + b_{01}z^{-1} + b_{02}z^{-2}}{1 + a_{01}z^{-1} + a_{02}z^{-2}}]^r \times \prod_{i=1}^{L} [\frac{b_{i0} + b_{i1}z^{-1} + b_{i2}z^{-2} + b_{i3}z^{-3} + b_{i4}z^{-4}}{1 + a_{i1}z^{-1} + a_{i2}z^{-2} + a_{i3}z^{-3} + a_{i4}z^{-4}}]$$

where b_{00}, b_{01}, b_{02}, a_{01}, and a_{02} are coefficients of the second-order section of the equalizer. b_{i0}, b_{i1}, b_{i2}, b_{i3}, b_{i4}, a_{i1}, a_{i2}, and a_{i3} are coefficients of the fourth-order sections of the equalizer. $L = (N - r) / 2$, where $r = 1$ when N is odd, and $r = 0$ when N is even. The fourth-order sections are factored into second-order sections so that you can implement them using biquad filters.

For more information on how coefficients are computed in terms of the design specifications, see the "Butterworth Designs" section in [1].

References

[1] Sophocles J. Orphanidis. "High-Order Digital Parametric Equalizer Design." *J. Audio Eng. Soc.* Vol. 53, November 2005, pp. 1026–1046.

See Also

See Also

dsp.ParametricEQFilter | `iircomb` | `iirnotch` | `iirpeak`

Introduced in R2015a

iirpeak

Second-order IIR peak or resonator filter

Syntax

```
[num,den] = iirpeak(w0,bw)
[num,den] = iirpeak(w0,bw,ab)
```

Description

`[num,den] = iirpeak(w0,bw)` turns a second-order digital peaking filter with the peak located at `w0`, and with the bandwidth at the +3 dB point set to `bw`. To design the filter, `w0` must meet the condition 0.0 < `w0` < 1.0, where 1.0 corresponds to π radians per sample in the frequency range.

The quality factor (Q factor) q for the filter is related to the filter bandwidth by `q` = `w0/bw` where ω_0 is `w0` the signal frequency to boost.

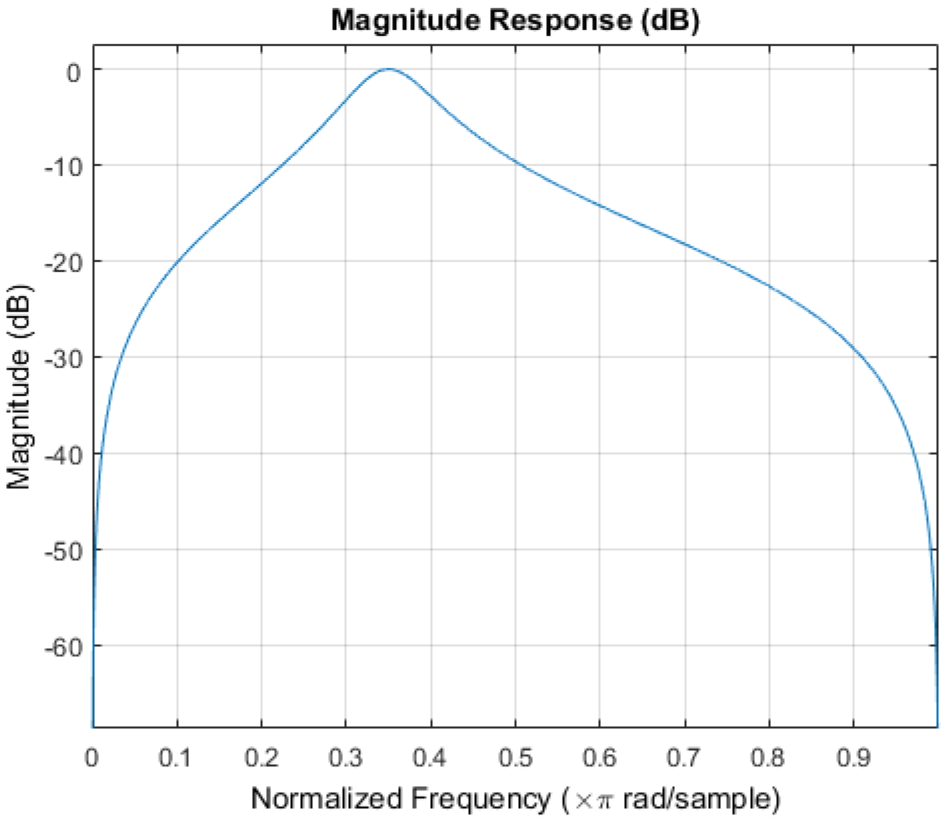
`[num,den] = iirpeak(w0,bw,ab)` returns a digital peaking filter whose bandwidth, `bw`, is specified at a level of `+ab` decibels. Including the optional input argument `ab` lets you specify the magnitude response bandwidth at a level that is not the default +3 dB point, such as +6 dB or 0 dB.

Examples

Design an IIR Peaking Filter Using iirpeak

Design and plot an IIR peaking filter that boosts the frequency at 1.75 Khz in a signal and has bandwidth of 500 Hz at the -3 dB point:

```
fs = 10000; wo = 1750/(fs/2);  bw = 500/(fs/2);
[b,a] = iirpeak(wo,bw);
fvtool(b,a);
```



The peak filter has the desired gain and bandwidth at 1.75 KHz.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

`firgr` | `iircomb` | `iirparameq` | `iirnotch`

Introduced in R2011a

iirpowcomp

Power complementary IIR filter

Syntax

```
[bp,ap] = iirpowcomp(b,a)
[bp,ap] = iirpowcomp(b,a,c)
```

Description

`[bp,ap] = iirpowcomp(b,a)` returns the coefficients of the power complementary IIR filter g(z) = `bp`(z)/`ap`(z) in vectors `bp` and `ap`, given the coefficients of the IIR filter h(z) = `b`(z)/`a`(z) in vectors `b` and `a`. `b` must be symmetric (Hermitian) or antisymmetric (antihermitian) and of the same length as `a`. The two power complementary filters satisfy the relation

$|H(w)|^2 + |G(w)|^2 = 1.$

`[bp,ap] = iirpowcomp(b,a,c)` where `c` is a complex scalar of unity magnitude, forces `bp` to satisfy the generalized hermitian property:

`conj(bp(end:-1:1)) = c*bp`.

When `c` is omitted, the function chooses `c` as follows:

- When `b` is real, the function chooses `c` as 1 or -1, whichever yields `bp` as real.
- When `b` is complex, `c` defaults to 1.

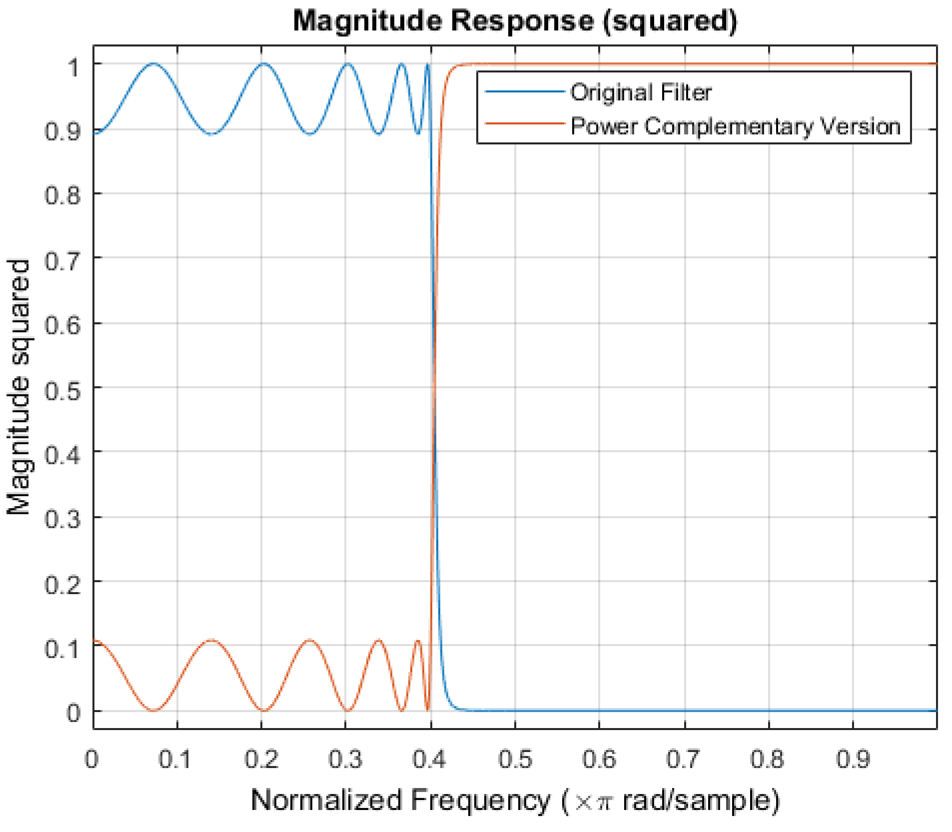
`ap` is always equal to `a`.

Examples

Power Complementary IIR Filter

```
[b,a]=cheby1(10,.5,.4);
[bp,ap]=iirpowcomp(b,a);
```

```
fvtool(b,a,bp,ap,'MagnitudeDisplay','Magnitude squared');
legend('Original Filter','Power Complementary Version');
```



See Also

See Also

Functions
`ca2tf` | `cl2tf` | `tf2ca` | `tf2cl`

Introduced in R2011a

iirrateup

Upsample IIR filter by integer factor

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirrateup(B,A,N)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirrateup(B,A,N)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter being transformed from any prototype by applying an `N`th-order rateup frequency transformation, where `N` is the upsample ratio. Transformation creates `N` equal replicas of the prototype filter frequency response.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

The relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Examples

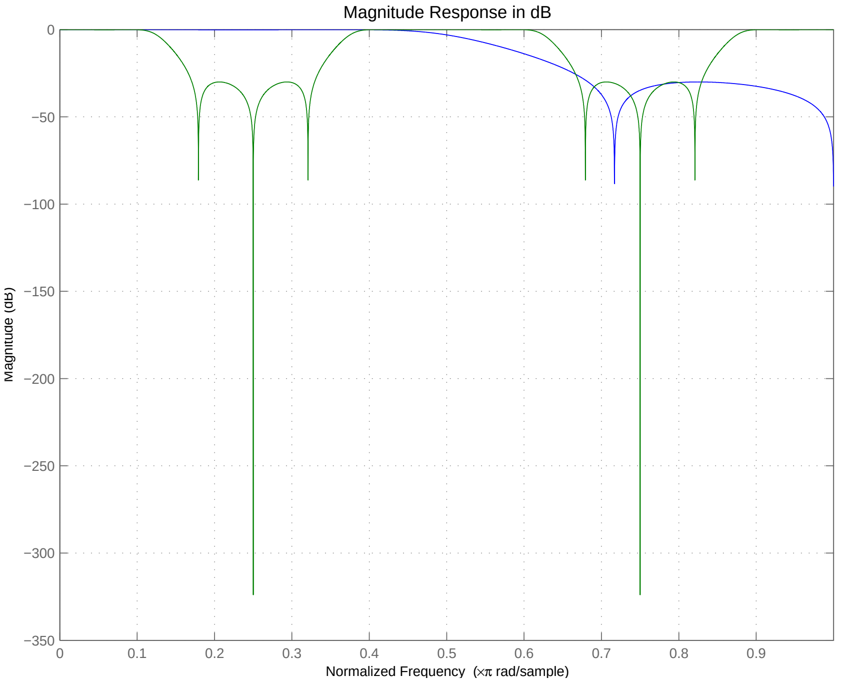
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
[num, den] = iirrateup(b, a, 4);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

As shown in the figure produced by FVTool, the transformed filter appears as expected.



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
N	Frequency multiplication ratio
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

iirftransf | zpkrateup | allpassrateup

Introduced in R2011a

iirshift

Shift frequency response of IIR filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirshift(B,A,Wo,Wt)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirshift(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a second-order real shift frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator of the allpass mapping filter, `AllpassDen`. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

This transformation places one selected feature of an original filter located at frequency W_o to the required target frequency location, W_t. This transformation implements the "DC Mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the real shift transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without designing them from the beginning.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

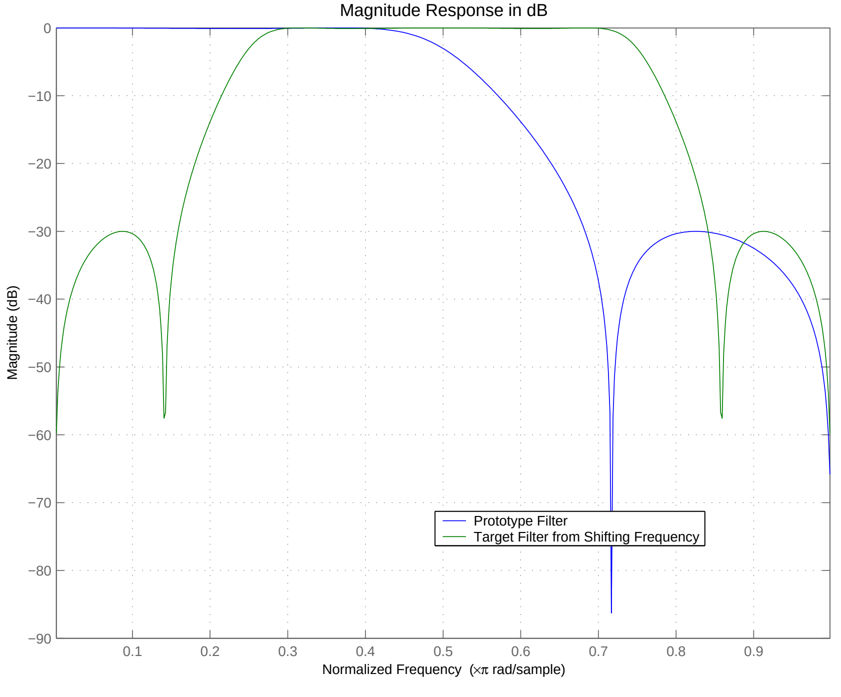
Perform the real frequency shift by defining where the selected feature of the prototype filter, originally at W_o=`0.5`, should be placed in the target filter, W_t=`0.75`:

```
Wo = 0.5; Wt = 0.75;
[num, den] = iirshift(b, a, Wo, Wt);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Shifting the specified feature from the prototype to the target generates the response shown in the figure.



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also

`iirftransf` | `zpkshift` | `allpassshift`

Introduced in R2011a

iirshiftc

Shift frequency response of IIR complex filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,Wo,Wc)
[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,0.5)
[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,-0.5)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,Wo,Wc)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a first-order complex frequency shift transformation. This transformation rotates all the features of an original filter by the same amount specified by the location of the selected feature of the prototype filter, originally at W_o, placed at W_t in the target filter.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,0.5)` calculates the allpass filter for doing the Hilbert transformation, i.e. a 90 degree counterclockwise rotation of an original filter in the frequency domain.

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,-0.5)` calculates the allpass filter for doing an inverse Hilbert transformation, i.e. a 90 degree clockwise rotation of an original filter in the frequency domain.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

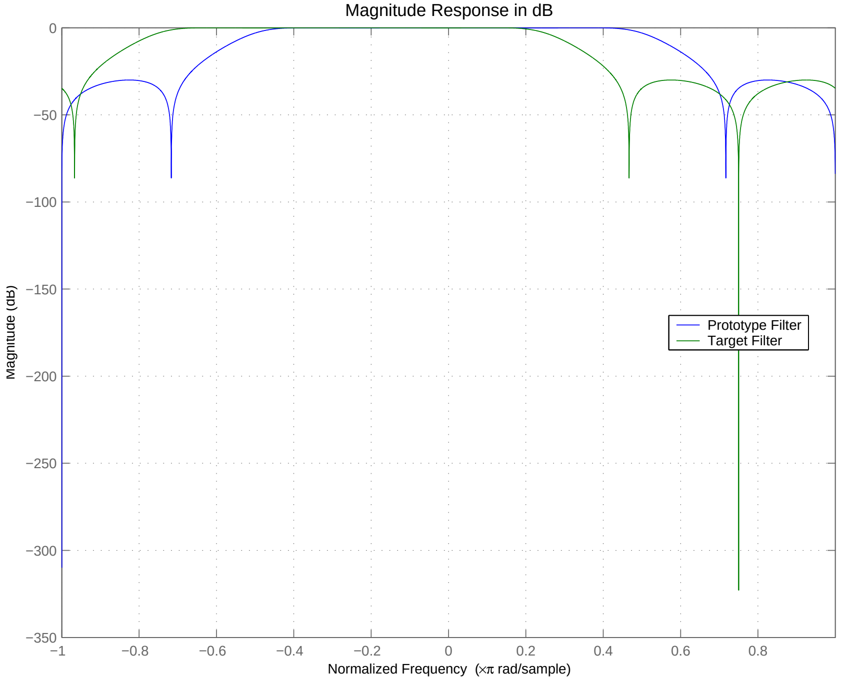
Rotate all features of the prototype filter in the frequency domain by the same amount by specifying where the selected feature of an original filter, W_o= `0.5`, should appear in the target filter, W_t= `0.25`:

```
[num, den] = iirshiftc(b, a, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

After applying the shift, the selected feature from the original filter is just where it should be, at W_t = `0.25`.



Arguments

Variable	Description
B	Numerator of the prototype lowpass filter

Variable	Description
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

References

Oppenheim, A.V., R.W. Schafer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

Dutta-Roy, S.C. and B. Kumar, "On digital differentiators, Hilbert transformers, and half-band low-pass filters," *IEEE Transactions on Education*, vol. 32, pp. 314-318, August 1989.

See Also

`iirftransf` | `zpkshiftc` | `allpassshiftc`

Introduced in R2011a

impz

Impulse response of discrete-time filter System object

Syntax

```
[impResp,t] = impz(sysobj)
[impResp,t] = impz(sysobj,n)
[impResp,t] = impz(sysobj,n,fs)
[impResp,t] = impz(sysobj,[],fs)
[impResp,t] = impz(sysobj,Name,Value)
impz(sysobj)
```

Description

`[impResp,t] = impz(sysobj)` computes the impulse response of the filter System object, `sysobj`, and returns the response in column vector `impResp`, and a vector of times (or sample intervals) in `t`, where `t = [0 1 2 ...k-1]'`. `k` is the number of filter coefficients.

`[impResp,t] = impz(sysobj,n)` computes the impulse response at `floor(n)` 1-second intervals. The time vector `t` equals `(0:floor(n)-1)'`.

`[impResp,t] = impz(sysobj,n,fs)` computes the impulse response at `floor(n)` 1/`fs`-second intervals. The time vector `t` equals `(0:floor(n)-1)'/fs`.

`[impResp,t] = impz(sysobj,[],fs)` computes the impulse response at `k` 1/`fs`-second intervals. `k` is the number of filter coefficients. The time vector `t` equals `(0:k-1)'/fs`.

`[impResp,t] = impz(sysobj,Name,Value)` returns an impulse response with additional options specified by one or more `Name,Value` pair arguments.

`impz(sysobj)` uses `fvtool` to plot the impulse response of the filter System object `sysobj`.

For more input options, refer to `impz` in Signal Processing Toolbox documentation.

Note You can use `impz` for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

Input Arguments

`sysobj`

Filter System object.

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpassFilter
dsp.AllpoleFilter
dsp.BiquadFilter
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.CICDecimator
dsp.CICInterpolator
dsp.CoupledAllpassFilter
dsp.FarrowRateConverter
dsp.FilterCascade
dsp.FIRDecimator
dsp.FIRFilter
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.FIRInterpolator
dsp.FIRRateConverter
dsp.HighpassFilter
dsp.IIRFilter
dsp.IIRHalfbandDecimator

Filter System objects
dsp.IIRHalfbandInterpolator
dsp.LowpassFilter
dsp.NotchPeakFilter
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

n

Length of the impulse response vector.

Default: Number of filter coefficients

fs

Sampling frequency.

Default: 1

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'Arithmetic' — Value types:
'double' | 'single' | 'fixed'

Specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data

System Object State	Coefficient Data Type	Rule
		type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

`impResp`

`n`-element impulse response vector. If `n` is not specified, the length of the impulse response vector equals the number of coefficients in the filter.

t

Time vector of length `n`, in seconds. `t` consists of `n` equally spaced points in the range `(0:floor(n)-1)'/fs`. If `n` is not specified, the function uses the number of coefficients of the filter.

Examples

Plot the Impulse Response of a Lowpass Elliptic Filter

Create a discrete-time filter for a fourth-order, lowpass elliptic filter with a cutoff frequency of 0.4 times the Nyquist frequency. Use a second-order sections structure to resist quantization errors. Plot the first 50 samples of the impulse response, along with the reference impulse response.

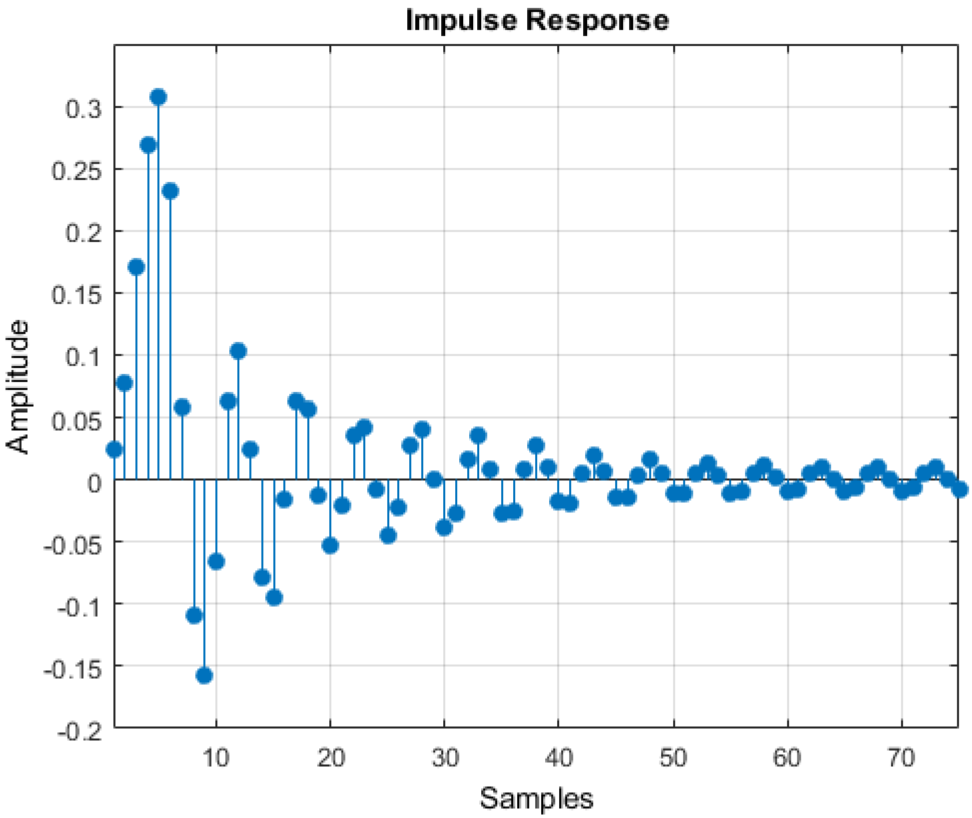
```
d = fdesign.lowpass(.4,.5,1,80);
```

Create a design object for the prototype filter. Use `ellip` to design a minimum order discrete-time biquad filter.

```
biquad = design(d,'ellip','Systemobject',true);
```

Plot the impulse response.

```
impz(biquad);
axis([1 75 -0.2 0.35])
```



See Also

See Also

`filter` | `stepz`

Introduced in R2011a

impzlength

Impulse response length

Syntax

```
len = impzlength(b,a)
len = impzlength(sos)
len = impzlength(d)

len = impzlength(hs)

len = impzlength( ___ ,tol)
```

Description

`len = impzlength(b,a)` returns the impulse response length for the causal discrete-time filter with the rational system function specified by the numerator, `b`, and denominator, `a`, polynomials in z^{-1}. For stable IIR filters, `len` is the effective impulse response sequence length. Terms in the IIR filter's impulse response after the `len`-th term are essentially zero.

`len = impzlength(sos)` returns the effective impulse response length for the IIR filter specified by the second order sections matrix, `sos`. `sos` is a K-by-6 matrix, where the number of sections, K, must be greater than or equal to 2. If the number of sections is less than 2, `impzlength` considers the input to be the numerator vector, `b`. Each row of `sos` corresponds to the coefficients of a second order (biquad) filter. The ith row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`len = impzlength(d)` returns the impulse response length for the digital filter, `d`. Use `designfilt` to generate `d` based on frequency-response specifications.

`len = impzlength(hs)` returns the impulse response length for the filter System object, `hs`. You must have the DSP System Toolbox software to use `impzlength` with a filter System object.

`len = impzlength( ___ ,tol)` specifies a tolerance for estimating the effective length of an IIR filter's impulse response. By default, `tol` is `5e-5`. Increasing the value of `tol` estimates a shorter effective length for an IIR filter's impulse response. Decreasing the value of `tol` produces a longer effective length for an IIR filter's impulse response.

Examples

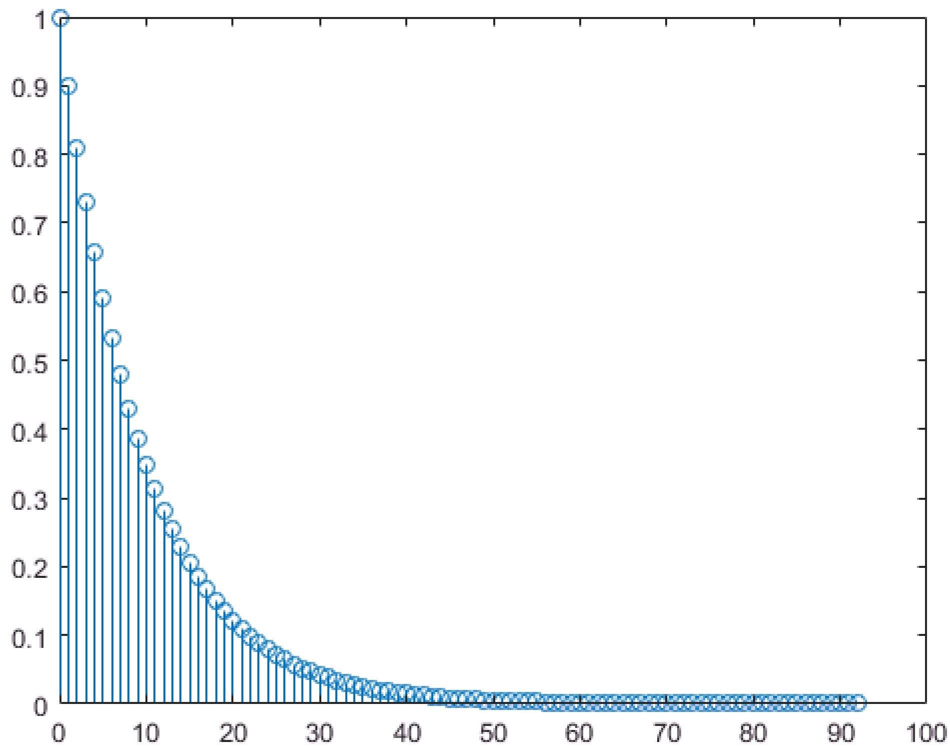
IIR Filter Effective Impulse Response Length — Coefficients

Create a lowpass allpole IIR filter with a pole at 0.9. Calculate the effective impulse response length. Obtain the impulse response. Plot the result.

```
b = 1;
a = [1 -0.9];
len = impzlength(b,a)
```

```
len = 93
```

```
[h,t] = impz(b,a);
stem(t,h)
```




```
h(len)
```

```
ans = 6.1704e-05
```

IIR Filter Effective Impulse Response Length — Second-Order Sections

Design a 4th-order lowpass elliptic filter with a cutoff frequency of 0.4π rad/sample. Specify 1 dB of passband ripple and 60 dB of stopband attenuation. Design the filter in pole-zero-gain form and obtain the second-order section matrix using `zp2sos`. Determine the effective impulse response sequence length from the second-order section matrix.

```
[z,p,k] = ellip(4,1,60,.4);
```

```
[sos,g] = zp2sos(z,p,k);
len = impzlength(sos)
```

```
len = 80
```

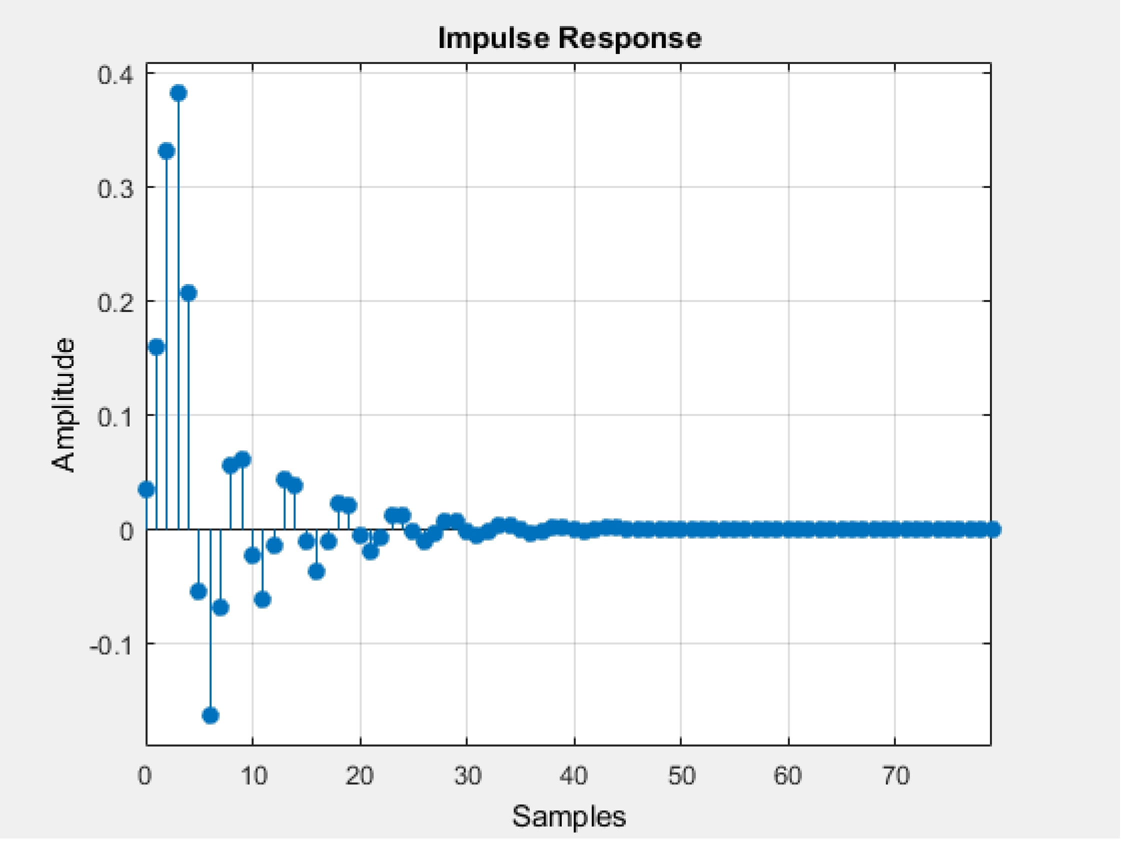
IIR Filter Effective Impulse Response Length — Digital Filter

Use `designfilt` to design a 4th-order lowpass elliptic filter with normalized passband frequency 0.4π rad/sample. Specify 1 dB of passband ripple and 60 dB of stopband attenuation. Determine the effective impulse response sequence length and visualize it.

```
d = designfilt('lowpassiir','FilterOrder',4,'PassbandFrequency',0.4, ...
               'PassbandRipple',1,'StopbandAttenuation',60, ...
               'DesignMethod','ellip');
len = impzlength(d)
```

```
len = 80
```

```
impz(d)
```



Impulse Response Length of Filter System object

This example requires DSP System Toolbox™ software.

Design a 4th-order lowpass elliptic filter with a cutoff frequency of 0.4*pi rad/sample. Specify 1 dB of passband ripple and 60 dB of stopband attenuation. Design the filter in pole-zero-gain form and obtain the second order section matrix using `zp2sos`. Create a biquad filter System object™ and input the System object to `impzlength`.

```
[z,p,k] = ellip(4,1,60,.4);
[sos,g] = zp2sos(z,p,k);
hBqdFilt = dsp.BiquadFilter('Structure','Direct form I',...
                                              'SOSMatrix', sos,...
```

```
                                              'ScaleValues',g);
len = impzlength(hBqdFilt)


len =

    80
```

Impulse Response Length — Filter System Objects

Design IIR Butterworth and FIR equiripple filters for data sampled at 1 kHz. The passband frequency is 100 Hz and the stopband frequency is 150 Hz. The passband ripple is 0.5 dB and there is 60 dB of stopband attenuation. Obtain System objects for the filters and compare the filter impulse response sequence lengths.

```
d = fdesign.lowpass('Fp,Fst,Ap,Ast',100,150,0.5,60,1000);
Hd1 = design(d,'butter','SystemObject',true);
Hd2 = design(d,'equiripple','SystemObject',true);
len = [impzlength(Hd1) impzlength(Hd2)]


len =

   183    49
```

Input Arguments

b — Numerator coefficients

vector | scalar

Numerator coefficients, specified as a scalar (allpole filter) or a vector.

Example: `b = fir1(20,0.25)`

Data Types: `single` | `double`
Complex Number Support: Yes

a — Denominator coefficients

vector | scalar

Denominator coefficients, specified as a scalar (FIR filter) or vector.

Data Types: `single` | `double`
Complex Number Support: Yes

`sos` — Matrix of second order sections

matrix

Matrix of second order sections, specified as a K-by-6 matrix. The system function of the K-th biquad filter has the rational Z-transform

$$H_k(z) = \frac{B_k(1) + B_k(2)z^{-1} + B_k(3)z^{-2}}{A_k(1) + A_k(2)z^{-1} + A_k(3)z^{-2}}.$$

The coefficients in the Kth row of the matrix, `sos`, are ordered as follows.

$$\begin{bmatrix} B_k(1) & B_k(2) & B_k(3) & A_k(1) & A_k(2) & A_k(3) \end{bmatrix}$$

The frequency response of the filter is the system function evaluated on the unit circle with

$$z = e^{j2\pi f}.$$

`d` — Digital filter

`digitalFilter` object

Digital filter, specified as a digitalFilter object. Use `designfilt` to generate a digital filter based on frequency-response specifications.

Example: `d = designfilt('lowpassiir','FilterOrder',3,'HalfPowerFrequency',0.5)` specifies a third-order Butterworth filter with normalized 3-dB frequency 0.5π rad/sample.

`hs` — Filter System object

filter System object

Filter System object, specified as one of the following:

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpassFilter
dsp.AllpoleFilter
dsp.BiquadFilter
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.CICDecimator
dsp.CICInterpolator
dsp.CoupledAllpassFilter
dsp.FarrowRateConverter
dsp.FilterCascade
dsp.FIRDecimator
dsp.FIRFilter
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.FIRInterpolator
dsp.FIRRateConverter
dsp.HighpassFilter
dsp.IIRFilter
dsp.IIRHalfbandDecimator
dsp.IIRHalfbandInterpolator
dsp.LowpassFilter
dsp.NotchPeakFilter
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

Using `impzlength` with a filter System object requires the DSP System Toolbox software.

`tol` — Tolerance for IIR filter effective impulse response length

`5e-5` (default) | positive scalar

Tolerance for IIR filter effective impulse response length, specified as a positive number. The tolerance determines the term in the absolutely summable sequence after which subsequent terms are considered to be 0. The default tolerance is `5e-5`. Increasing the tolerance returns a shorter effective impulse response sequence length. Decreasing the tolerance returns a longer effective impulse response sequence length.

Output Arguments

`len` — Length of impulse response

positive integer

Length of the impulse response, specified as a positive integer. For stable IIR filters with absolutely summable impulse responses, `impzlength` returns an effective length for the impulse response beyond which the coefficients are essentially zero. You can control this cutoff point by specifying the optional `tol` input argument.

See Also

See Also

digitalFilter | `designfilt` | `impz` | `zp2sos`

Introduced in R2013a

info

Information about filter

Syntax

```
info(obj)
info(obj,'short')
info(obj,'long')
s = info(obj)
```

Description

`info(obj)` returns very basic information about the filter System object, `obj`. The particulars depend on the filter type and structure.

`info(obj,'short')` returns the same information as `info(obj)`.

`info(obj,'long')` returns the following information about the filter:

- Specifications such as the filter structure and filter order
- Information about the design method and options
- Performance measurements for the filter response, such as the passband cutoff or stopband attenuation, included in the `measure` method
- Cost of implementing the filter in terms of operations required to apply the filter to data, included in the `cost` method

When the filter uses fixed-point arithmetic, the function returns additional information about the filter, including the arithmetic setting and details about the filter internals.

`s = info(obj)` returns filter information in the variable `s`. You can also provide the optional arguments `'short'` and `'long'` with this syntax.

Input Arguments

`obj`

One of the following types of filter System objects:

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpassFilter
dsp.AllpoleFilter
dsp.BiquadFilter
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.CICDecimator
dsp.CICInterpolator
dsp.CoupledAllpassFilter
dsp.FarrowRateConverter
dsp.FilterCascade
dsp.FIRDecimator
dsp.FIRFilter
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.FIRInterpolator
dsp.FIRRateConverter
dsp.HighpassFilter
dsp.IIRFilter
dsp.IIRHalfbandDecimator
dsp.IIRHalfbandInterpolator
dsp.LowpassFilter
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

'short'

Input argument that instructs the function to return basic information about the filter.

'long'

Input argument that instructs the function to return in-depth information about the filter.

Output Arguments

s

Variable for storing filter information.

Examples

Obtain Filter Information

Obtain short-format and long-format information about a filter.

```
d = fdesign.lowpass;
f = design(d,'SystemObject',true);
info(f)
```

```
ans =

  6×35 char array

    'Discrete-Time FIR Filter (real)    '
    '-------------------------------    '
    'Filter Structure  : Direct-Form FIR'
    'Filter Length     : 43             '
    'Stable            : Yes            '
    'Linear Phase      : Yes (Type 1)   '
```

```
info(f,'long')
```

```
ans =

  45×45 char array

    'Discrete-Time FIR Filter (real)              '
    '-------------------------------              '
```

```
'Filter Structure  : Direct-Form FIR          '
'Filter Length     : 43                       '
'Stable            : Yes                      '
'Linear Phase      : Yes (Type 1)             '
'                                             '
'Design Method Information                    '
'Design Algorithm : equiripple                '
'                                             '
'Design Options                               '
'Density Factor : 16                          '
'Maximum Phase  : false                       '
'Minimum Order  : any                         '
'Minimum Phase  : false                       '
'Stopband Decay : 0                           '
'Stopband Shape : flat                        '
'SystemObject   : true                        '
'Uniform Grid   : true                        '
'                                             '
'Design Specifications                        '
'Sample Rate     : N/A (normalized frequency) '
'Response        : Lowpass                    '
'Specification   : Fp,Fst,Ap,Ast              '
'Stopband Atten. : 60 dB                      '
'Passband Ripple : 1 dB                       '
'Stopband Edge   : 0.55                       '
'Passband Edge   : 0.45                       '
'                                             '
'Measurements                                 '
'Sample Rate      : N/A (normalized frequency)'
'Passband Edge    : 0.45                      '
'3-dB Point       : 0.46957                   '
'6-dB Point       : 0.48314                   '
'Stopband Edge    : 0.55                      '
'Passband Ripple  : 0.89042 dB                '
'Stopband Atten.  : 60.945 dB                 '
'Transition Width : 0.1                       '
'                                             '
'Implementation Cost                          '
'Number of Multipliers            : 43        '
'Number of Adders                 : 42        '
'Number of States                 : 42        '
'Multiplications per Input Sample : 43        '
'Additions per Input Sample       : 42        '
```

See Also

See Also

Functions
`coeffs` | `isfir` | `islinphase` | `isstable`

Introduced in R2011a

int

States from CIC filter

Compatibility

`mfilt` will be removed in a future release. Refer to the reference page for a specific `mfilt` object to see its recommended replacement.

Syntax

`integerstates = int(hm.states)`

Description

`integerstates = int(hm.states)` returns the states of a CIC filter in matrix form, rather than as the native `filtstates` object. An important point about `int` is that it quantizes the state values to the smallest number of bits possible while maintaining the values accurately.

Examples

For many users, the states of multirate filters are most useful as a matrix, but the CIC filters store the states as objects. Here is how you get the states from you CIC filter as a matrix.

```
hm = mfilt.cicinterp;
hs = hm.states; % Returns a FILTSTATES.CIC object hs.
states = int(hs); % Convert object hs to a signed integer matrix.
```

After using int to convert the states object to a matrix, here is what you get.

Before converting:

```
hm.states

ans =
```

```
    Integrator: [2x1 States]
          Comb: [2x1 States]
```

After the conversion and assigning the states to `states`:

```
states

states =

           0            0
           0            0
```

See Also

`filtstates.cic` | dsp.CICInterpolator | dsp.CICDecimator

Introduced in R2011a

isallpass

Determine whether filter is allpass

Syntax

```
flag = isallpass(b,a)
flag = isallpass(sos)
flag = isallpass(d)
flag = isallpass(...,tol)
flag = isallpass(hs,...)
flag = isallpass(hs,'Arithmetic',arithtype)
```

Description

`flag = isallpass(b,a)` returns a logical output, `flag`, equal to `true` if the filter specified by numerator coefficients, `b`, and denominator coefficients, `a`, is an allpass filter. If the filter is not an allpass filter, `flag` is equal to `false`.

`flag = isallpass(sos)` returns `true` if the filter specified by second order sections matrix, `sos`, is an allpass filter. `sos` is a *K*-by-6 matrix, where the number of sections, *K*, must be greater than or equal to 2. Each row of `sos` corresponds to the coefficients of a second order (biquad) filter. The *i*th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`flag = isallpass(d)` returns `true` if the digital filter, `d`, is an allpass filter. Use `designfilt` to generate `d` based on frequency-response specifications.

`flag = isallpass(...,tol)` uses the tolerance, `tol`, to determine when two numbers are close enough to be considered equal. If not specified, `tol`, defaults to `eps^(2/3)`. Specifying a tolerance may be most helpful in fixed-point allpass filters.

`flag = isallpass(hs,...)` returns `true` if the filter System object, `hs`, is an allpass filter. You must have the DSP System Toolbox software to use this syntax.

`flag = isallpass(hs,'Arithmetic',arithtype)` analyzes the filter System object `hs` based on the specified *`arithtype`*. *`arithtype`* can be `'double'`, `'single'`, or 'fixed'. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed' , the arithmetic changes depending on the setting of the CoefficientDataType property and whether the System object is locked or unlocked. You must have the DSP System Toolbox software to use this syntax.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Examples

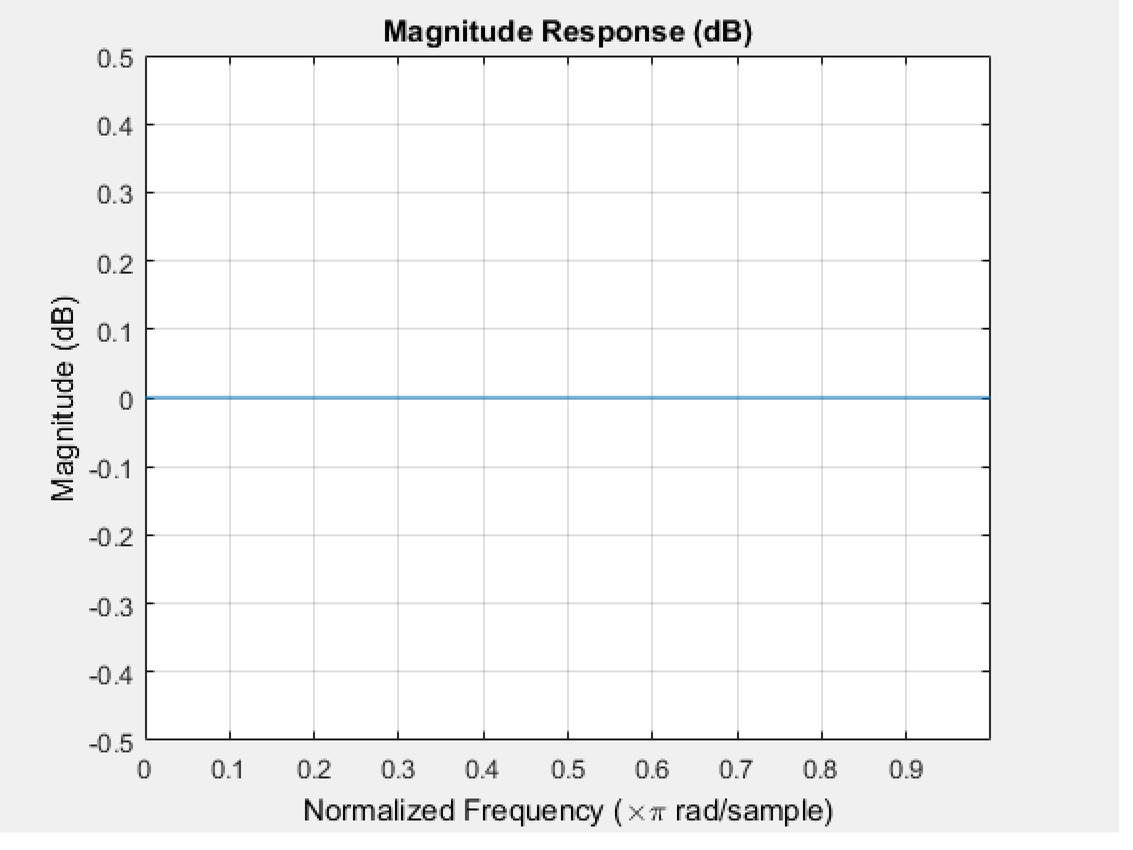
Allpass Filters

Create an allpass filter and verify that the frequency response is allpass.

```
b = [1/3 1/4 1/5 1];
a = fliplr(b);
flag = isallpass(b,a)
```

```
flag = logical
   1
```

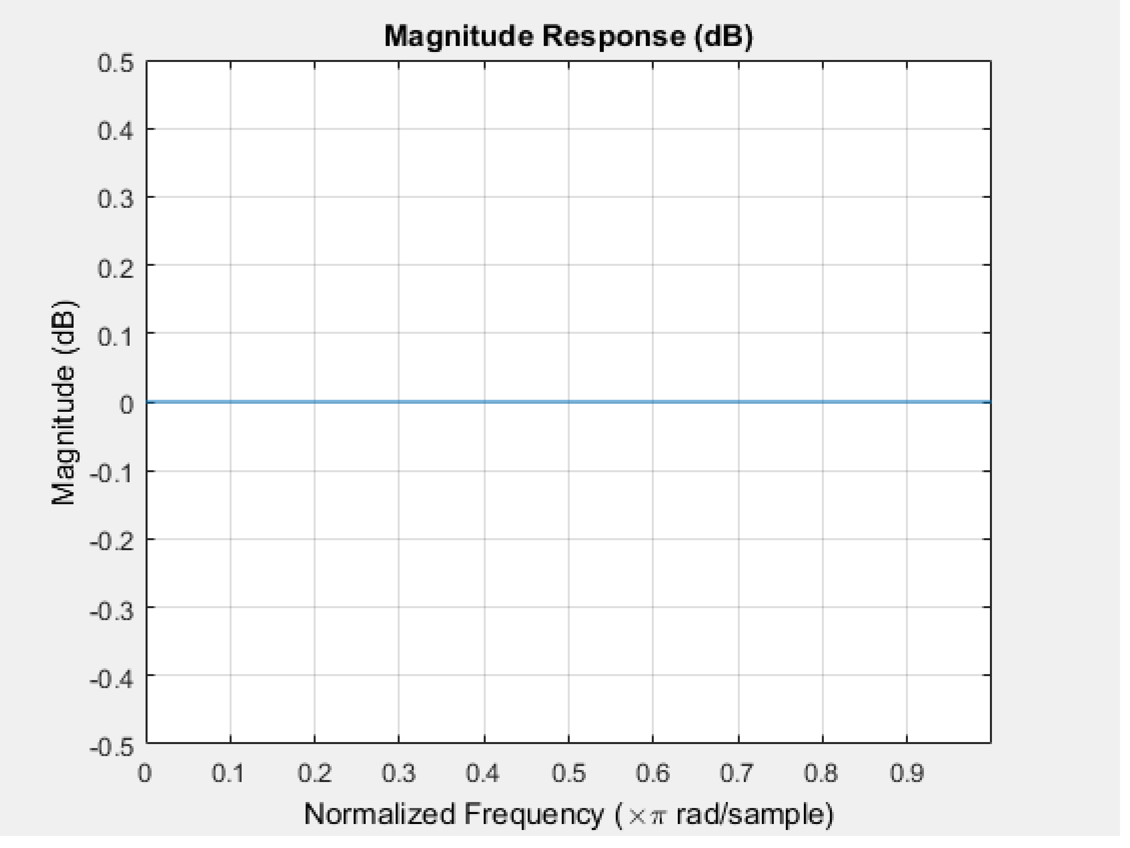
```
fvtool(b,a)
```



Create a lattice allpass filter and verify that the filter is allpass.

```
k = [1/2 1/3 1/4 1/5];
[b,a] = latc2tf(k,'allpass');
flag_isallpass = isallpass(b,a)

flag_isallpass = logical
   1
```

```
fvtool(b,a)
```



See Also

`designfilt` | digitalFilter | `islinphase` | `ismaxphase` | `isminphase` | `isstable`

Introduced in R2013a

isfir

Determine whether filter System object is FIR

Syntax

```
isfir(sysobj)
```

Description

`isfir(sysobj)` determines whether the filter System object `sysobj` is an FIR filter. If the filter is FIR, `isfir` returns `1`.

To determine whether `sysobj` is an FIR filter, `isfir(sysobj)` inspects if the filter in the transfer function form has a scalar denominator. If it does, it is an FIR filter.

Examples

Determine Whether an FIR Filter

Design a Lowpass FIR Filter.

```
d = fdesign.lowpass;
h = design(d,'Systemobject',true)
```

```
h =

  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: [1×43 double]
    InitialConditions: 0

  Use get to show all properties
```

Determine if the filter is an FIR filter using `isfir`.

```
isfir(h)
```

```
ans =

  logical

   1
```

`isfir` returns 1 to indicate that the filter is an FIR filter.

See Also

See Also

Functions
`isallpass` | `islinphase` | `ismaxphase` | `isminphase` | `isreal` | `issos` | `isstable`

Introduced in R2011a

islinphase

Determine whether filter has linear phase

Syntax

```
flag = islinphase(b,a)
flag = islinphase(sos)
flag = islinphase(d)
flag = islinphase(...,tol)
flag = islinphase(hs,...)
flag = islinphase(hs,'Arithmetic',arithtype)
```

Description

`flag = islinphase(b,a)` returns a logical output, `flag`, equal to `true` if the filter coefficients in `b` and `a` define a linear phase filter. `flag` is equal to `false` if the filter does not have linear phase.

`flag = islinphase(sos)` returns `true` if the filter specified by second order sections matrix, `sos`, has linear phase. `sos` is a *K*-by-6 matrix, where the number of sections, *K*, must be greater than or equal to 2. Each row of `sos` corresponds to the coefficients of a second order (biquad) filter. The *i*th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`flag = islinphase(d)` returns `true` if the digital filter, `d`, has linear phase. Use `designfilt` to generate `d` based on frequency-response specifications.

`flag = islinphase(...,tol)` uses the tolerance, `tol`, to determine when two numbers are close enough to be considered equal. If not specified, `tol`, defaults to `eps^(2/3)`.

`flag = islinphase(hs,...)` determines whether the filter System object, `hs`, has linear phase. You must have the DSP System Toolbox to use `islinphase` with a System object.

`flag = islinphase(hs,'Arithmetic',arithtype)` analyzes the filter System object `hs` based on the specified *`arithtype`*. *`arithtype`* can be one of `'double'`, 'single', or 'fixed'. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed' , the arithmetic changes depending on the setting of the CoefficientDataType property and whether the System object is locked or unlocked. You must have the DSP System Toolbox to use islinphase with a System object.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

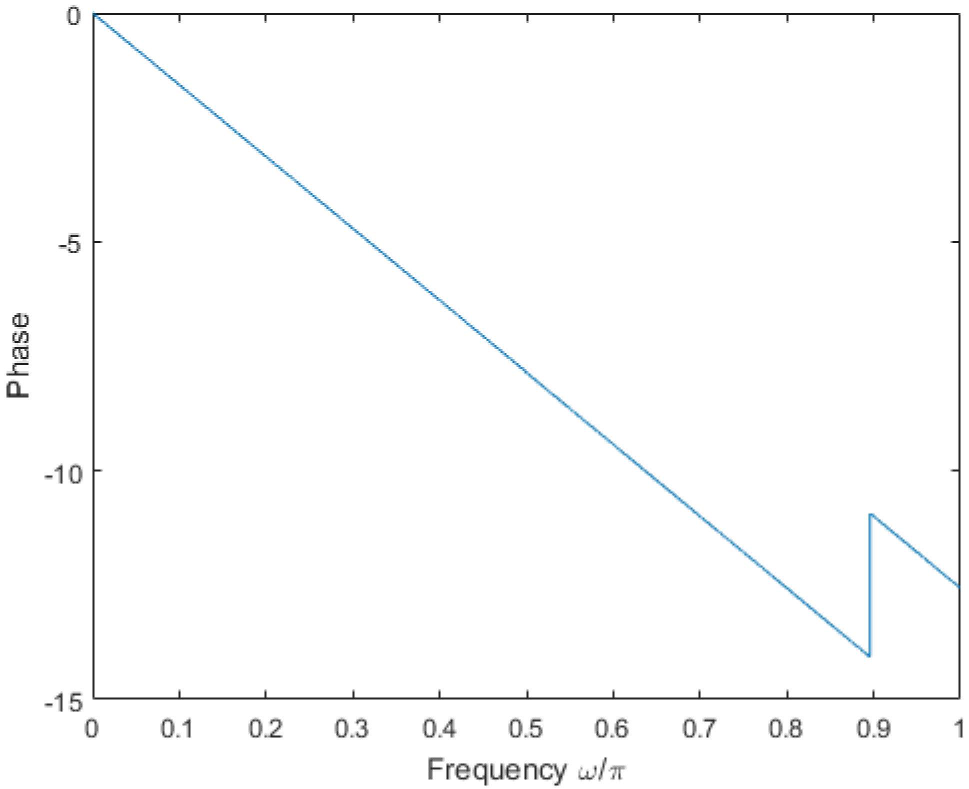
Examples

Linear and Nonlinear Phase

Use the window method to design a tenth-order lowpass FIR filter with normalized cutoff frequency 0.55. Verify that the filter has linear phase.

```
firSpecs = fdesign.lowpass('N,Fc',10,0.55);
lpFIR = design(firSpecs,'window','SystemObject',true);

flag = islinphase(lpFIR)
[phs,w] = phasez(lpFIR);

plot(w/pi,phs)
xlabel('Frequency \omega/\pi')
ylabel('Phase')
```

```
flag =

  logical

   1
```



IIR filters in general do not have linear phase. Verify the statement by constructing eighth-order Butterworth, Chebyshev, and elliptic filters with similar specifications.

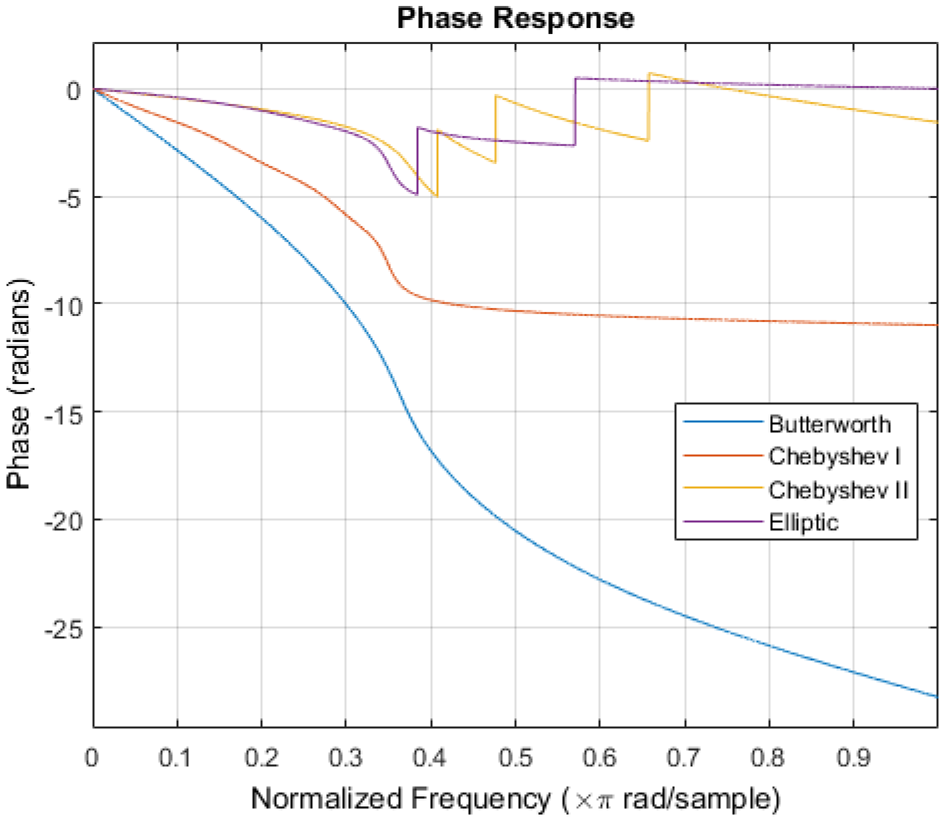
```
ord = 4;

Wp = 0.35;
Wst = 0.4;
atten = 20;
rippl = 1;

buttSpecs = fdesign.lowpass('Fp,Fst,Ap,Ast',Wp,Wst,rippl,atten);
buttIIR = design(buttSpecs,'butter','SystemObject',true);

chb1Specs = fdesign.lowpass('Fp,Fst,Ap,Ast',Wp,Wst,rippl,atten);
```

```
chb1IIR = design(chb1Specs,'cheby1','SystemObject',true);

chb2Specs = fdesign.lowpass('Fp,Fst,Ap,Ast',Wp,Wst,rippl,atten);
chb2IIR = design(chb2Specs,'cheby2','SystemObject',true);

ellpSpecs = fdesign.lowpass('Fp,Fst,Ap,Ast',Wp,Wst,rippl,atten);
ellpIIR = design(ellpSpecs,'ellip','SystemObject',true);
```

Plot the phase responses of the filters. Determine whether they have linear phase.

```
fv = fvtool(buttIIR,chb1IIR,chb2IIR,ellpIIR,'Analysis','phase');
legend(fv,'Butterworth','Chebyshev I','Chebyshev II','Elliptic')

phs = [islinphase(buttIIR) islinphase(chb1IIR) ...
       islinphase(chb2IIR) islinphase(ellpIIR)]
```

```
phs =

  1×4 logical array

   0   0   0   0
```



See Also

`designfilt` | digitalFilter | `isallpass` | `ismaxphase` | `isminphase` | `isstable`

Introduced in R2013a

ismaxphase

Determine whether filter is maximum phase

Syntax

```
flag = ismaxphase(b,a)
flag = ismaxphase(sos)
flag = ismaxphase(d)
flag = ismaxphase(...,tol)
flag = ismaxphase(hs,...)
flag = ismaxphase(hs,'Arithmetic',arithtype)
```

Description

`flag = ismaxphase(b,a)` returns a logical output, `flag`, equal to `true` if the filter specified by numerator coefficients, `b`, and denominator coefficients, `a`, is a maximum phase filter.

`flag = ismaxphase(sos)` returns `true` if the filter specified by second order sections matrix, `sos`, is a maximum phase filter. `sos` is a *K*-by-6 matrix, where the number of sections, *K*, must be greater than or equal to 2. Each row of `sos` corresponds to the coefficients of a second order (biquad) filter. The *i*th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`flag = ismaxphase(d)` returns `true` if the digital filter, `d`, has maximum phase. Use `designfilt` to generate `d` based on frequency-response specifications.

`flag = ismaxphase(...,tol)` uses the tolerance, `tol`, to determine when two numbers are close enough to be considered equal. If not specified, `tol`, defaults to `eps^(2/3)`.

`flag = ismaxphase(hs,...)` returns `true`if the filter System object `hs` is a maximum phase filter. You must have the DSP System Toolbox software to use this syntax.

`flag = ismaxphase(hs,'Arithmetic',arithtype)` analyzes the filter System object `hs` based on the specified *`arithtype`*. *`arithtype`* can be `'double'`, `'single'`, or `'fixed'`. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked. You must have the DSP System Toolbox software to use this syntax.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Examples

Maximum- and Minimum-Phase Filters

Design maximum-phase and minimum-phase lattice filters and verify their phase type.

```
k = [1/6 1/1.4];
bmax = latc2tf(k,'max');
bmin = latc2tf(k,'min');
max_flag = ismaxphase(bmax)
```

```
max_flag = logical
   1
```

```
min_flag = isminphase(bmin)
```

```
min_flag = logical
   1
```

Given a filter defined with a set of single precision numerator and denominator coefficients, check if it is maximum phase for different values of the tolerance.

```
b = single([1 -0.9999]);
a = single([1 0.45]);
max_flag1 = ismaxphase(b,a)
```

```
max_flag1 = logical
   0
```

```
max_flag2 = ismaxphase(b,a,1e-3)
```

```
max_flag2 = logical
   1
```

See Also

designfilt | digitalFilter | isallpass | islinphase | isminphase | isstable

Introduced in R2013a

isminphase

Determine whether filter is minimum phase

Syntax

```
flag = isminphase(b,a)
flag = isminphase(sos)
flag = isminphase(d)
flag = isminphase(...,tol)
flag = isminphase(hs,...)
isminphase(hs,'Arithmetic',arithtype)
```

Description

A filter is *minimum phase* when all the zeros of its transfer function are on or inside the unit circle, or the numerator is a scalar. An equivalent definition for a minimum phase filter is a causal and stable system with a causal and stable inverse.

`flag = isminphase(b,a)` returns a logical output, `flag`, equal to `true` if the filter specified by numerator coefficients, `b`, and denominator coefficients, `a`, is a minimum phase filter.

`flag = isminphase(sos)` returns `true` if the filter specified by second order sections matrix, `sos`, is minimum phase. `sos` is a *K*-by-6 matrix, where the number of sections, *K*, must be greater than or equal to 2. Each row of `sos` corresponds to the coefficients of a second order (biquad) filter. The *i*th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`flag = isminphase(d)` returns `true` if the digital filter, `d`, has minimum phase. Use `designfilt` to generate `d` based on frequency-response specifications.

`flag = isminphase(...,tol)` uses the tolerance, `tol`, to determine when two numbers are close enough to be considered equal. If not specified, `tol`, defaults to `eps^(2/3)`.

`flag = isminphase(hs,...)` determines whether the filter System object `hs` is minimum phase, returning `1` if true and `0` if false. You must have the DSP System Toolbox software to use this syntax.

`isminphase(hs,'Arithmetic',arithtype)` analyzes the filter System object `hs` based on the specified *`arithtype`*. *`arithtype`* can be `'double'`, `'single'`, or `'fixed'`. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked. You must have the DSP System Toolbox software to use this syntax.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Examples

Minimum Phase Filters

Design a sixth-order lowpass Butterworth IIR filter using second order sections. Specify a normalized 3-dB frequency of 0.15. Check if the filter has minimum phase.

```
[z,p,k] = butter(6,0.15);
SOS = zp2sos(z,p,k);
min_flag = isminphase(SOS)
```

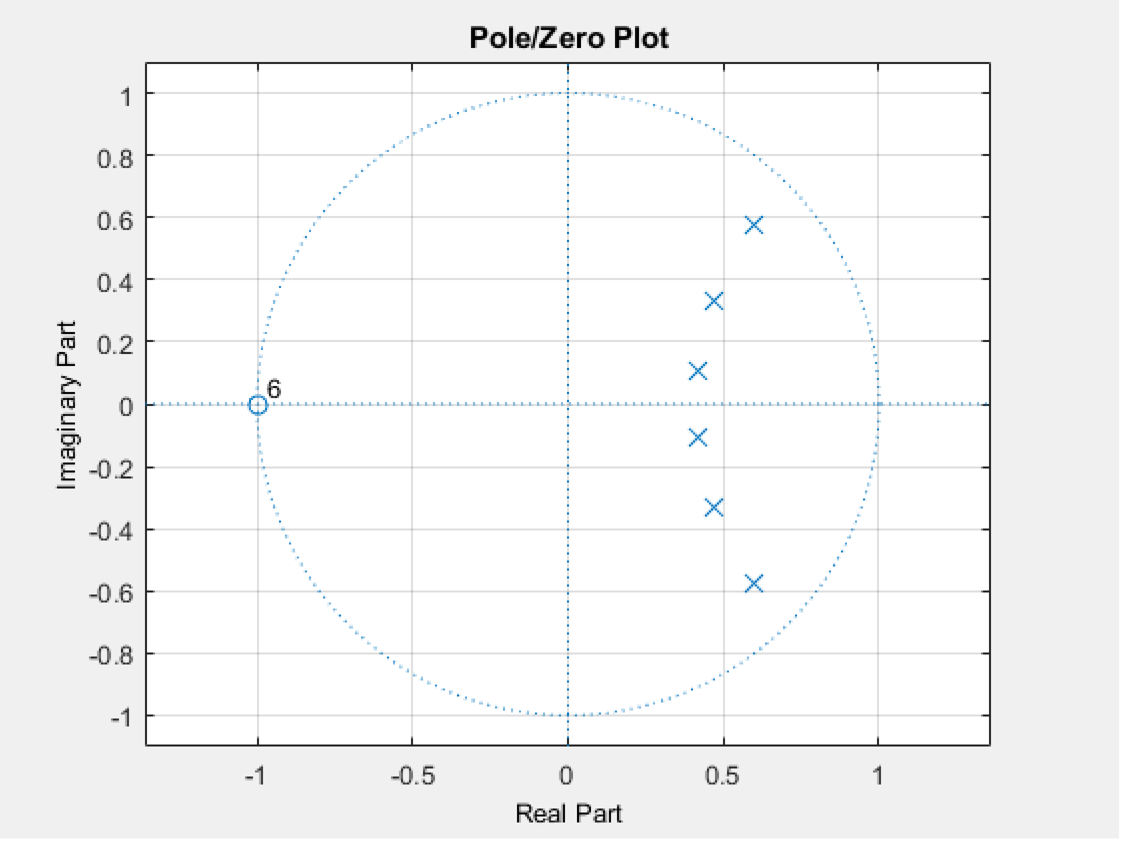
```
min_flag = logical
   1
```

Redesign the filter using `designfilt`. Check that the zeros and poles of the transfer function are on or within the unit circle.

```
d = designfilt('lowpassiir','DesignMethod','butter','FilterOrder',6, ...
               'HalfPowerFrequency',0.25);
d_flag = isminphase(d)
```

```
d_flag = logical
   1
```

```
zplane(d)
```



Given a filter defined with a set of single-precision numerator and denominator coefficients, check if it has minimum phase for different tolerance values.

```
b = single([1 1.00001]);
a = single([1 0.45]);
min_flag1 = isminphase(b,a)
```

```
min_flag1 = logical
   0
```

```
min_flag2 = isminphase(b,a,1e-3)
```

```
min_flag2 = logical
   1
```

See Also

`designfilt` | digitalFilter | `isallpass` | `islinphase` | `ismaxphase` | `isstable`

Introduced in R2013a

isreal

Determine whether filter uses real coefficients

Syntax

```
isreal(hd)
isreal(hs)
```

Description

`isreal(hd)` returns `1` (or true) if all filter coefficients for the filter `hd` are real, and returns `0` (or false) otherwise. Complex filters have one or more coefficients with nonzero imaginary parts.

`isreal(hs)` determines whether the filter coefficients of the filter System object `hs` are real, returning `1` if true and `0` if false.

Note Quantizing a filter cannot make a real filter into a complex filter.

Examples

Check If the Filter Coefficients are Real

Create a `dsp.BiquadFilter` System object™. Pass a fixed-point input to the object. Test the coefficients of the fixed-point filter to see if they are strictly real.

```
d = fdesign.lowpass('n,fp,ap,ast',5,0.4,0.5,20);
biquadFilter = design(d,'ellip','SystemObject',true);
IsRealBefore = isreal(biquadFilter)
```

```
IsRealBefore =

  logical
```

```
   1
```

Pass a fixed-point input to the object.

```
fiInput = fi(randn(1000,2),1,32,16);
fiOutput = biquadFilter(fiInput);
IsRealAfter = isreal(biquadFilter)
```

```
IsRealAfter =

  logical

   1
```

`isreal` returns a 1, indicating that the filter coefficients are real.

See Also

`isfir` | `islinphase` | `ismaxphase` | `isminphase` | `issos` | `isstable` | `isallpass`

Introduced in R2011a

issos

Determine whether filter is SOS form

Syntax

```
issos(hd)
issoss(hs)
```

Description

`issos(hd)` determines whether quantized filter `hq` consists of second-order sections. Returns `1` if all sections of quantized filter `hq` have order less than or equal to two, and `0` otherwise.

`issoss(hs)` determines whether the filter System object `hs` consists of second-order sections, returning `1` if true and `0` if false.

Examples

Design a Lowpass SOS Filter

By default, fdesign and design return SOS filters when possible. This example designs a lowpass SOS filter that uses fixed-point arithmetic.

```
d = fdesign.lowpass('n,fp,ap,ast',40,0.55,0.1,60);
hd = design(d,'ellip','SystemObject',true);
IsSOS=issos(hd)
```

```
IsSOS =

  logical

   1
```

The filter is in second-order section form.

See Also

`isallpass` | `isfir` | `islinphase` | `ismaxphase` | `isminphase` | `isreal` | `isstable`

Introduced in R2011a

isstable

Determine whether filter is stable

Syntax

```
flag = isstable(b,a)
flag = isstable(sos)
flag = isstable(d)
flag = isstable(hs)
flag = isstable(hs,'Arithmetic',arithtype)
```

Description

`flag = isstable(b,a)` returns a logical output, `flag`, equal to `true` if the filter specified by numerator coefficients, `b`, and denominator coefficients, `a`, is a stable filter. If the poles lie on or outside the circle, `isstable` returns `false`. If the poles are inside the circle, `isstable` returns `true`.

`flag = isstable(sos)` returns `true` if the filter specified by second order sections matrix, `sos`, is stable. `sos` is a *K*-by-6 matrix, where the number of sections, *K*, must be greater than or equal to 2. Each row of `sos` corresponds to the coefficients of a second order (biquad) filter. The *i*th row of the `sos` matrix corresponds to `[bi(1) bi(2) bi(3) ai(1) ai(2) ai(3)]`.

`flag = isstable(d)` returns `true` if the digital filter, `d`, is stable. Use `designfilt` to generate `d` based on frequency-response specifications.

`flag = isstable(hs)` returns `true` if the filter System object `hs` is stable. You must have the DSP System Toolbox software to use this syntax.

`flag = isstable(hs,'Arithmetic',`*`arithtype`*`)` analyzes the filter System object `hs` based on the specified *`arithtype`*. *`arithtype`* can be `'double'`, `'single'`, or `'fixed'`. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked. You must have the DSP System Toolbox software to use this syntax.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Examples

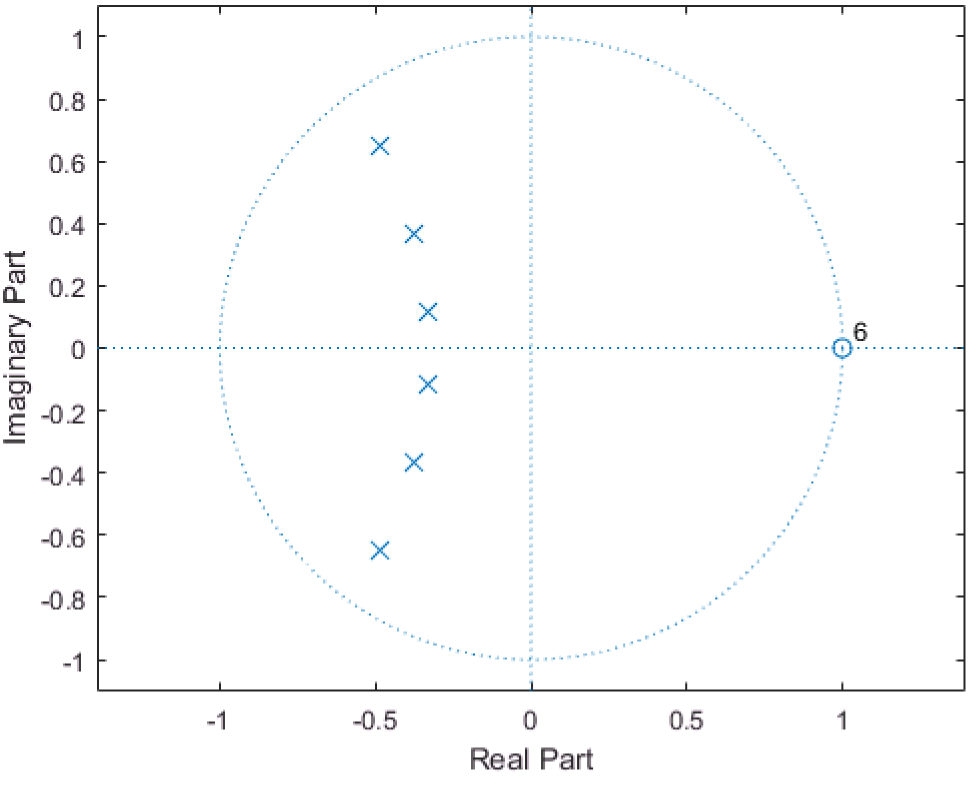
Filter Stability

Design a sixth-order Butterworth highpass IIR filter using second order sections. Specify a normalized 3-dB frequency of 0.7. Determine if the filter is stable.

```
[z,p,k] = butter(6,0.7,'high');
SOS = zp2sos(z,p,k);
flag = isstable(SOS)
```

```
flag = logical
   1
```

```
zplane(z,p)
```

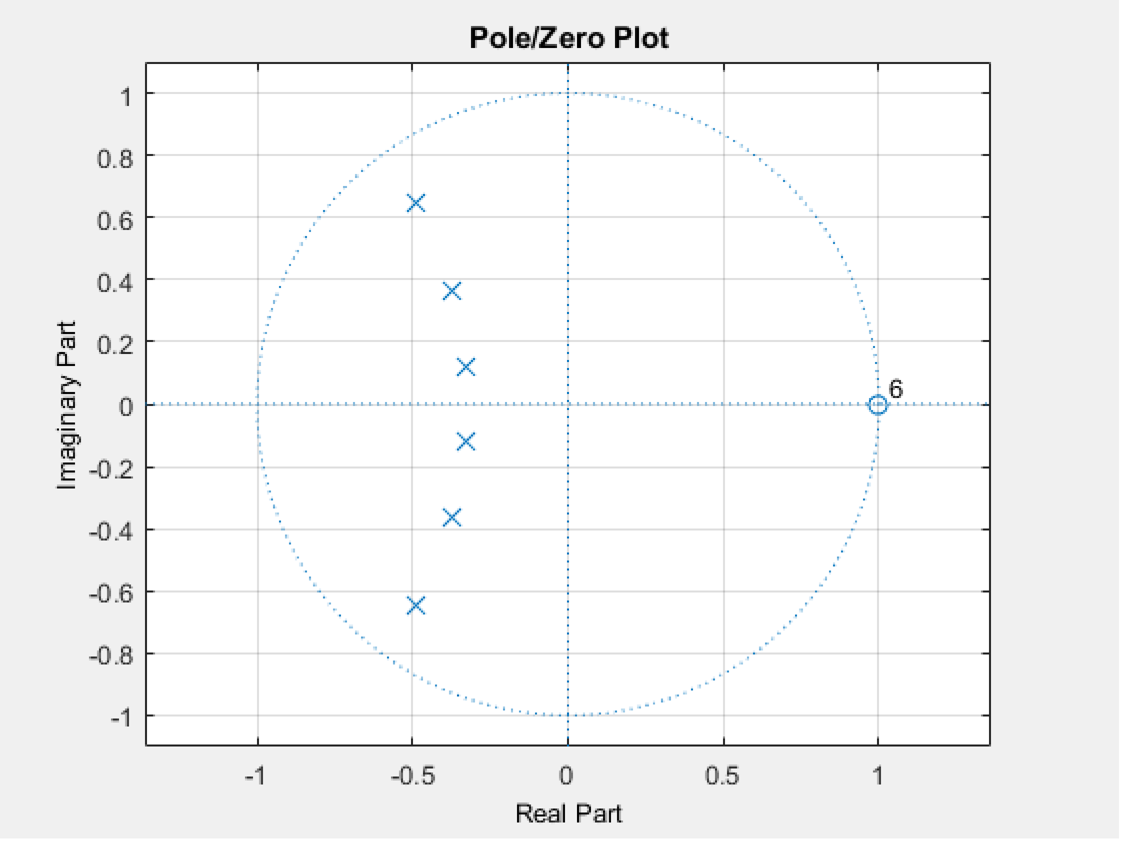


Redesign the filter using `designfilt` and check it for stability.

```
d = designfilt('highpassiir','DesignMethod','butter','FilterOrder',6, ...
                 'HalfPowerFrequency',0.7);
dflg = isstable(d)
```

```
dflg = logical
   1
```

```
zplane(d)
```



Create a filter and determine its stability at double and single precision.

```
b = [1 -0.5];
a = [1 -0.999999999];
act_flag1 = isstable(b,a)
```

```
act_flag1 = logical
   1
```

```
act_flag2 = isstable(single(b),single(a))
```

```
act_flag2 = logical
   0
```

See Also

designfilt | digitalFilter | isallpass | islinphase | ismaxphase | isminphase | zplane

Introduced in R2013a

kaiserwin

Kaiser window filter from specification object

Syntax

```
kFilter = design(d,'kaiserwin','SystemObject',true)
kFilter = design(d,'kaiserwin',designoption,value,designoption,...
value,'SystemObject',true)
```

Description

`kFilter = design(d,'kaiserwin','SystemObject',true)` designs a digital filter `kFilter` that uses a Kaiser window. For `kaiserwin` to work properly, the filter order in the specifications object must be even. In addition, higher order filters (filter order greater than 120) tend to be more accurate for smaller transition widths. `kaiserwin` returns a warning when your filter order may be too low to design your filter accurately.

`kFilter = design(d,'kaiserwin',designoption,value,designoption,... value,'SystemObject',true)` returns a filter where you specify design options as input arguments and the design process uses the Kaiser window technique.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `kaiserwin`, refer to the command line help system. For example, to get specific information about using `kaiserwin` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'kaiserwin')
```

Examples

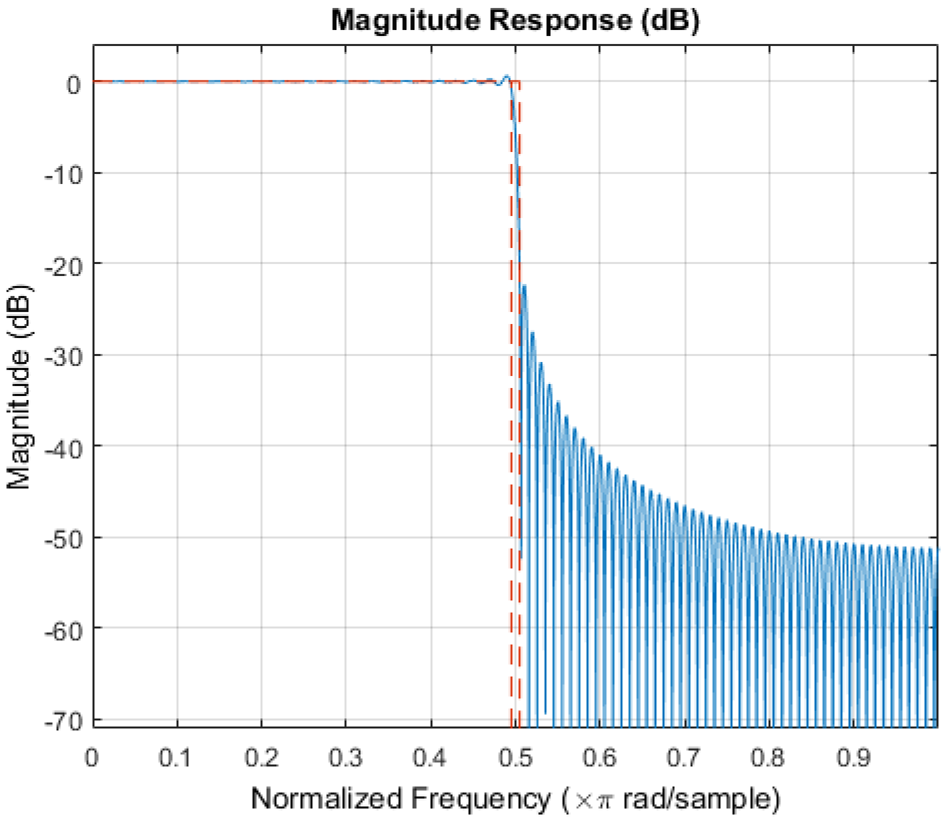
Design a Direct Form FIR Filter

This example designs a direct form FIR filter from a halfband filter specification object.

```
d = fdesign.halfband('n,tw',200,0.01);
hbFilter = design(d,'kaiserwin','filterstructure','dffir',...
    'SystemObject',true)
```

```
hbFilter = 

  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: [1×201 double]
    InitialConditions: 0

  Use get to show all properties
```

```
fvtool(hbFilter);
```



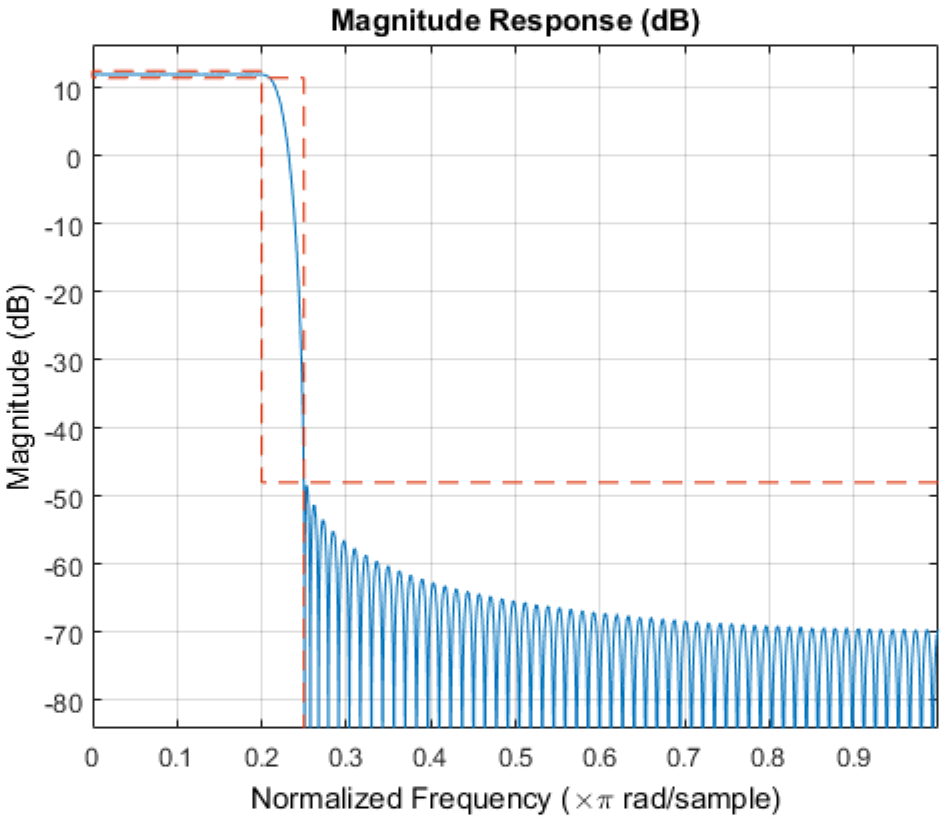
In this example, `kaiserwin` uses an interpolating filter specification object.

```
d = fdesign.interpolator(4,'lowpass');
interpFilter= design(d,'kaiserwin','SystemObject',true)
```

```
interpFilter = 

  dsp.FIRInterpolator with properties:

        NumeratorSource: 'Property'
              Numerator: [1×147 double]
    InterpolationFactor: 4
```

```
  Use get to show all properties
```

```
fvtool(interpFilter);
```



See Also

equiripple | firls

Introduced in R2011a

lagrange

Fractional delay filter from `fdesign.fracdelay` specification object

Syntax

```
Hd = design(d,'lagrange')
hd = design(d,'lagrange',FilterStructure,structure)
```

Description

`Hd = design(d,'lagrange')` designs a fractional delay filter using the Lagrange method based on the specifications in `d`.

`hd = design(d,'lagrange',FilterStructure,`*`structure`*`)` specifies the Lagrange design method and the *`structure`* filter structure for `hd`. The sole valid filter structure string for `structure` is `fd`, describing the fractional delay structure.

Examples

This example uses a fractional delay of 0.30 samples. The `help` and `designopts` commands provide the details about designing fractional delay filters.

```
d=fdesign.fracdelay(.30)

d =

               Response: 'Fractional Delay'
          Specification: 'N'
            Description: {'Filter Order'}
              FracDelay: 0.3
    NormalizedFrequency: true
            FilterOrder: 3

designmethods(d)

Design Methods for class fdesign.fracdelay (N):
```

```
lagrange

help(d,'lagrange')

DESIGN Design a Lagrange fractional delay filter.
HD = DESIGN(D, 'lagrange') designs a Lagrange filter specified by the
  FDESIGN object D, and returns the DFILT object HD.

HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a filter with the
 structure STRUCTURE.  STRUCTURE is 'farrowfd' by default and can be any of
 the following:

'farrowfd'
'fd'

 % Example #1 - Design a linear Lagrange fractional delay filter of 0.2 samples.
 h = fdesign.fracdelay(0.2,'N',2);
 Hd = design(h, 'lagrange', 'FilterStructure', 'farrowfd')

 % Example #2 - Design a cubic Lagrange fractional delay filter
 Fs = 8000;      % Sampling frequency of 8kHz
 fdelay = 50e-6; % Fractional delay of 50 microseconds.
 h  = fdesign.fracdelay(fdelay,'N',3,Fs);
 Hd = design(h, 'lagrange', 'FilterStructure', 'farrowfd');
```

This example designs a linear Lagrange fractional delay filter where you set the delay to 0.2 seconds and the filter order N to 2.

```
h = fdesign.fracdelay(0.2,'N',2);
hd = design(h,'lagrange','FilterStructure','farrowfd')
```

Design a cubic Lagrange fractional delay filter with filter order equal to 3.

```
Fs = 8000;      % Sampling frequency of 8 kHz.
fdelay = 50e-6; % Fractional delay of 50 microseconds.
h  = fdesign.fracdelay(fdelay,'N',3,Fs);
hd = design(h,'lagrange','FilterStructure','farrowfd');
```

References

Laakso, T. I., V. Välimäki, M. Karjalainen, and Unto K. Laine, "Splitting the Unit Delay - Tools for Fractional Delay Filter Design," *IEEE Signal Processing Magazine*, Vol. 13, No. 1, pp. 30-60, January 1996.

See Also

`design` | `designmethods` | `fdesign` | `designopts` | `fdesign.fracdelay`

Introduced in R2011a

liblinks

Check model for blocks from specific DSP System Toolbox libraries

Syntax

```
liblinks(lib)
liblinks(lib,sys)
liblinks(lib,sys,c)
```

Description

`liblinks(lib)` returns a cell array of character vectors that lists the blocks in the current model that are linked to the specified libraries. The input `lib` provides a cell array of character vectors with the library names. Use the library name visible in the title bar when you open a library model.

`liblinks(lib,sys)` acts on the named model `sys`.

`liblinks(lib,sys,c)` changes the foreground color of the returned blocks to the color `c`. Possible values of `c` are `'blue'`, `'green'`, `'red'`, `'cyan'`, `'magenta'`, `'yellow'`, or `'black'`.

Examples

Check for blocks from the Sources library in the specified model:

```
rlsdemo
liblinks('dspsrcs4',gcs)
```

See Also

`dsp_links`

Introduced before R2006a

limitcycle

Response of single-rate, fixed-point IIR filter

Syntax

```
report = limitcycle(hd)
report = limitcycle(hd,ntrials,inputlengthfactor,stopcriterion)
```

Description

`report = limitcycle(hd)` returns the structure `report` that contains information about how filter `hd` responds to a zero-valued input vector. By default, the input vector has length equal to twice the impulse response length of the filter.

`limitcycle` returns a structure whose elements contain the details about the limit cycle testing. As shown in this table, the report includes the following details.

Output Object Property	Description
`LimitCycleType`	Contains one of the following results: • `Granular` — indicates that a granular overflow occurred. • `Overflow` — indicates that an overflow limit cycle occurred. • `None` — indicates that the test did not find any limit cycles.
`Zi`	Contains the initial condition value(s) that caused the detected limit cycle to occur.
`Output`	Contains the output of the filter in the steady state.
`Trial`	Returns the number of the Monte Carlo trial on which the limit cycle testing stopped. For example, `Trial = 10` indicates that testing stopped on the tenth Monte Carlo trial.

Using an input vector longer than the filter impulse response ensures that the filter is in steady-state operation during the limit cycle testing. `limitcycle` ignores output that occurs before the filter reaches the steady state. For example, if the filter impulse length is 500 samples, `limitcycle` ignores the filter output from the first 500 input samples.

To perform limit cycle testing on your IIR filter, you must set the filter `Arithmetic` property to `fixed` and `hd` must be a fixed-point IIR filter of one of the following forms:

- `df1` — direct-form I
- `df1t` — direct-form I transposed
- `df1sos` — direct-form I with second-order sections
- `df1tsos` — direct-form I transposed with second-order sections
- `df2` — direct-form II
- `df2t` — direct-form II transposed
- `df2sos` — direct-form II with second-order sections
- `df2tsos` — direct-form II transposed with second-order sections

When you use `limitcycle` without optional input arguments, the default settings are

- Run 20 Monte Carlo trials
- Use an input vector twice the length of the filter impulse response
- Stop testing if the simulation process encounters either a granular or overflow limit cycle

To determine the length of the filter impulse response, use `impzlength`:

```
impzlength(hd)
```

During limit cycle testing, if the simulation runs reveal both overflow and granular limit cycles, the overflow limit cycle takes precedence and is the limit cycle that appears in the report.

Each time you run `limitcycle`, it uses a different sequence of random initial conditions, so the results can differ from run to run.

Each Monte Carlo trial uses a new set of randomly determined initial states for the filter. Test processing stops when `limitcycle` detects a zero-input limit cycle in filter `hd`.

`report = limitcycle(hd,ntrials,inputlengthfactor,stopcriterion)` returns a report of how filter `hd` responds to a zero-valued input vector, using the following optional input arguments:

- `ntrials` — Number of Monte Carlo trials (default is 20).
- `inputlengthfactor` — integer factor used to calculate the length of the input vector. The length of the input vector comes from (`impzlength(hd) * inputlengthfactor`), where `inputlengthfactor` = 2 is the default value.
- `stopcriterion` — the criterion for stopping the Monte Carlo trial processing. `stopcriterion` can be set to **`either`** (the default), **`granular`**, **`overflow`**. This table describes the results of each stop criterion.

stopcriterion Setting	Description
`either`	Stop the Monte Carlo trials when `limitcycle` detects either a granular or overflow limit cycle.
`granular`	Stop the Monte Carlo trials when `limitcycle` detects a granular limit cycle.
`overflow`	Stop the Monte Carlo trials when `limitcycle` detects an overflow limit cycle.

Note An important feature is that if you specify a specific limit cycle stop criterion, such as `overflow`, the Monte Carlo trials do not stop when testing encounters a granular limit cycle. You receive a warning that no `overflow` limit cycle occurred, but consider that a `granular` limit cycle might have occurred.

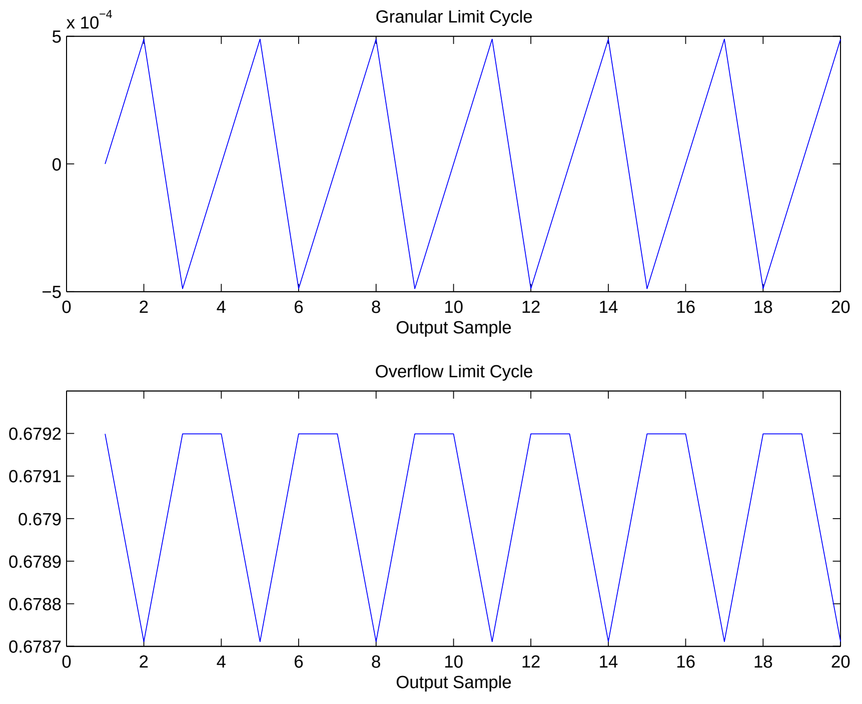
Examples

In this example, there is a region of initial conditions in which no limit cycles occur and a region where they do. If no limit cycles are detected before the Monte Carlo trials are over, the state sequence converges to zero. When a limit cycle is found, the states do not end at zero. Each time you run this example, it uses a different sequence of random initial conditions, so the plot you get can differ from the one displayed in the following figure.

```
s = [1 0 0 1 0.9606 0.9849];
hd = dfilt.df2sos(s);
hd.arithmetic = 'fixed';
```

```
greport = limitcycle(hd,20,2,'granular')
oreport = limitcycle(hd,20,2,'overflow')
figure,
subplot(211),plot(greport.Output(1:20)), title('Granular Limit Cycle');
subplot(212),plot(oreport.Output(1:20)), title('Overflow Limit Cycle');

greport =

    LimitCycle: 'granular'
            Zi: [2x1 double]
        Output: [1303x1 embedded.fi]
         Trial: 1

oreport =

    LimitCycle: 'overflow'
            Zi: [2x1 double]
        Output: [1303x1 embedded.fi]
         Trial: 2
```

The plots shown in this figure present both limit cycle types — the first displays the small amplitude granular limit cycle, the second the larger amplitude overflow limit cycle.



As you see from the plots, and as is generally true, overflow limit cycles are much greater magnitude than granular limit cycles. This is why `limitcycle` favors overflow limit cycle detection and reporting.

See Also

`freqz` | `noisepsd`

Introduced in R2011a

maxflat

Maxflat FIR filter

Syntax

```
maxflatFilter = design(d,'maxflat','SystemObject',true)
maxflatFilter =
design(d,'maxflat','FilterStructure',structure,'SystemObject',true)
```

Description

`maxflatFilter = design(d,'maxflat','SystemObject',true)` designs a maximally flat filter, `maxflatFilter`, from a filter specification object, `d`.

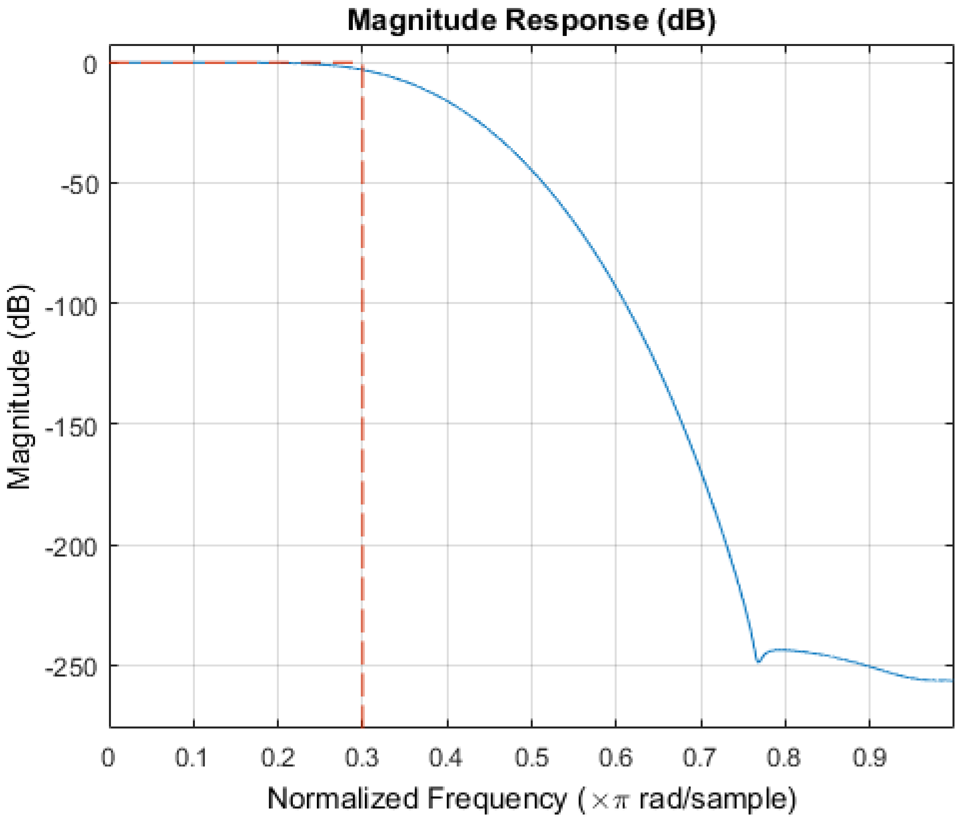
`maxflatFilter =`
`design(d,'maxflat','FilterStructure',structure,'SystemObject',true)` designs a maximally flat filter where `structure` is one of the following:

- `'dffir'`, a discrete-time, direct-form FIR filter (the default value)
- `'dffirt'`, a discrete-time, direct-form FIR transposed filter
- `'dfsymfir'`, a discrete-time, direct-form symmetric FIR filter

Examples

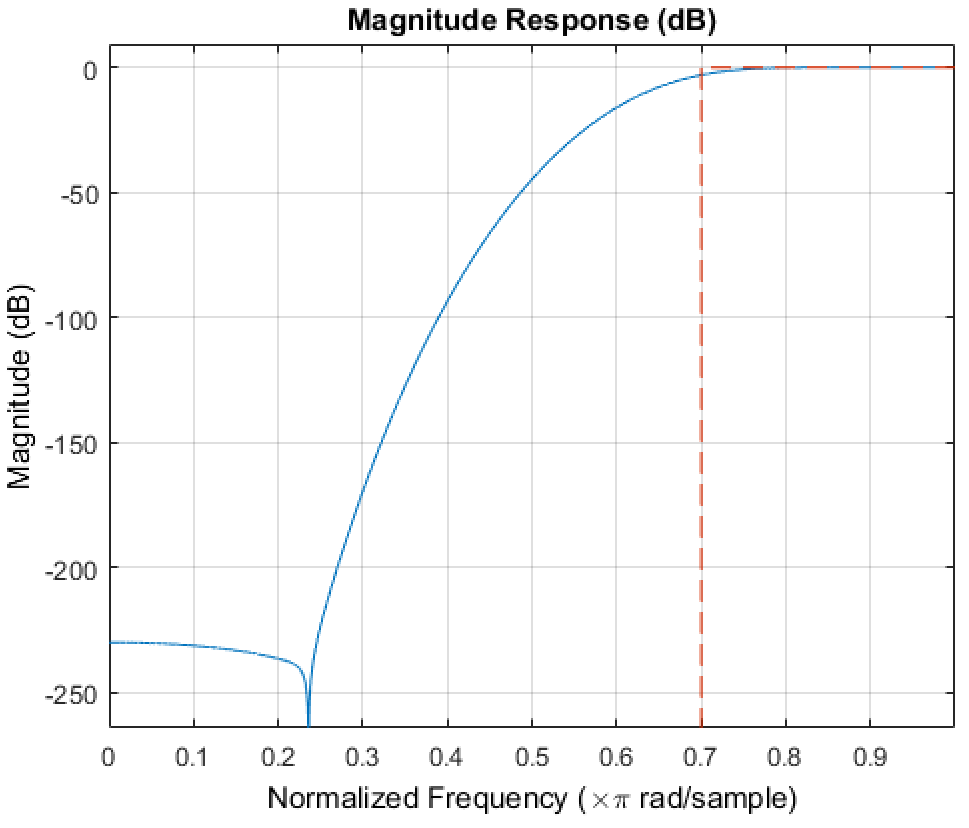
Design a Lowpass Filter with a Maximally Flat FIR Structure

```
d  = fdesign.lowpass('N,F3dB', 50, 0.3);
flatLowpass = design(d, 'maxflat','SystemObject',true);
fvtool(flatLowpass);
```



Design a Highpass Filter With a Maximally Flat Symmetric FIR Structure

```
d  = fdesign.highpass('N,F3dB', 50, 0.7);
flatHighpass = design(d,'maxflat','FilterStructure','dfsymfir',...
    'SystemObject',true);
fvtool(flatHighpass)
```



Introduced in R2011a

maximizestopband

Maximize stopband attenuation of fixed-point filter

Syntax

```
Hq = maximizestopband(Hd,Wordlength)
Hq = maximizestopband(Hd,Wordlength,'Ntrials',N)
```

Description

`Hq = maximizestopband(Hd,Wordlength)` quantizes the single-stage or multistage FIR filter `Hd` and returns the fixed-point filter `Hq` with wordlength `wordlength` that maximizes the stopband attenuation. `Hd` must be generated using `fdesign` and `design`. For multistage filters, `wordlength` can either be a scalar or vector. If `wordlength` is a scalar, the same wordlength is used for all stages. If `wordlength` is a vector, each stage uses the corresponding element in the vector. The vector length must equal the number of stages. `maximizestopband` uses a stochastic noise-shaping procedure by default to minimize the wordlength. To obtain repeatable results on successive function calls, initialize the uniform random number generator `rand`

`Hq = maximizestopband(Hd,Wordlength,'Ntrials',N)` specifies the number of Monte Carlo trials to use in the maximization. `Hq` is the fixed-point filter with the largest stopband attenuation among the trials. The number of Monte Carlo trials `N` defaults to 1.

You must have the Fixed-Point Designer software installed to use this function.

Examples

Maximize Stopband Attenuation

Maximize stopband attenuation for 16-bit fixed-point filter.

```
Hf = fdesign.lowpass('Fp,Fst,Ap,Ast',0.4,0.45,0.5,60);
```

```
Hd = design(Hf,'equiripple');
```

Use 16 bits to represent coefficients.

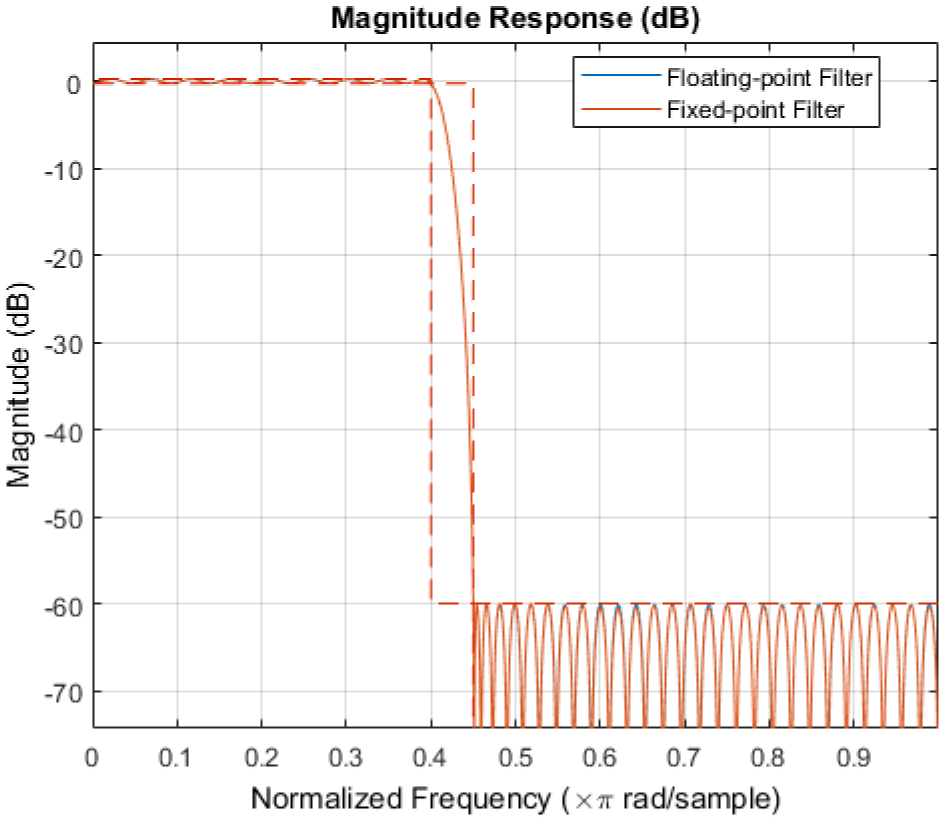
```
WL = 16;
Hq = maximizestopband(Hd,WL);
```

Compare stopband attenuation

```
md = measure(Hd)


md =

Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.4
3-dB Point       : 0.41178
6-dB Point       : 0.41845
Stopband Edge    : 0.45
Passband Ripple  : 0.49369 dB
Stopband Atten.  : 60.0697 dB
Transition Width : 0.05
```

```
mq = measure(Hq)


mq =

Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.4
3-dB Point       : 0.41178
6-dB Point       : 0.41845
Stopband Edge    : 0.45
Passband Ripple  : 0.49773 dB
Stopband Atten.  : 59.9728 dB
Transition Width : 0.05
```

```
hfvt=fvtool(Hd,Hq,'ShowReference','off');
legend(hfvt,'Floating-point Filter','Fixed-point Filter');
```



Tutorials

- "Fixed-Point Data Types"

See Also

`constraincoeffwl` | `measure` | `rand` | `design` | `fdesign` | `minimizecoeffwl`

Topics

"Fixed-Point Data Types"

Introduced in R2011a

maxstep

Maximum step size for adaptive filter convergence

Compatibility

`maxstep` will be removed in a future release. Use dsp.LMSFilter or dsp.BlockLmsFilter instead.

Syntax

```
mumax = maxstep(ha,x)
[mumax,mumaxmse] = maxstep(ha,x)
```

Description

`mumax = maxstep(ha,x)` predicts a bound on the step size to provide convergence of the mean values of the adaptive filter coefficients. The columns of the matrix `x` contain individual input signal sequences. The signal set is assumed to have zero mean or nearly so.

`[mumax,mumaxmse] = maxstep(ha,x)` predicts a bound on the adaptive filter step size to provide convergence of the LMS adaptive filter coefficients in the mean-square sense. `maxstep` issues a warning when `ha.stepsize` is outside of the range 0 < `ha.stepsize` < `mumaxmse`/2.

`maxstep` is available for the following adaptive filter objects:

- `adaptfilt.blms`
- `adaptfilt.blmsfft`
- `adaptfilt.lms`
- `adaptfilt.nlms` (uses a different syntax. Refer to the text below.)
- `adaptfilt.se`

Note With `adaptfilt.nlms` filter objects, `maxstep` uses the following slightly different syntax:

```
mumax = maxstep(ha)
[mumax,mumaxmse] = maxstep(ha)
```

The maximum step size for convergence is fully defined by the filter object `ha`. Matrix `x` is not necessary. If you include an `x` input matrix, MATLAB returns an error.

Examples

Analyze and simulate a 32-coefficient (31st-order) LMS adaptive filter object. To demonstrate the adaptation process, run 2000 iterations and 50 trials.

```
% Specify [numiterations,numexamples] = size(x);
x = zeros(2000,50);
d = x;
obj = fdesign.lowpass('n,fc',31,0.5);
hd = design(obj,'window'); % FIR filter to identified.
coef = cell2mat(hd.coefficients); % Convert cell array to matrix.

for k=1:size(x,2);   % Create input and desired response signal
                     % matrices.
% Set the (k)th input to the filter.
  x(:,k) = filter(sqrt(0.75),[1 -0.5],sign(randn(size(x,1),1)));
  n = 0.1*randn(size(x,1),1);  % (k)th observation noise signal.
  d(:,k) = filter(coef,1,x(:,k))+n; % (k)th desired signal end.
end
mu = 0.1;                      % LMS step size.
ha = adaptfilt.lms(32,mu);
[mumax,mumaxmse] = maxstep(ha,x);

Warning: Step size is not in the range 0 < mu < mumaxmse/2:
Erratic behavior might result.

mumax

mumax =

    0.0623

mumaxmse

mumaxmse =

    0.0530
```

See Also

`msepred` | `msesim` | `filter`

Introduced in R2011a

measure

Measure frequency response characteristics of filter System object

Syntax

```
measure(sysobj)
M = measure(sysobj)
M = measure(sysobj,'Arithmetic',ARITH,...)
M = measure(sysobj,'freqspec', freqspecvalue,...)
```

Description

`measure(sysobj)` displays measurements of various quantities from the frequency response of the filter System object, `sysobj`. Measurements include the actual passband ripple, the minimum stopband attenuation, the frequency point at which the filter's gain is 3 dB below the nominal passband gain, etc. You must construct `sysobj` using `fdesign` and `design` with the name-value pair argument `'SystemObject', true`. You can optionally specify additional options by one or more `Name,Value` pair arguments.

`M = measure(sysobj)` returns the measurements, *M*, such that the measurements can be queried programmatically. For example, to query the 3 dB point, type `M.F3dB`. Type `get(M)` to see the full list of properties that can be queried. Note that different filter responses generate different measurements.

`M = measure(sysobj,'Arithmetic',ARITH,...)` analyzes the filter System object, `sysobj`, based on the arithmetic specified in the `ARITH` input. `ARITH` can be set to one of `'double'`, `'single'`, or `'fixed'`. When the arithmetic input is not specified and the filter System object is in an unlocked state, the analysis tool assumes a double precision filter.

`M = measure(sysobj,'freqspec', freqspecvalue,...)` passes the frequency value as an input to `measure` in order to determine the corresponding magnitude measurements. For designs that do not specify some of the frequency constraints, you can determine the corresponding magnitude measurements using this option.

In the following example, the passband edge, passband ripple, and the transition width of the IIR filter are unknown.

```
designLowpass = fdesign.lowpass('N,F3dB,Ast',8,0.5,80);
chebFilter = design(designLowpass,'cheby2');
measure(chebFilter)
```

```
Sample Rate      : N/A (normalized frequency)
Passband Edge    : Unknown
3-dB Point       : 0.5
6-dB Point       : 0.51823
Stopband Edge    : 0.68727
Passband Ripple  : Unknown
Stopband Atten.  : 79.9994 dB
Transition Width : Unknown
```

Specify the passband edge to be `0.4`, and measure the passband ripple and the transition width of this filter.

```
measure(chebFilter,'Fpass',0.4)
```

```
Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.4
3-dB Point       : 0.5
6-dB Point       : 0.51823
Stopband Edge    : 0.68727
Passband Ripple  : 0.013644 dB
Stopband Atten.  : 79.9994 dB
Transition Width : 0.28727
```

When `sysobj` is a generic discrete-time filter, for example, a single-rate lowpass filter, `measure(sysobj)` returns the following filter specifications.

Lowpass Filter Specification	Description
`Sample Rate`	Filter sampling frequency.
`Passband Edge`	Location of the edge of the passband as it enters transition.
`3-dB Point`	Location of the –3 dB point on the response curve.
`6-dB Point`	Location of the –6 dB point on the response curve.
`Stopband Edge`	Location of the edge of the transition band as it enters the stopband.
`Passband Ripple`	Ripple in the passband.

Lowpass Filter Specification	Description
`Stopband Atten`	Attenuation in the stopband.
`Transition Width`	Width of the transition between the passband and stopband, in normalized frequency or absolute frequency. Measured between `Fpass` and `Fstop`.

When `sysobj` is a bandstop filter, `measure(sysobj)` returns these specifications for the resulting bandstop filter.

Bandstop Filter Specification	Description
`Sample Rate`	Filter sampling frequency.
`First Passband Edge`	Location of the edge of the first passband.
`First 3-dB Point`	Location of the edge of the –3 dB point in the first transition band.
`First 6-dB Point`	Location of the edge of the –6 dB point in the first transition band.
`First Stopband Edge`	Location of the start of the stopband.
`Second Stopband Edge`	Location of the end of the stopband.
`Second 6-dB Point`	Location of the edge of the –6 dB point in the second transition band.
`Second 3-dB Point`	Location of the edge of the –3 dB point in the second transition band.
`Second Passband Edge`	Location of the start of the second passband.
`First Passband Ripple`	Ripple in the first passband.
`Stopband Atten`	Attenuation in the stopband.
`Second Passband Ripple`	Ripple in the second passband.
`First Transition Width`	Width of the first transition region. Measured between the –3 and –6 dB points.
`Second Transition Width`	Width of the second transition region. Measured between the –6 and –3 dB points.

When `sysobj` is an interpolator, decimator, or a rate converter, `measure(sysobj)` returns these specifications for the resulting filter.

Interpolator Filter Specification	Description
`Sample Rate`	Filter sampling frequency.
`First Passband Edge`	Location of the edge of the passband as it enters transition.
`3-dB Point`	Location of the –3 dB point on the response curve.
`6-dB Point`	Location of the –6 dB point on the response curve.
`Stopband Edge`	Location of the edge of the transition band as it enters the stopband.
`Passband Ripple`	Ripple in the passband.
`Stopband Atten`	Attenuation in the stopband.
`Transition Width`	Width of the transition between the passband and stopband, in normalized frequency or absolute frequency. Measured between `Fpass` and `Fstop`.

Input Arguments

`sysobj`

Filter System object.

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpassFilter
dsp.AllpoleFilter
dsp.BiquadFilter
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.CICDecimator
dsp.CICInterpolator
dsp.CoupledAllpassFilter

Filter System objects
dsp.FarrowRateConverter
dsp.FilterCascade
dsp.FIRDecimator
dsp.FIRFilter
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.FIRInterpolator
dsp.FIRRateConverter
dsp.HighpassFilter
dsp.IIRFilter
dsp.IIRHalfbandDecimator
dsp.IIRHalfbandInterpolator
dsp.LowpassFilter
dsp.NotchPeakFilter
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Arithmetic'` — Value types:
`‘double’ | 'single' | 'fixed'`

Specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	'Same as input'	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Examples

Measure the Filter Specifications

Create a lowpass filter and check whether the actual filter meets the specifications. For this case, use normalized frequency for Fs, the default setting.

```
desigLowpass = fdesign.lowpass('Fp,Fst,Ap,Ast',0.45,0.55,0.1,80)
```

```
desigLowpass = 

  lowpass with properties:

               Response: 'Lowpass'
          Specification: 'Fp,Fst,Ap,Ast'
            Description: {4×1 cell}
    NormalizedFrequency: 1
                  Fpass: 0.4500
                  Fstop: 0.5500
                  Apass: 0.1000
                  Astop: 80
```

```
designmethods(desigLowpass,'SystemObject',true)
```

```
Design Methods that support System objects for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage
```

Use the default equiripple design method.

```
equiFilter = design(desigLowpass,'SystemObject',true)
```

```
equiFilter =

  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: [1×70 double]
    InitialConditions: 0

  Use get to show all properties
```

Measure the specifications of the designed lowpass filter.

```
measure(equiFilter)
```

```
ans =

Sample Rate      : N/A (normalized frequency)
Passband Edge    : 0.45
3-dB Point       : 0.47798
6-dB Point       : 0.48913
Stopband Edge    : 0.55
Passband Ripple  : 0.095021 dB
Stopband Atten.  : 80.1164 dB
Transition Width : 0.1
```

`Stopband Edge`, `Passband Edge`, `Passband Ripple`, and `Stopband Atten.` all meet the specifications.

Now, using `Fs` in linear frequency, create a bandpass filter, and measure the magnitude response characteristics.

```
designBandpass = fdesign.bandpass
```

```
designBandpass =

  bandpass with properties:

               Response: 'Bandpass'
          Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
            Description: {7×1 cell}
```

```
    NormalizedFrequency: 1
                 Fstop1: 0.3500
                 Fpass1: 0.4500
                 Fpass2: 0.5500
                 Fstop2: 0.6500
                 Astop1: 60
                  Apass: 1
                 Astop2: 60
```

Convert to Linear Frequency.

```
normalizefreq(designBandpass,false,1.5e3)
```

```
bpFilter = design(designBandpass,'cheby2','SystemObject',true);
```

Measure the specifications of the designed bandpass filter.

```
measure(bpFilter)
```

```
ans =

Sample Rate              : 1.5 kHz
First Stopband Edge      : 262.5 Hz
First 6-dB Point         : 319.9585 Hz
First 3-dB Point         : 324.9744 Hz
First Passband Edge      : 337.5 Hz
Second Passband Edge     : 412.5 Hz
Second 3-dB Point        : 425.0256 Hz
Second 6-dB Point        : 430.0415 Hz
Second Stopband Edge     : 487.5 Hz
First Stopband Atten.    : 60 dB
Passband Ripple          : 0.17985 dB
Second Stopband Atten.   : 60 dB
First Transition Width   : 75 Hz
Second Transition Width  : 75 Hz
```

Tips

For designs that do not specify some of the frequency constraints, the function may not be able to determine corresponding magnitude measurements. In these cases, a constraint can be passed in to `measure` to determine such measurements. For example:

```
f = fdesign.lowpass('N,F3dB,Ast',8,0.5,80);
H = design(f,'cheby2','SystemObject',true);
measure(H)
```

returns values of `Unknown` for the passband edge, passband ripple, and transition width measurements, but

```
f = fdesign.lowpass('N,F3dB,Ast',8,0.5,80);
H = design(f,'cheby2','SystemObject',true);
measure(H,'Fpass',0.4)
```

provides measurements for all returned values.

See Also

`design` | `fdesign` | `normalizefreq`

Introduced in R2011a

mfilt

Multirate filter

Compatibility

`mfilt` will be removed in a future release. See dsp.CICDecimator, dsp.CICInterpolator, dsp.FIRDecimator, dsp.FIRInterpolator, dsp.FilterCascade, dsp.FarrowRateConverter, dsp.FIRRateConverter, dsp.IIRHalfbandDecimator, or dsp.IIRHalfbandInterpolator instead.

Syntax

`hm = mfilt.`*`structure`*`(input1,input2,...)`

Description

`hm = mfilt.`*`structure`*`(input1,input2,...)` returns the object `hm` of type *`structure`*. As with `dfilt` objects, you must include the *`structure`* string to construct a multirate filter object. You can, however, construct a default multirate filter object of a given structure by not including input arguments in your calling syntax.

Multirate filters include decimators and interpolators, and fractional decimators and fractional interpolators where the resulting interpolation or decimation factor is not an integer.

Structures

Each of the following multirate filter structures has a reference page of its own.

Filter Structure String	Description of Resulting Multirate Filter	Coefficient Mapping Support in `realizemdl` and `block`
`mfilt.cascade`	Cascade multirate filters to form another filter	Supported

Filter Structure String	Description of Resulting Multirate Filter	Coefficient Mapping Support in `realizemdl` and `block`
`mfilt.cicdecim`	Cascaded integrator-comb decimator	Not supported
`mfilt.cicinterp`	Cascaded integrator-comb interpolator	Not supported
`mfilt.farrowsrc`	Multirate Farrow filter	Supported.
`mfilt.fftfirinterp`	Overlap-add FIR polyphase interpolator	Not supported
`mfilt.firdecim`	Direct-form FIR polyphase decimator	Supported
`mfilt.firinterp`	Direct-form FIR polyphase interpolator	Supported
`mfilt.firsrc`	Direct-form FIR polyphase sample rate converter	Supported
`mfilt.firtdecim`	Direct-form transposed FIR polyphase decimator	Supported
`mfilt.holdinterp`	FIR hold interpolator	Not supported
`mfilt.iirdecim`	IIR decimator	Supported
`mfilt.iirinterp`	IIR interpolator	Supported
`mfilt.linearinterp`	FIR Linear interpolator	Supported
`mfilt.iirwdfdecim`	IIR wave digital filter decimator	Supported
`mfilt.iirwdfinterp`	IIR wave digital filter interpolator	Supported

Copying mfilt Objects

To create a copy of an `mfilt` object, use the `copy` method.

```
h2 = copy(hd)
```

Note The syntax `hd2 = hd` copies only the object handle. It does not create a new object. `hd2` and `hd` are not independent. If you change the property value for one of the two, such as `hd2`, you are changing the property for both.

Examples

Decimation by a factor of two. Convert input sampled at 48 kHz to 24 kHz:

```
Fs = 4.8e4;
t =0:1/Fs:1-(1/Fs);
x = cos(2*pi*4000*t);
Hm=mfilt.firdecim(2);
% Note cutoff frequency of 1/2 normalized frequency
fvtool(Hm);
% Note the group delay of 34 samples
fvtool(Hm,'analysis','grpdelay');
y = filter(Hm,x);
% Note delay in output is consistent with 36/2
stem(y(1:48),'markerfacecolor',[0 0 1]);
```

Using existing coefficients to decimate a signal by a factor of two:

```
M = 2; % Decimation factor
  b = firhalfband('minorder',.45,0.0001);
  Hm = mfilt.firdecim(M,b);
  % Decimate a signal which consists of the sum of 2 sinusoids.
  N = 160;
  x = sin(2*pi*.05*[0:N-1]+pi/3)+cos(2*pi*.03*[0:N-1]+pi/3);
  y = filter(Hm,x);
```

Note Multirate filters can also have complex coefficients. For example, you can specify complex coefficients in the argument `num` passed to the filter structure. This works for all multirate filter structures.

```
m = 2;
num = [0.5 0.5+1j*0.2];
Hm = mfilt.firdecim(m, num);
y = filter(Hm, [1:10]);
```

See Also

`mfilt.firinterp` | `mfilt.firsrc` | `mfilt.firtdecim`

Introduced in R2011a

mfilt.cascade

Cascade filter objects

Compatibility

`mfilt.cascade` will be removed in a future release. Use dsp.FilterCascade instead.

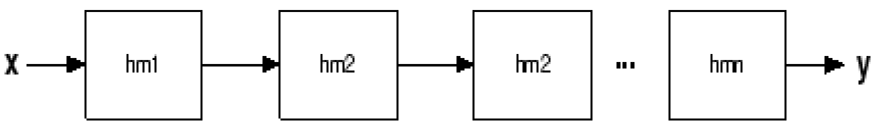
Syntax

```
hm = cascade(hm1,hm2,...,hmn)
```

Description

`hm = cascade(hm1,hm2,...,hmn)` creates filter object `hm` by cascading (connecting in series) the individual filter objects `hm1`, `hm2`, and so on to `hmn`.

In block diagram form, the cascade looks like this, with x as the input to the filter `hm` and y the output from the cascade filter `hm`:



`mfilt.cascade` accepts any combination of `mfilt` and `dfilt` objects (discrete time filters) to cascade, as well as Farrow filter objects.

Examples

Create a variety of `mfilt` objects and cascade them together.

```
hm(1) = mfilt.firdecim(12);
hm(2) = mfilt.firdecim(4);
h1 = mfilt.cascade(hm(1),hm(2));
hm(3) = mfilt.firinterp(4);
```

```
hm(4) = mfilt.firinterp(12);
h2 = mfilt.cascade(hm(3),hm(4));
% Cascade h1 and h2 together
h3 = mfilt.cascade(h1,h2,9600);
```

See Also

`dfilt.cascade`

Introduced in R2011a

mfilt.cicdecim

Fixed-point CIC decimator

Compatibility

`mfilt.cicdecim` will be removed in a future release. Use dsp.CICDecimator instead.

Syntax

```
hm = mfilt.cicdecim(r,m,n,iwl,owl,wlps)
```

Description

`hm = mfilt.cicdecim(r,m,n,iwl,owl,wlps)` returns a cascaded integrator-comb (CIC) decimation filter object.

All of the input arguments are optional. To enter any optional value, you must include all optional values to the left of your desired value.

When you omit one or more input options, the omitted option applies the default values shown in the table below.

The following table describes the input arguments for creating `hm`.

Input Arguments	Description
`r`	Decimation factor applied to the input signal. Sharpens the response curve to let you change the shape of the response. `r` must be an integer value greater than or equal to 1. The default value is 2.
`m`	Differential delay. Changes the shape, number, and location of nulls in the filter response. Increasing `m` increases the sharpness of the nulls and the response between nulls. In practice, differential delay values of 1 or 2 are the most common. `m` must be an integer value greater than or equal to 1. The default value is 1.

Input Arguments	Description
`n`	Number of sections. Deepens the nulls in the response curve. Note that this is the number of either comb or integrator sections, not the total section count. 2 is the default value.
`iwl`	Word length of the input signal. Use any integer number of bits. The default value is 16 bits.
`owl`	Word length of the output signal. It can be any positive integer number of bits. By default, `owl` is 16 bits.
`wlps`	Defines the number of bits per word in each filter section while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter `wlps` as a scalar or vector of length `2*n`, where `n` is the number of sections. When `wlps` is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the decimator. When you elect to specify `wlps` as an input argument, the `FilterInternals` property automatically switches from the default value of `'FullPrecision'` to `'SpecifyWordLengths'`.

Constraints and Word Length Considerations

CIC decimators have the following constraint — the word lengths of the filter section must be monotonically decreasing. The word length of each filter section must be the same size as, or smaller than, the word length of the previous filter section.

The formula for B_{max}, the most significant bit at the filter output, is given in the Hogenauer paper in the References below.

$$B_{\max} = (N \log_2 RM + B_{in} - 1)$$

where B_{in} is the number of bits of the input.

The cast operations shown in the diagram in “Algorithms” on page 5-1124 perform the changes between the word lengths of each section. When you specify word lengths that do not follow the constraints above, the constructor returns an error.

When you specify the word lengths correctly, the most significant bit B_{max} stays the same throughout the filter, while the word length of each section either decreases or stays the same. This can cause the fraction length to change throughout the filter as least significant bits are truncated to decrease the word length, as shown in “Algorithms” on page 5-1124.

Properties of the Object

Objects have properties that control the way the object behaves. This table lists all the properties for the filter, with a description of each.

Name	Values	Default	Description
`Arithmetic`	`fixed`	`fixed`	Reports the kind of arithmetic the filter uses. CIC decimators are always fixed-point filters.
`DecimationFactor`	Any positive integer	2	Amount to reduce the input sampling rate.
`DifferentialDelay`	Any positive integer	1	Sets the differential delay for the filter. Usually a value of one or two is appropriate.
`FilterStructure`	`mfilt` structure string	None	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` objects.
`FilterInternals`	`FullPrecision`, `MinWordLengths`, `SpecifyPrecision`, `SpecifyWordLengths`	`FullPrecision`	Set the usage mode for the filter. Refer to “Usage Modes” on page 5-1117 below for details.
`InputFracLength`	Any positive integer	15	The number of bits applied to the fraction length to interpret the input data to the filter.

Name	Values	Default	Description
`InputOffset`	Integers in the range 0 ≤ **`InputOffset`** ≤ **`r-1`**	0	Contains a value derived from the number of input samples and the decimation factor — `InputOffset = mod(length(nx),m)` where `nx` is the number of input samples that have been processed so far and `m` is the decimation factor. The `InputOffset` property applies only when you set the `PersistentMemory` property to `true`. See "InputOffset" on page 6-113 for more information.
`InputWordLength`	Any positive integer	16	The number of bits applied to the word length to interpret the input data to the filter.
`NumberOfSections`	Any positive integer	2	Number of sections used in the decimator. Generally called `n`. Reflects either the number of decimator or comb sections, not the total number of sections in the filter.
`OutputFracLength`	Any positive integer	15	The number of bits applied to the fraction length to interpret the output data from the filter. Read-only.

Name	Values	Default	Description
`OutputWordLength`	Any positive integer	16	The number of bits applied to the word length to interpret the output data from the filter.
`PersistentMemory`	`false` or `true`	`false`	Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. When `PersistentMemory` is `false`, you cannot access the filter states. Setting `PersistentMemory` to `true` reveals the `States` property so you can modify the filter states.

Name	Values	Default	Description
`SectionWord Lengths`	Any integer or a vector of length 2*`n`.	16	Defines the bits per section used while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter `SectionWordLengths`

Name	Values	Default	Description
			as a scalar or vector of length 2*n, where n is the number of sections. When SectionWordLengths is a scalar, the scalar value is applied to each filter section. When SectionWordLengths is a vector of values, the values apply to the sections in order. The default is 16 for each section in the decimator. Available when FilterInternals is 'SpecifyWordLengths'.
States	filtstates.cic object	m+1-by-n matrix of zeros, after you call function int.	Stored conditions for the filter, including values for the integrator and comb sections before and after filtering. m is the differential delay of the comb section and n is the number of sections in the filter. The integrator states are stored in the first matrix row. States for the comb section fill the remaining rows in the matrix. Available for modification when PersistentMemory is true. Refer to the filtstates object in Signal Processing Toolbox documentation for more general information about the filtstates object.

Usage Modes

There are four modes of usage for this which are set using the `FilterInternals` property

- `FullPrecision` — All word and fraction lengths set to B_{max} + 1, called B_{accum} by Fred Harris in [3]. Full Precision is the default setting.
- `MinWordLengths` — Automatically set the sections for minimum word lengths.
- `SpecifyWordLengths` — Specify the word lengths for each section.
- `SpecifyPrecision` — Specify precision by providing values for the word and fraction lengths for each section.

Full Precision

In full precision mode, the word lengths of all sections and the output are set to B_{accum} as defined by

$$B_{accum} = ceil(N_{\mathrm{secs}}(Log_2(D \times M)) + InputWordLength)$$

where N_{secs} is the number of filter sections.

Section fraction lengths and the fraction length of the output are set to the input fraction length.

Here is the display for this mode:

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'FullPrecision'
```

Minimum Wordlengths

In minimum word length mode, you control the output word length explicitly. When the output word length is less than B_{accum}, roundoff noise is introduced at the output of the filter. Hogenauer's bit pruning theory (refer to [1]) states that one valid design criterion is to make the word lengths of the different sections of the filter smaller than B_{accum} as well, so that the roundoff noise introduced by all sections does not exceed the roundoff noise introduced at the output.

In this mode, the design calculates the word lengths of each section to meet the Hogenauer criterion. The algorithm subtracts the number of bits computed using eq. 21 in Hogenauer's paper from B_{accum} to determine the word length each section.

To compute the fraction lengths of the different sections, the algorithm notes that the bits thrown out for this word length criterion are least significant bits (LSB), therefore each bit thrown out at a particular section decrements the fraction length of that section by one bit compared to the input fraction length. Setting the output wordlength for the filter automatically sets the output fraction length as well.

Here is the display for this mode:

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'MinWordLengths'

OutputWordLength: 16
```

Specify word lengths

In this mode, the design algorithm discards the LSBs, adjusting the fraction length so that unrecoverable overflow does not occur, always producing a reasonable output.

You can specify the word lengths for all sections and the output, but you cannot control the fraction lengths for those quantities.

To specify the word lengths, you enter a vector of length 2*(`NumberOfSections`), where each vector element represents the word length for a section. If you specify a scalar, such as B_{accum}, the full-precision output word length, the algorithm expands that scalar to a vector of the appropriate size, applying the scalar value to each section.

The CIC design does not check that the specified word lengths are monotonically decreasing. There are some cases where the word lengths are not necessarily monotonically decreasing, for example

```
hcic=mfilt.cicdecim;
hcic.FilterInternals='minwordlengths';
hcic.Outputwordlength=14;
```

which are valid CIC filters but the word lengths do not decrease monotonically across the sections.

Here is the display for the `SpecifyWordLengths` mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyWordLengths'

SectionWordLengths: [19 18 18 17]

OutputWordLength: 16
```

Specify precision

In this mode, you have full control over the word length and fraction lengths of all sections and the filter output.

When you elect the `SpecifyPrecision` mode, you must enter a vector of length 2*(`NumberOfSections`) with elements that represent the word length for each section. When you enter a scalar such as B_{accum}, `mfilt.cicdecim` expands that scalar to a vector of the appropriate size and applies the scalar value to each section and the output. The design does not check that this vector is monotonically decreasing.

Also, you must enter a vector of length 2*(`NumberOfSections`) with elements that represent the fraction length for each section as well. When you enter a scalar such as B_{accum}, `mfilt.cicdecim` applies scalar expansion as done for the word lengths.

Here is the SpecifyPrecision display.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyPrecision'

SectionWordLengths: [19 18 18 17]
SectionFracLengths: [14 13 13 12]

OutputWordLength: 16
OutputFracLength: 11
```

About the States of the Filter

In the states property you find the states for both the integrator and comb portions of the filter. states is a matrix of dimensions m + 1-by-n, with the states apportioned as follows:

- States for the integrator portion of the filter are stored in the first row of the state matrix.
- States for the comb portion fill the remaining rows in the state matrix.

To review the states of a CIC filter, use int to assign the states to a variable in MATLAB. As an example, here are the states for a CIC decimator hm before and after filtering a data set.

```
x = fi(ones(1,10),true,16,0); % Fixed-point input data.
hm = mfilt.cicdecim(2,1,2,16,16,16);
sts=int(hm.states)
set(hm,'InputFracLength',0); % Integer input specified.
y=filter(hm,x);
sts=int(hm.states)
```

STS is an integer matrix that int returns from the contents of the filtstates.cic object in hm.

Design Considerations

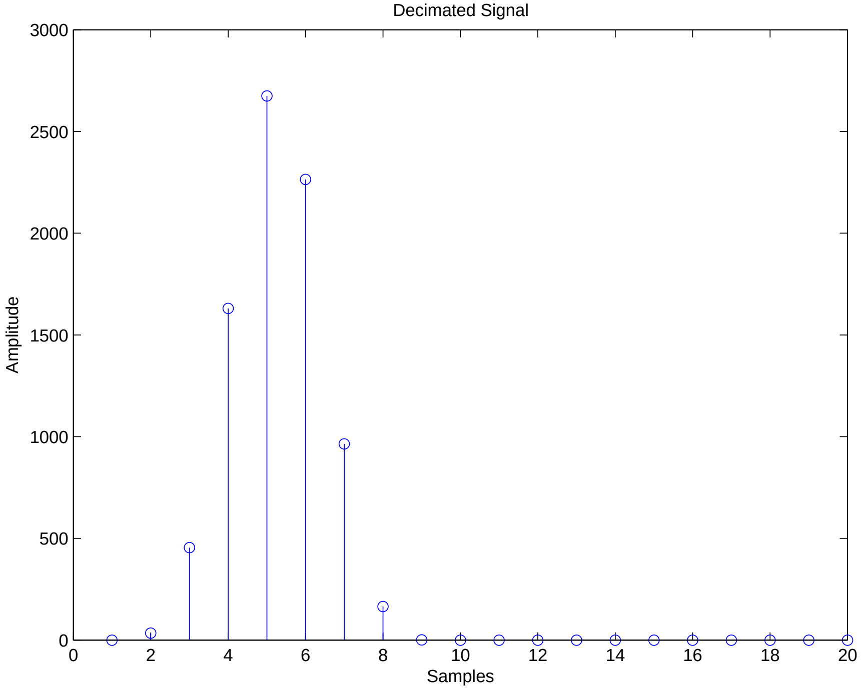
When you design your CIC decimation filter, remember the following general points:

- The filter output spectrum has nulls at $\omega = k * 2\pi/rm$ radians, $k = 1,2,3....$
- Aliasing and imaging occur in the vicinity of the nulls.
- n, the number of sections in the filter, determines the passband attenuation. Increasing n improves the filter ability to reject aliasing and imaging, but it also increases the droop (or rolloff) in the filter passband. Using an appropriate FIR filter in series after the CIC decimation filter can help you compensate for the induced droop.
- The DC gain for the filter is a function of the decimation factor. Raising the decimation factor increases the DC gain.

Examples

This example applies a decimation factor r equal to 8 to a 160-point impulse signal. The signal output from the filter has 160/r, or 20, points or samples. Choosing 10 bits for the word length represents a fairly common setting for analog to digital converters. The plot shown after the code presents the stem plot of the decimated signal, with 20 samples remaining after decimation:

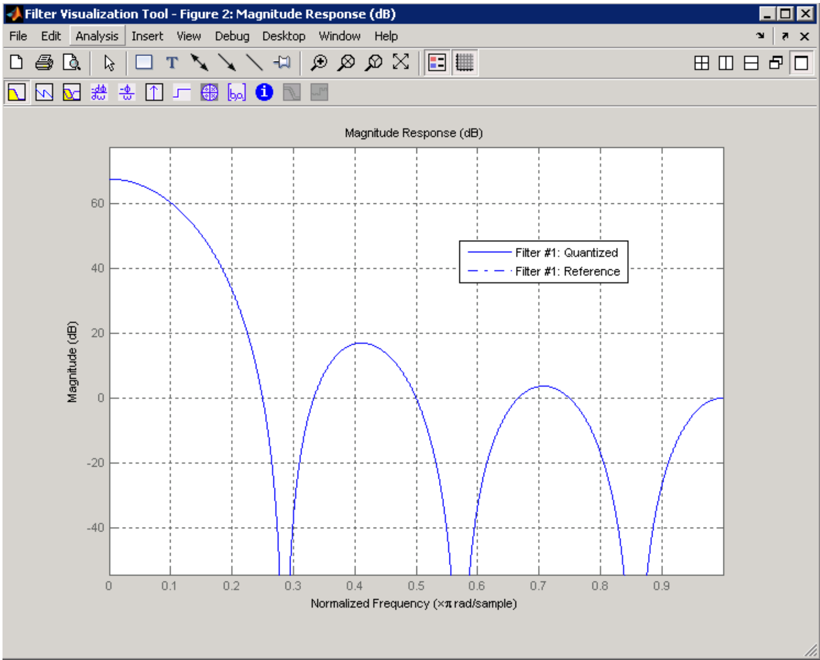
```
m = 2;  % Differential delays in the filter.
n = 4;  % Filter sections
r = 8;   % Decimation factor
x = int16(zeros(160,1)); x(1) = 1;    % Create a 160-point
                                      % impulse signal.
hm = mfilt.cicdecim(r,m,n); % Expects 16-bit input
                            % by default.
y = filter(hm,x);
stem(double(y));    % Plot output as a stem plot.
xlabel('Samples'); ylabel('Amplitude');
title('Decimated Signal');
```



The next example demonstrates one way to compute the filter frequency response, using a 4-section decimation filter with the decimation factor set to 7:

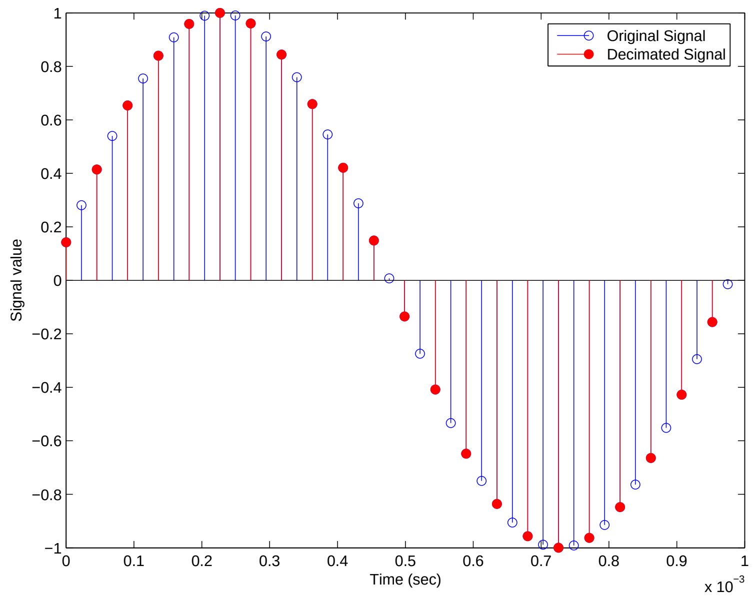
```
hm = mfilt.cicdecim(7,1,4);
fvtool(hm)
```

FVTool provides ways for you to change the title and x labels to match the figure shown. Here's the frequency response plot for the filter. For details about the transfer function used to produce the frequency response, refer to [1] in the References section.



This final example demonstrates the decimator for converting from 44.1 kHz audio to 22.05 kHz — decimation by two. To overlay the before and after signals, scale the output and plot the signals on a stem plot.

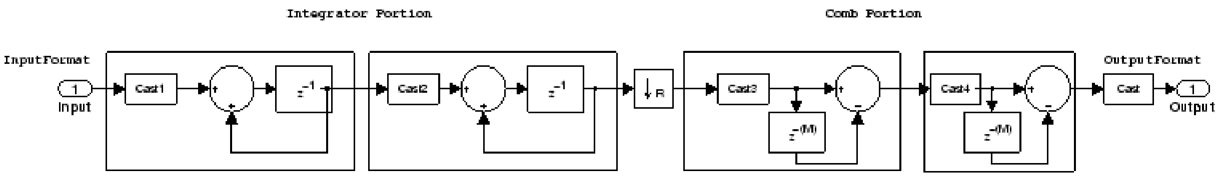
```
r = 2;                  % Decimation factor.
hm = mfilt.cicdecim(r); % Use default NumberOfSections &
                        % DifferentialDelay property values.
fs = 44.1e3;            % Original sampling frequency: 44.1kHz.
n = 0:10239;            % 10240 samples, 0.232 second long signal.
x  = sin(2*pi*1e3/fs*n);% Original signal, sinusoid at 1kHz.

y_fi = filter(hm,x); % 5120 samples, still 0.232 seconds.

% Scale the output to overlay the stem plots.
x = double(x);
y = double(y_fi);
y = y/max(abs(y));
stem(n(1:44)/fs,x(2:45)); hold on;  % Plot original signal
                                     % sampled at 44.1kHz.
stem(n(1:22)/(fs/r),y(3:24),'r','filled'); % Plot decimated
                                           % signal (22.05kHz)
                                           % in red.
xlabel('Time (seconds)');ylabel('Signal Value');
```



Algorithms

To show how the CIC decimation filter is constructed, the following figure presents a block diagram of the filter structure for a two-section CIC decimation filter (n = 2). *fs* is the high sampling rate, the input to the decimation process.

For details about the bits that are removed in the Comb section, refer to [1] in References.



`mfilt.cicdecim` calculates the fraction length at each section of the decimator to avoid overflows at the output of the filter.

References

[1] Hogenauer, E. B., “An Economical Class of Digital Filters for Decimation and Interpolation,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-29(2): pp. 155-162, 1981

[2] Meyer-Baese, Uwe, “Hogenauer CIC Filters,” in *Digital Signal Processing with Field Programmable Gate Arrays*, Springer, 2001, pp. 155-172

[3] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice-Hall PTR, 2004 , pp. 343

See Also

`mfilt` | `mfilt.cicinterp`

Introduced in R2011a

mfilt.cicinterp

Fixed-point CIC interpolator

Compatibility

`mfilt.cicinterp` will be removed in a future release. Use dsp.CICInterpolator instead.

Syntax

```
hm = mfilt.cicinterp(R,M,N,ILW,OWL,WLPS)
hm = mfilt.cicinterp
hm = mfilt.cicinterp(R,...)
```

Description

`hm = mfilt.cicinterp(R,M,N,ILW,OWL,WLPS)` constructs a cascaded integrator-comb (CIC) interpolation filter object that uses fixed-point arithmetic.

All of the input arguments are optional. To enter any optional value, you must include all optional values to the left of your desired value.

When you omit one or more input options, the omitted option applies the default values shown in the table below.

The following table describes the input arguments for creating `hm`.

Input Arguments	Description
R	Interpolation factor applied to the input signal. Sharpens the response curve to let you change the shape of the response. R must be an integer value greater than or equal to 1. The default value is 2.
M	Differential delay. Changes the shape, number, and location of nulls in the filter response. Increasing M increases the sharpness of the nulls and the response between nulls. In practice, differential delay values of 1 or 2 are the most common. M must be an integer value greater than or equal to 1. The default value is 1.

Input Arguments	Description
N	Number of sections. Deepens the nulls in the response curve. Note that this is the number of either comb or integrator sections, not the total section count. By default, the filter has two sections.
IWL	Word length of the input signal. Use any integer number of bits. The default value is 16 bits.
OWL	Word length of the output signal. It can be any positive integer number of bits. By default, OWL is 16 bits.
WLPS	Defines the number of bits per word in each filter section while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter WLPS as a scalar or vector of length 2*N, where N is the number of sections. When WLPS is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the integrator. When you elect to specify wlps as an input argument, the FilterInternals property automatically switches from the default value of 'FullPrecision' to 'SpecifyWordLengths'.

hm = mfilt.cicinterp constructs the CIC interpolator using the default values for the optional input arguments.

hm = mfilt.cicinterp(R,...) constructs the CIC interpolator applying the values you provide for R and any other values you specify as input arguments.

Constraints and Conversions

In Hogenauer [1], the author describes the constraints on CIC interpolator filters. mfilt.cicinterp enforces a constraint—the word lengths of the filter sections must be non-decreasing. That is, the word length of each filter section must be the same size as, or greater than, the word length of the previous filter section.

The formula for W_j, the minimum register width, is derived in [1]. The formula for W_j is given by

$$W_j = ceil(B_{in} + \log_2 G_j)$$

where G_j, the maximum register growth up to the jth section, is given by

$$G_j = \begin{cases} 2^j, & j = 1,2,\ldots,N \\ \dfrac{2^{2N-j}(RM)^{j-N}}{R}, & j = N+1,\ldots,2N \end{cases}$$

When the differential delay, M, is 1, there is also a special condition for the register width of the last comb, W_N, that is given by

$$W_N = B_{in} + N - 1 \quad if \quad M = 1$$

The conversions denoted by the cast blocks in the integrator diagrams in "Algorithms" on page 5-1135 perform the changes between the word lengths of each section. When you specify word lengths that do not follow the constraints described in this section, `mfilt.cicinterp` returns an error.

The fraction lengths and scalings of the filter sections do not change. At each section the word length is either staying the same or increasing. The signal scaling can change at the output after the final filter section if you choose the output word length to be less than the word length of the final filter section.

Properties of the Object

The following table lists the properties for the filter with a description of each.

Name	Values	Default	Description
`Arithmetic`	`fixed`	`fixed`	Reports the kind of arithmetic the filter uses. CIC interpolators are always fixed-point filters.
`InterpolationFactor`	Any positive integer	2	Amount to increase the input sampling rate.
`DifferentialDelay`	Any positive integer	1	Sets the differential delay for the filter. Usually a value of one or two is appropriate.
`FilterStructure`	`mfilt` structure string	None	Reports the type of filter object, such as a interpolator or fractional integrator. You cannot set this property — it is always read only and results from your choice of `mfilt` objects.
`FilterInternals`	`FullPrecision`, `MinWordLengths`, `SpecifyWordLengths`, `SpecifyPrecision`	`FullPrecision`	Set the usage mode for the filter. Refer to "Usage Modes" on page 5-1132 below for details.
`InputFracLength`	Any positive integer	16	The number of bits applied as the fraction length to interpret the input data to the filter.
`InputWordLength`	Any positive integer	16	The number of bits applied to the word length to interpret the input data to the filter.
`NumberOfSections`	Any positive integer	2	Number of sections used in the interpolator. Generally

Name	Values	Default	Description
			called `n`. Reflects either the number of interpolator or comb sections, not the total number of sections in the filter.
`OutputFracLength`	Any positive integer	15	The number of bits applied to the fraction length to interpret the output data from the filter. Read-only.
`OutputWordLength`	Any positive integer	16	The number of bits applied to the word length to interpret the output data from the filter.
`PersistentMemory`	`false` or `true`	`false`	Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. When `PersistentMemory` is `false`, you cannot access the filter states. Setting `PersistentMemory` to `true` reveals the `States` property so you can modify the filter states.

Name	Values	Default	Description
SectionWordLengths	Any integer or a vector of length 2^N, where N is a positive integer. This property only applies when the FilterInternals is SpecifyWordLengths.	16	Defines the bits per section used while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter SectionWordLengths as a scalar or vector of length 2*n, where n is the number of sections. When SectionWordLengths is a scalar, the scalar value is applied to each filter section. When SectionWordLengths is a vector of values, the values apply to the sections in order. The default is 16 for each section in the interpolator. Available when FilterInternals is 'SpecifyWordLengths'.

Name	Values	Default	Description
States	filtstates.cic object	m+1-by-n matrix of zeros, after you call function int.	Stored conditions for the filter, including values for the integrator and comb sections before and after filtering. m is the differential delay of the comb section and n is the number of sections in the filter. The integrator states are stored in the first matrix row. States for the comb section fill the remaining rows in the matrix. Available for modification when PersistentMemory is true. Refer to the filtstates object in Signal Processing Toolbox documentation for more general information about the filtstates object.

Usage Modes

There are usage modes which are set using the FilterInternals property:

- FullPrecision — In this mode, the word and fraction lengths of the filter sections and outputs are automatically selected for you. The output and last section word lengths are set to:

  $$\text{wordlength} = \text{ceil}(\log_2((RM)^N \,/\, R)) + I,$$

  where R is the interpolation factor, M is the differential delay, N is the number of filter sections, and I denotes the input word length.
- MinWordLengths — In this mode, you specify the word length of the filter output in the OutputWordLength property. The word lengths of the filter sections are automatically set in the same way as in the FullPrecision mode. The section fraction lengths are set to the input fraction length. The output fraction length is set

to the input fraction length minus the difference between the last section and output word lengths.

- SpecifyWordLengths — In this mode, you specify the word lengths of the filter sections and output in the SectionWordLengths and OutputWordLength properties. The fraction lengths of the filter sections are set such that the spread between word length and fraction length is the same as in full-precision mode. The output fraction length is set to the input fraction length minus the difference between the last section and output word lengths.
- SpecifyPrecision — In this mode, you specify the word and fraction lengths of the filter sections and output in the SectionWordLengths, SectionFracLengths, OutputWordLength, and OutputFracLength properties.

About the States of the Filter

In the states property you find the states for both the integrator and comb portions of the filter. states is a matrix of dimensions m+1-by-n, with the states apportioned as follows:

- States for the integrator portion of the filter are stored in the first row of the state matrix.
- States for the comb portion fill the remaining rows in the state matrix.

To review the states of a CIC filter, use the int method to assign the states. As an example, here are the states for a CIC interpolator hm before and after filtering data:

```
x = fi(cos(pi/4*[0:99]),true,16,0); % Fixed-point input data
hm = mfilt.cicinterp(2,1,2,16,16,16);
% get initial states-all zero
sts=int(hm.states)
set(hm,'InputFracLength',0); % Integer input specified
y=filter(hm,x);
sts=int(hm.states)
%sts =
%
%     -1     -1
%     -1     -1
```

Design Considerations

When you design your CIC interpolation filter, remember the following general points:

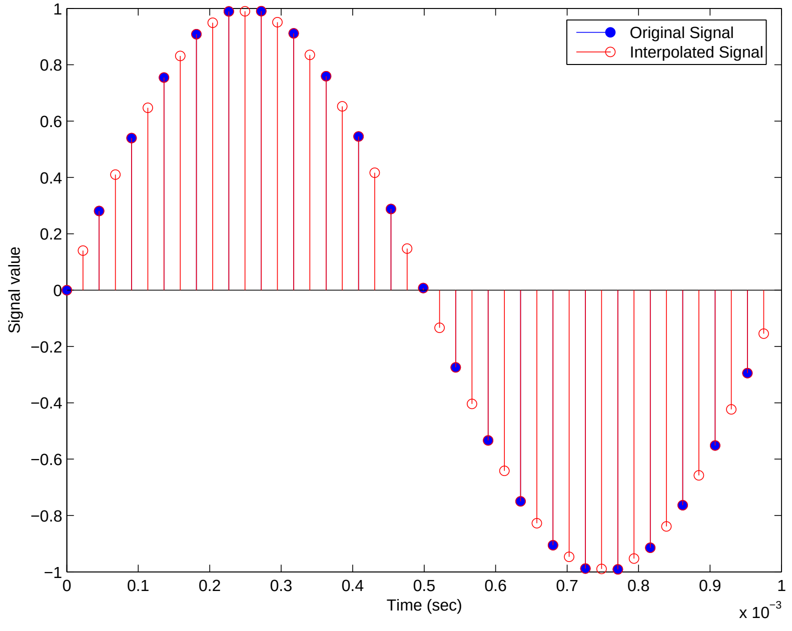
- The filter output spectrum has nulls at $\omega = k * 2\pi/\mathtt{rm}$ radians, $k = 1,2,3....$
- Aliasing and imaging occur in the vicinity of the nulls.
- n, the number of sections in the filter, determines the passband attenuation. Increasing n improves the filter ability to reject aliasing and imaging, but it also increases the droop or rolloff in the filter passband. Using an appropriate FIR filter in series after the CIC interpolation filter can help you compensate for the induced droop.
- The DC gain for the filter is a function of the interpolation factor. Raising the interpolation factor increases the DC gain.

Examples

Demonstrate interpolation by a factor of two, in this case from 22.05 kHz to 44.1 kHz. Note the scaling required to see the results in the stem plot and to use the full range of the int16 data type.

```
R = 2;                      % Interpolation factor.
hm = mfilt.cicinterp(R);    % Use default NumberOfSections and
                            % DifferentialDelay property values.
fs = 22.05e3;               % Original sample frequency:22.05 kHz.
n = 0:5119;                % 5120 samples, .232 second long signal.
x = sin(2*pi*1e3/fs*n);    % Original signal, sinusoid at 1 kHz.

y_fi = filter(hm,x); % 5120 samples, still 0.232 seconds.

% Scale the output to overlay stem plots correctly.
x = double(x);
y = double(y_fi);
y = y/max(abs(y));
stem(n(1:22)/fs,x(1:22),'filled'); % Plot original signal sampled
                                   % at 22.05 kHz.
hold on;
stem(n(1:44)/(fs*R),y(4:47),'r');  % Plot interpolated signal
                                   % (44.1 kHz) in red.
xlabel('Time (sec)');ylabel('Signal Value');
```

As you expect, the plot shows that the interpolated signal matches the input sine shape, with additional samples between each original sample.



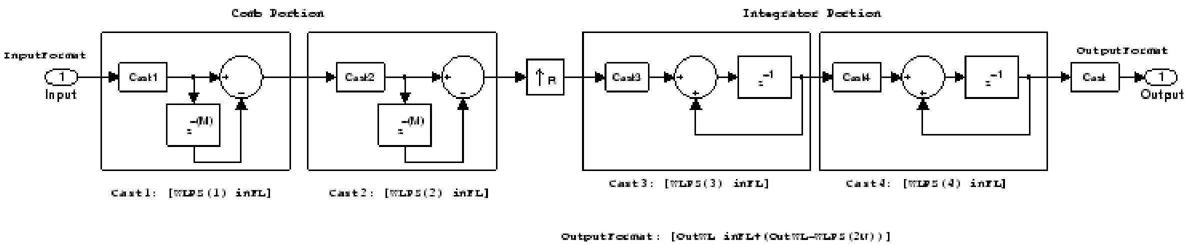
Use the filter visualization tool (FVTool) to plot the response of the interpolator object. For example, to plot the response of an interpolator with an interpolation factor of 7, 4 sections, and 1 differential delay, do something like the following:

```
hm = mfilt.cicinterp(7,1,4)
fvtool(hm)
```

Algorithms

To show how the CIC interpolation filter is constructed, the following figure presents a block diagram of the filter structure for a two-section CIC interpolation filter (`n` = 2). *fs* is the high sampling rate, the output from the interpolation process.

For details about the bits that are removed in the integrator section, refer to [1] in References.



When you select `MinWordLengths`, the filter section word lengths are automatically set to the minimum number of bits possible in a valid CIC interpolator. `mfilt.cicinterp` computes the wordlength for each section so the roundoff noise introduced by all sections is less than the roundoff noise introduced by the quantization at the output.

References

[1] Hogenauer, E. B., “An Economical Class of Digital Filters for Decimation and Interpolation,” IEEE Transactions on Acoustics, Speech, and Signal Processing, ASSP-29(2): pp. 155-162, 1981

[2] Meyer-Baese, Uwe, “Hogenauer CIC Filters,” in Digital Signal Processing with Field Programmable Gate Arrays, Springer, 2001, pp. 155-172

[3] Harris, Fredric J., *Multirate Signal Processing for Communication Systems*, Prentice-Hall PTR, 2004 , pp. 343

Introduced in R2011a

mfilt.farrowsrc

Sample rate converter with arbitrary conversion factor

Compatibility

`mfilt.farrowsrc` will be removed in a future release. Use dsp.FarrowRateConverter instead.

Syntax

```
hm = mfilt.farrowsrc(L,M,C)
hm = mfilt.farrowsrc
hm = mfilt.farrowsrc(l,...)
```

Description

`hm = mfilt.farrowsrc(L,M,C)` returns a filter object that is a natural extension of `dfilt.farrowfd` with a time-varying fractional delay. It provides a economical implementation of a sample rate converter with an arbitrary conversion factor. This filter works well in the interpolation case, but may exhibit poor anti-aliasing properties in the decimation case.

Note: You can use the `realizemdl` method to create a Simulink block of a filter created using `mfilt.farrowsrc`.

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
`l`	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. The default value of `l` is 3.

Input Argument	Description
`m`	Decimation factor for the filter. `m` specifies the amount to decrease the input sampling rate. The default value for `m` is 2.
`c`	Coefficients for the filter. When no input arguments are specified, the default coefficients are `[-1 1; 1, 0]`

`hm = mfilt.farrowsrc` constructs the filter using the default values for `l`, `m`, and `c`.

`hm = mfilt.farrowsrc(l,...)` constructs the filter using the input arguments you provide and defaults for the argument you omit.

mfilt.farrowsrc Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.farrowsrc` objects. The next table describes each property for an `mfilt.farrowsrc` filter object.

Name	Values	Description
`FilterStructure`	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object.
`Arithmetic`	String	Reports the arithmetic precision used by the filter.
`Coefficients`	Vector	Vector containing the coefficients of the FIR lowpass filter
`InterpolationFactor`	Integer	Interpolation factor for the filter. It specifies the amount to increase the input sampling rate.
`DecimationFactor`	Integer	Decimation factor for the filter. It specifies the amount to increase the input sampling rate.
`PersistentMemory`	`false` or `true`	Determines whether the filter states are restored to their starting values for each filtering operation. The starting

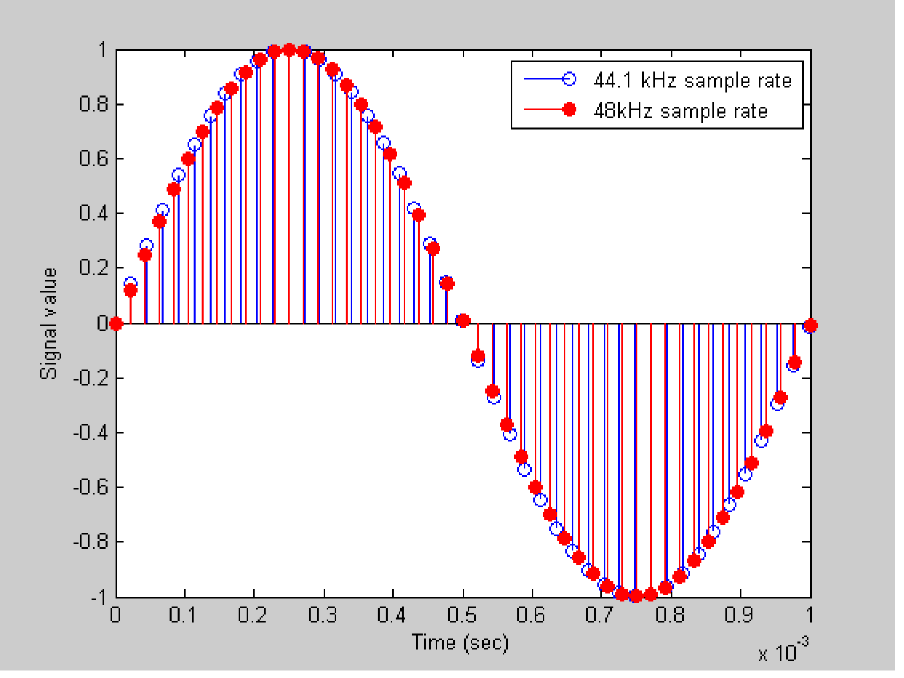
Name	Values	Description
		values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.

Examples

Interpolation by a factor of 8. This object removes the spectral replicas in the signal after interpolation.

```
[L,M] = rat(48/44.1);
Hm = mfilt.farrowsrc(L,M);            % We use the default filter
Fs = 44.1e3;                          % Original sampling frequency
n = 0:9407;                           % 9408 samples, 0.213 seconds long
x  = sin(2*pi*1e3/Fs*n);              % Original signal, sinusoid at 1kHz
y = filter(Hm,x);                     % 10241 samples, still 0.213 seconds
stem(n(1:45)/Fs,x(1:45))              % Plot original sampled at 44.1kHz
hold on
% Plot fractionally interpolated signal (48kHz) in red
stem((n(2:50)-1)/(Fs*L/M),y(2:50),'r','filled')
xlabel('Time (sec)');ylabel('Signal value')
legend('44.1 kHz sample rate','48kHz sample rate')
```

The results of the example are shown in the following figure:



See Also

dsp.FarrowRateConverter

Introduced in R2011a

mfilt.fftfirinterp

Overlap-add FIR polyphase interpolator

Compatibility

`mfilt.fftfirinterp` will be removed in a future release. Use dsp.FIRInterpolator instead.

Syntax

```
hm = mfilt.fftfirinterp(l,num,bl)
hm = mfilt.fftfirinterp
hm = mfilt.fftfirinterp(l,...)
```

Description

`hm = mfilt.fftfirinterp(l,num,bl)` returns a discrete-time FIR filter object that uses the overlap-add method for filtering input data.

The input arguments are optional. To enter any optional value, you must include all optional values to the left of your desired value.

When you omit one or more input options, the omitted option applies the default values shown in the table below.

The number of FFT points is given by `[bl+ceil(length(num)/l)-1]`. It is to your advantage to choose `bl` such that the number of FFT points is a power of two—using powers of two can improve the efficiency of the FFT and the associated interpolation process.

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
`l`	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for `l` it defaults to 2.
`num`	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When `num` is not provided as an input, `fftfirinterp` uses a lowpass Nyquist filter with gain equal to `l` and cutoff frequency equal to `π/l` by default.
`bl`	Length of each block of input data used in the filtering. `bl` must be an integer. When you omit input `bl`, it defaults to 100

`hm = mfilt.fftfirinterp` constructs the filter using the default values for `l`, `num`, and `bl`.

`hm = mfilt.fftfirinterp(l,...)` constructs the filter using the input arguments you provide and defaults for the argument you omit.

mfilt.fftfirinterp Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.fftfirinterp` objects. The next table describes each property for an `mfilt.fftfirinterp` filter object.

Name	Values	Description
`FilterStructure`		Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object.
`Numerator`		Vector containing the coefficients of the FIR lowpass filter used for interpolation.
`InterpolationFactor`		Interpolation factor for the filter. It specifies the amount to increase the input sampling rate. It must be an integer.
`BlockLength`		Length of each block of input data used in the filtering.

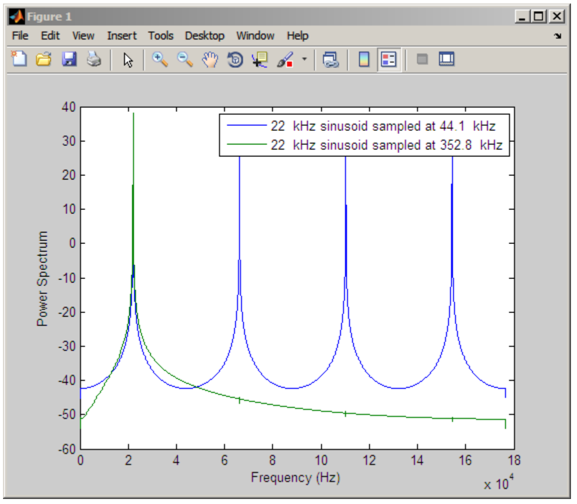
Name	Values	Description
PersistentMemory	false or true	Determines whether the filter states are restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.
States		Stored conditions for the filter, including values for the interpolator states.

Examples

Interpolation by a factor of 8. This object removes the spectral replicas in the signal after interpolation.

```
l = 8;                        % Interpolation factor
hm = mfilt.fftfirinterp(l);   % We use the default filter
n = 8192;                     % Number of points
hm.blocklength = n;           % Set block length to number of points
fs = 44.1e3;                  % Original sample freq: 44.1 kHz.
n = 0:n-1;                    % 0.1858 secs of data
x = sin(2*pi*n*22e3/fs);      % Original signal, sinusoid at 22 kHz
y = filter(hm,x);             % Interpolated sinusoid
xu = l*upsample(x,8);         % Upsample to compare--the spectrum
                              % does not change
[px,f]=periodogram(xu,[],65536,l*fs);% Power spectrum of original
                                     % signal
[py,f]=periodogram(y,[],65536,l*fs); % Power spectrum of
                                     % interpolated signal
plot(f,10*log10(([fs*px,l*fs*py])))
legend('22  kHz sinusoid sampled at 44.1  kHz',...
'22  kHz sinusoid sampled at 352.8  kHz')
xlabel('Frequency (Hz)'); ylabel('Power Spectrum');
```

To see the results of the example, look at this figure.



See Also

`mfilt.firinterp` | `mfilt.holdinterp` | `mfilt.linearinterp` | `mfilt.firsrc` | `mfilt.cicinterp`

Introduced in R2011a

mfilt.firdecim

Direct-form FIR polyphase decimator

Compatibility

`mfilt.firdecim` will be removed in a future release. Use dsp.FIRDecimator instead.

Syntax

```
hm = mfilt.firdecim(m)
hm = mfilt.firdecim(m,num)
```

Description

`hm = mfilt.firdecim(m)` returns a direct-form FIR polyphase decimator object `hm` with a decimation factor of `m`. A lowpass Nyquist filter of gain 1 and cutoff frequency of `π/m` is designed by default. This filter allows some aliasing in the transition band but it very efficient because the first polyphase component is a pure delay.

`hm = mfilt.firdecim(m,num)` uses the coefficients specified by `num` for the decimation filter. This lets you specify more completely the FIR filter to use for the decimator.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
m	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for m it defaults to 2.
num	Vector containing the coefficients of the FIR lowpass filter used for decimation. When num is not provided as an input, mfilt.firdecim constructs a lowpass Nyquist filter with gain of 1 and cutoff frequency equal to π/m by default. The default length for the Nyquist filter is 24*m. Therefore, each polyphase filter component has length 24.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating mfilt.firdecim objects. The next table describes each property for an mfilt.firdecim filter object.

Name	Values	Description
Arithmetic	Double, single, fixed	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operation mode for your filter.
DecimationFactor	Integer	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer.
FilterStructure	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice

Name	Values	Description
		of `mfilt` object. Describes the signal flow for the filter object.
`InputOffset`	Integers	Contains a value derived from the number of input samples and the decimation factor — `InputOffset = mod(length(nx),m)` where `nx` is the number of input samples that have been processed so far and `m` is the decimation factor.
`Numerator`	Vector	Vector containing the coefficients of the FIR lowpass filter used for decimation.
`PersistentMemory`	`false`, `true`	Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` set to `false` returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to `true` allows you to modify the `States`, `InputOffset`, and `PolyphaseAccum` properties.
`PolyphaseAccum`	0 in `double`, `single`, or `fixed` for the different filter arithmetic settings.	Differentiates between the adders in the filter that work in full precision at all times (`PolyphaseAccum`) and the adders in the filter that the user controls and that may introduce quantization effects when `FilterInternals` is set to `SpecifyPrecision`.

Name	Values	Description
`States`	`Double`, `single`, or `fi` matching the filter arithmetic setting.	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Double is the default setting for floating-point filters in double arithmetic.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the filter. You see one or more of these properties when you set Arithmetic to fixed. Some of the properties have different default values when they refer fixed point filters. One example is the property `PolyphaseAccum` which stores data as doubles when you use your filter in double-precision mode, but stores a `fi` object in fixed-point mode.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use `info(hm)` where `hm` is a filter.

For further information about the properties of this filter or any `mfilt` object, refer to "Multirate Filter Properties" on page 6-104.

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits [32]	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters.
`AccumWordLength`	Any integer number of bits [39]	Sets the word length used to store data in the accumulator.
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by

Name	Values	Description
		setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
`CoeffWordLength`	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
`FilterInternals`	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Any integer number of bits[16]	Specifies the word length applied to interpret input data.
`OutputFracLength`	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `FilterInternals` to `SpecifyPrecision`.
`OutputWordLength`	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting `FilterInternals` to `SpecifyPrecision`.

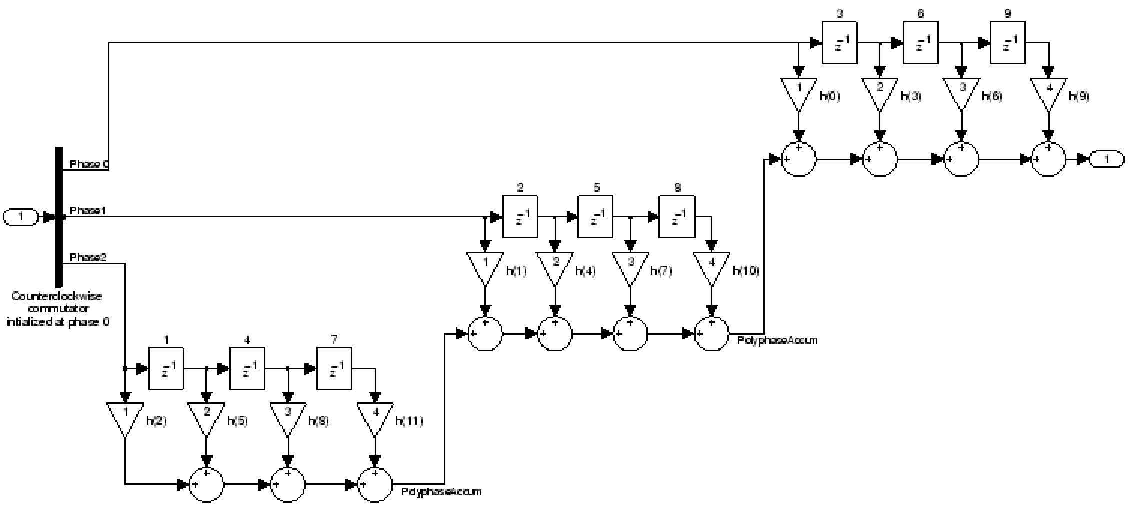
Name	Values	Description
`OverflowMode`	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
`RoundMode`	`[convergent]`, `ceil`, `fix`, `floor`, `nearest`, `round`	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `ceil` - Round toward positive infinity. • `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • `fix` - Round toward zero. • `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.

Name	Values	Description
`States`	`fi` object	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation. For information about the ordering of the states, refer to the filter structure section.

Filter Structure

To provide decimation, `mfilt.firdecim` uses the following structure. At the input you see a commutator that operates counterclockwise, moving from position 0 to position 2, position 1, and back to position 0 as input samples enter the filter.

The following figure details the signal flow for the direct form FIR filter implemented by `mfilt.firdecim`.



Notice the order of the states in the filter flow diagram. States 1 through 9 appear in the diagram above each delay element. State 1 applies to the first delay element in phase

2. State 2 applies to the first delay element in phase 1. State 3 applies to the first delay element in phase 0. State 4 applies to the second delay in phase 2, and so on. When you provide the states for the filter as a vector to the `States` property, the above description explains how the filter assigns the states you specify.

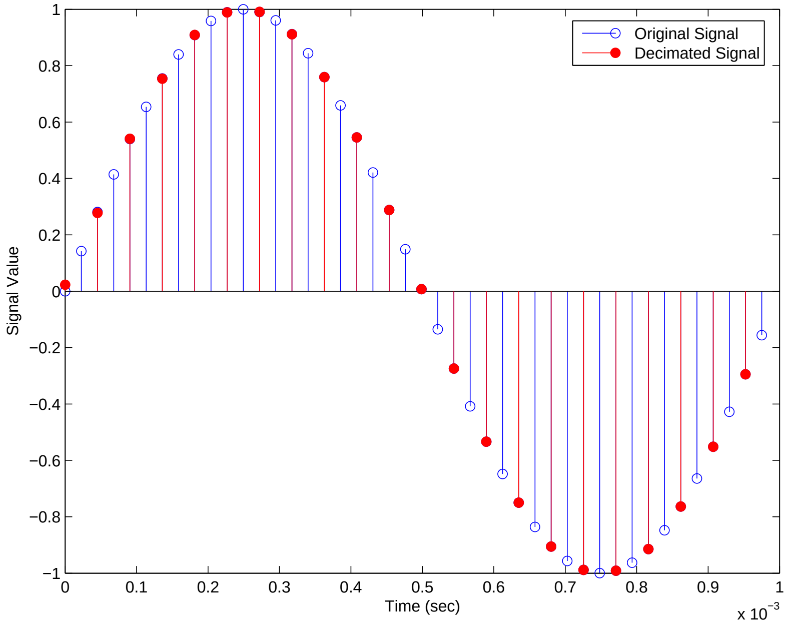
In property value form, the states for a filter `hm` are

```
hm.states=[1:9];
```

Examples

Convert an input signal from 44.1 kHz to 22.05 kHz using decimation by a factor of 2. In the figure that appears after the example code, you see the results of the decimation.

```
m = 2;                          % Decimation factor.
hm = mfilt.firdecim(m);         % Use the default filter.
fs = 44.1e3;                    % Original sample freq: 44.1kHz.
n = 0:10239;                    % 10240 samples, 0.232 second long
                                % signal.
x  = sin(2*pi*1e3/fs*n);        % Original signal--sinusoid at 1kHz.
y = filter(hm,x);               % 5120 samples, 0.232 seconds.
stem(n(1:44)/fs,x(1:44))        % Plot original sampled at 44.1 kHz.
hold on                         % Plot decimated signal (22.05 kHz)
                                % in red.
stem(n(1:22)/(fs/m),y(13:34),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value')
```



See Also

`mfilt.firtdecim` | `mfilt.firsrc` | `mfilt.cicdecim`

Introduced in R2011a

mfilt.firfracdecim (Obsolete)

Direct-form FIR polyphase fractional decimator

Syntax

```
hm = mfilt.firfracdecim(l,m,num)
```

Description

`mfilt.firfracdecim` will be removed in a future release. Use `mfilt.firsrc` instead.

`hm = mfilt.firfracdecim(l,m,num)` returns a direct-form FIR polyphase fractional decimator. Input argument `l` is the interpolation factor. `l` must be an integer. When you omit `l` in the calling syntax, it defaults to 2. `m` is the decimation factor. It must be an integer. If not specified, it defaults to l+1.

`num` is a vector containing the coefficients of the FIR lowpass filter used for decimation. If you omit `num`, a lowpass Nyquist filter of gain l and cutoff frequency of π/`max(l,m)` is used by default.

By specifying both a decimation factor and an interpolation factor, you can decimate your input signal by noninteger amounts. The fractional decimator first interpolates the input, then decimates to result in an output signal whose sample rate is `l/m` of the input rate. By default, the resulting decimation factor is 2/3 when you do not provide `l` and `m` in the calling syntax. Specify `l` smaller than `m` for proper decimation.

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
`l`	Interpolation factor for the filter. It must be an integer. When you do not specify a value for `l` it defaults to 2.
`num`	Vector containing the coefficients of the FIR lowpass filter used for decimation. When `num` is not provided as an input, `firfracdecim` uses a lowpass Nyquist filter with gain equal to `l` and cutoff frequency equal to π/`max(l,m)` by default.

Input Argument	Description
m	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for m it defaults to l + 1.

mfilt.firfracdecim Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating mfilt.firfracdecim objects. The next table describes each property for an mfilt.firfracdecim filter object.

Name	Values	Description
FilterStructure	String	Reports the type of filter object, such as a decimator or fractional decimator. You cannot set this property — it is always read only and results from your choice of mfilt object.
InputOffset	Integers	Contains the number of input data samples processed without generating an output sample. The default value is 0.
Numerator	Vector	Vector containing the coefficients of the FIR lowpass filter used for interpolation.
RateChangeFactors	[l,m]	Reports the decimation (m) and interpolation (l) factors for the filter object. Combining these factors results in the final rate change for the signal.
PersistentMemory	false or true	Determines whether the filter states are restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.

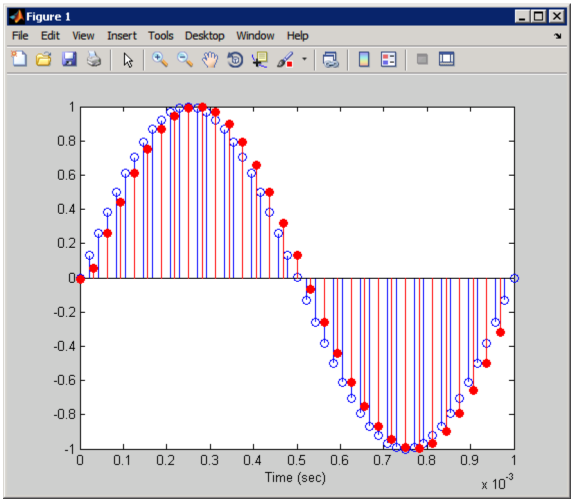
Name	Values	Description
`States`	Matrix	Stored conditions for the delays between each interpolator phase, the filter states, and the states at the output of each phase in the filter. The number of states is (`lh`-1)*`m`+(l-1)*(`lo`+`mo`) where `lh` is the length of each subfilter, and `l` and `m` are the interpolation and decimation factors. `lo` and `mo`, the input and output delays between each interpolation phase, are integers from Euclid's theorem such that `lo`*l-`mo`*`m` = -1 (refer to the reference for more details). Use `euclidfactors` to get `lo` and `mo` for an `mfilt.firfracdecim` object

Examples

To demonstrate `firfracdecim`, perform a fractional decimation by a factor of 2/3. This is one way to downsample a 48 kHz signal to 32 kHz, commonly done in audio processing.

```
l  = 2; m = 3;               % Interpolation/decimation factors.
hm = mfilt.firfracdecim(l,m); % We use the default
fs = 48e3;                   % Original sample freq: 48 kHz.
n  = 0:10239;                % 10240 samples, 0.213 second long
                             % signal
x  = sin(2*pi*1e3/fs*n);     % Original signal, sinusoid at 1 kHz
y  = filter(hm,x);           % 9408 samples, still 0.213 seconds
stem(n(1:49)/fs,x(1:49)); hold on; % Plot original signal sampled
                                   % at 48 kHz
stem(n(1:32)/(fs*l/m),y(13:44),'r','filled') % Plot decimated
                                             % signal at 32 kHz
xlabel('Time (sec)');
```

As shown, the plot clearly demonstrates the reduced sampling frequency of 32 kHz.



References

Fliege, N.J., *Multirate Digital Signal Processing*, John Wiley & Sons, Ltd., 1994

See Also

`mfilt.firsrc` | `mfilt.firinterp` | `mfilt.firdecim`

Introduced in R2011a

mfilt.firfracinterp (Obsolete)

Direct-form FIR polyphase fractional interpolator

Syntax

```
hm = mfilt.firfracinterp(l,m,num)
```

Description

`mfilt.firfracinterp` will be removed in a future release. Use `mfilt.firsrc` instead.

`hm = mfilt.firfracinterp(l,m,num)` returns a direct-form FIR polyphase fractional interpolator mfilt object. `l` is the interpolation factor. It must be an integer. If not specified, `l` defaults to 3.

`m` is the decimation factor. Like `l`, it must be an integer. If you do not specify `m` in the calling syntax, it defaults to 1. If you also do not specify a value for `l`, `m` defaults to 2.

`num` is a vector containing the coefficients of the FIR lowpass filter used for interpolation. If omitted, a lowpass Nyquist filter of gain `l` and cutoff frequency of π/`max(l,m)` is used by default.

By specifying both a decimation factor and an interpolation factor, you can interpolate your input signal by noninteger amounts. The fractional interpolator first interpolates the input, then decimates to result in an output signal whose sample rate is `l/m` of the input rate. For proper interpolation, you specify `l` to be greater than `m`. By default, the resulting interpolation factor is 3/2 when you do not provide `l` and `m` in the calling syntax.

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
l	Interpolation factor for the filter. l specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for l it defaults to 3.
num	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When num is not provided as an input, firfracinterp uses a lowpass Nyquist filter with gain equal to l and cutoff frequency equal to π/max(l,m) by default.
m	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for m it defaults to 1. When you do not specify l as well, m defaults to 2.

mfilt.firfracinterp Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.firfracinterp` objects. The next table describes each property for an `mfilt.firfracinterp` filter object.

Name	Values	Description
FilterStructure		Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of mfilt object.
Numerator		Vector containing the coefficients of the FIR lowpass filter used for interpolation.
RateChangeFactors	[l,m]	Reports the decimation (m) and interpolation (l) factors for the filter object. Combining these factors results in the final rate change for the signal.
PersistentMemory	false or true	Determines whether the filter states are restored to their starting values for each filtering operation. The starting values are the values in place when

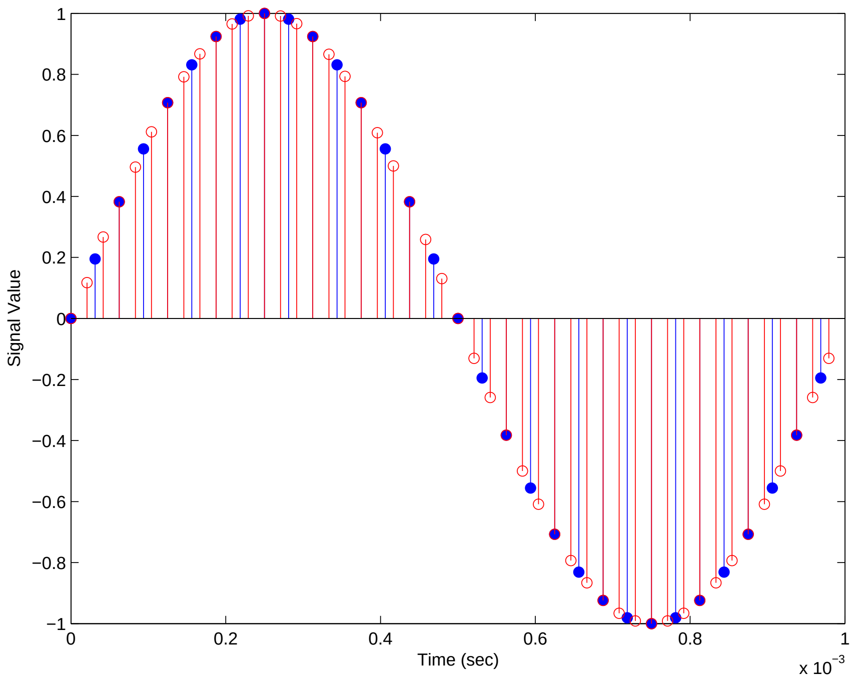
Name	Values	Description
		you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to the default values any state that the filter changes during processing. States that the filter does not change are not affected.
`States`	Matrix	Stored conditions for the filter, including values for the interpolator and comb states.

Examples

To convert a signal from 32 kHz to 48 kHz requires fractional interpolation. This example uses the `mfilt.firfracinterp` object to upsample an input signal. Setting `l` = 3 and `m` = 2 returns the same `mfilt` object as the default `mfilt.firfracinterp` object.

```
l  = 3; m = 2;               % Interpolation/decimation factors.
hm = mfilt.firfracinterp(l,m);    % We use the default filter
fs = 32e3;                   % Original sample freq: 32 kHz.
n = 0:6799;                  % 6800 samples, 0.212 second long signal
x  = sin(2*pi*1e3/fs*n);     % Original signal, sinusoid at 1 kHz
y = filter(hm,x);            % 10200 samples, still 0.212 seconds
stem(n(1:32)/fs,x(1:32),'filled') % Plot original sampled at
                                  % 32 kHz
hold on;
% Plot fractionally interpolated signal (48 kHz) in red
stem(n(1:48)/(fs*l/m),y(20:67),'r')
xlabel('Time (sec)');ylabel('Signal Value')
```

The ability to interpolate by fractional amounts lets you raise the sampling rate from 32 to 48 kHz, something you cannot do with integral interpolators. Both signals appear in the following figure.



See Also

`mfilt.firsrc` | `mfilt.firinterp` | `mfilt.firdecim`

Introduced in R2011a

mfilt.firinterp

FIR filter-based interpolator

Compatibility

`mfilt.firinterp` will be removed in a future release. Use dsp.FIRInterpolator instead.

Syntax

```
Hm = mfilt.firinterp(L)
Hm = mfilt.firinterp(L,num)
```

Description

`Hm = mfilt.firinterp(L)` returns a FIR polyphase interpolator object `Hm` with an interpolation factor of `L` and gain equal to `L`. `L` defaults to 2 if unspecified.

`Hm = mfilt.firinterp(L,num)` uses the values in the vector `num` as the coefficients of the interpolation filter.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `Hm` as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
`l`	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for `l` it defaults to 2.
`num`	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When `num` is not provided as an input, `firinterp` uses a lowpass Nyquist filter with gain equal to `l` and cutoff frequency equal to `π/l` by default. The default length for the Nyquist filter is 24*`l`. Therefore, each polyphase filter component has length 24.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.firinterp` objects. The next table describes each property for an `mfilt.firinterp` filter object.

Name	Values	Description
`Arithmetic`	`Double`, `single`, `fixed`	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operation mode for your filter.
`FilterStructure`	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object. Describes the signal flow for the filter object.
`InterpolationFactor`	Integer	Interpolation factor for the filter. `l` specifies the amount to increase the sampling rate of the input signal. It must be an integer.

Name	Values	Description
`Numerator`	Vector	Vector containing the coefficients of the FIR lowpass filter used for decimation.
`PersistentMemory`	`[false]`, `true`	Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` set to `false` returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to `true` allows you to modify the `States` property.
`States`	`Double`, `single`, matching the filter arithmetic setting.	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the `mfilt.firinterp` filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where `hm` is a filter.

For further information about the properties of this filter or any `mfilt` object, refer to "Multirate Filter Properties" on page 6-104.

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits. [32]	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and

Name	Values	Description
		lattice filters. IIR filters have two similar properties — DenAccumFracLength and NumAccumFracLength — that let you set the precision for numerator and denominator operations separately.
AccumWordLength	Any integer number of bits[39]	Sets the word length used to store data in the accumulator.
Arithmetic	fixed for fixed-point filters	Setting this to fixed allows you to modify other filter properties to customize your fixed-point filter.
CoeffAutoScale	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength property value to specify the precision used.
CoeffWordLength	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data.

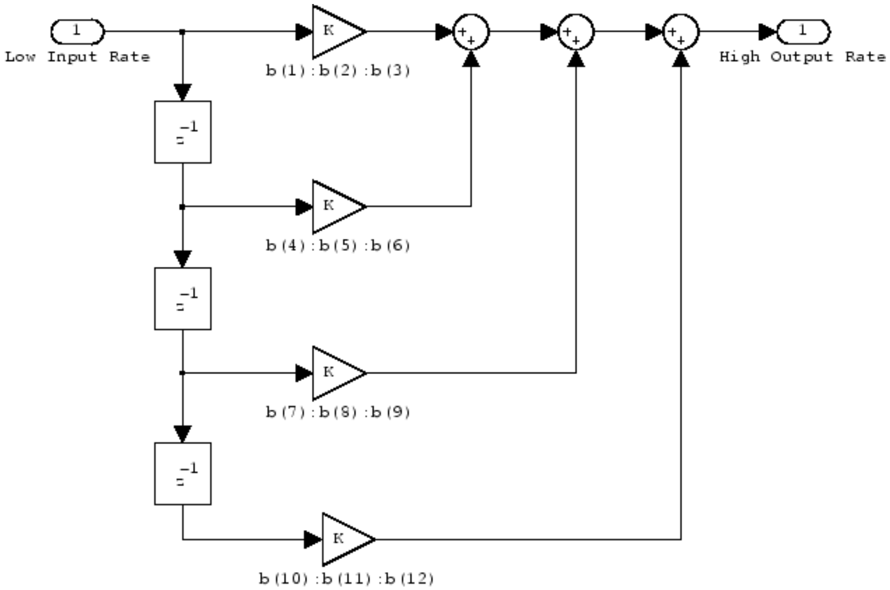
Name	Values	Description
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
OutputFracLength	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilterInternals to SpecifyPrecision.
OutputWordLength	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.

Name	Values	Description
`RoundMode`	[`convergent`], `ceil`, `fix`, `floor`, `nearest`, `round`	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `ceil` - Round toward positive infinity. • `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • `fix` - Round toward zero. • `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	`fi` object to match the filter arithmetic setting.	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation.

Filter Structure

To provide interpolation, `mfilt.firinterp` uses the following structure.

The following figure details the signal flow for the direct form FIR filter implemented by `mfilt.firinterp`. In the figure, the delay line updates happen at the lower input rate. The remainder of the filter — the sums and coefficients — operate at the higher output rate.



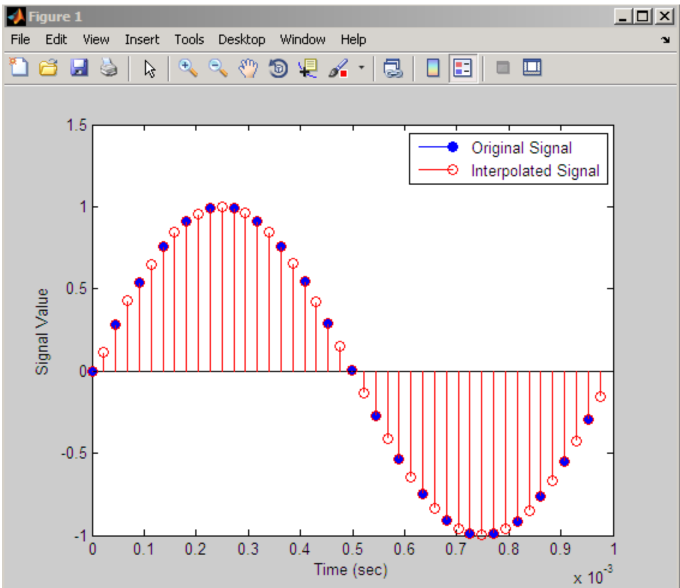
Examples

This example uses `mfilt.firinterp` to double the sample rate of a 22.05 kHz input signal. The output signal ends up at 44.1 kHz. Although `l` is set explicitly to `2`, this represents the default interpolation value for `mfilt.firinterp` objects.

```
L = 2;                      % Interpolation factor.
Hm = mfilt.firinterp(L);    % Use the default filter.
fs = 22.05e3;                % Original sample freq: 22.05 kHz.
n = 0:5119;                   % 5120 samples, 0.232s long signal.
x  = sin(2*pi*1e3/fs*n);    % Original signal, sinusoid at 1 kHz.
y = filter(Hm,x);           % 10240 samples, still 0.232s.
```

```
stem(n(1:22)/fs,x(1:22),'filled') % Plot original sampled at
                                  % 22.05 kHz.
hold on;

% Plot interpolated signal (44.1 kHz) in red
stem(n(1:44)/(fs*L),y(25:68),'r')
xlabel('Time (sec)');ylabel('Signal Value')
legend('Original Signal','Interpolated Signal');
```

With interpolation by 2, the resulting signal perfectly matches the original, but with twice as many samples — one between each original sample, as shown in the following figure.



See Also

mfilt.holdinterp | mfilt.linearinterp | mfilt.fftfirinterp | mfilt.firsrc | mfilt.cicinterp

Introduced in R2011a

mfilt.firsrc

Direct-form FIR polyphase sample rate converter

Compatibility

`mfilt.firsrc` will be removed in a future release. Use dsp.FIRRateConverter instead.

Syntax

```
hm = mfilt.firsrc(l,m,num)
```

Description

`hm = mfilt.firsrc(l,m,num)` returns a direct-form FIR polyphase sample rate converter. `l` specifies the interpolation factor. It must be an integer and when omitted in the calling syntax, it defaults to 2.

`m` is the decimation factor. It must be an integer. If not specified, `m` defaults to 1. If `l` is also not specified, `m` defaults to 3 and the overall rate change factor is 2/3.

You specify the coefficients of the FIR lowpass filter used for sample rate conversion in `num`. If omitted, a lowpass Nyquist filter with gain `l` and cutoff frequency of π/`max`(`l`,`m`) is the default.

Combining an interpolation factor and a decimation factor lets you use `mfilt.firsrc` to perform fractional interpolation or decimation on an input signal. Using an `mfilt.firsrc` object applies a rate change factor defined by `l/m` to the input signal. For proper rate changing to occur, `l` and `m` must be relatively prime — meaning the ratio `l/m` cannot be reduced to a ratio of smaller integers.

When you are doing sample-rate conversion with large values of `l` or `m`, such as `l` or `m` greater than 20, using the `mfilt.firsrc` structure is the most effective approach.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

```
set(hm,'arithmetic','single');
```

- To change to fixed-point filtering, enter

```
set(hm,'arithmetic','fixed');
```

Note: You can use the `realizemdl` method to create a Simulink block of a filter created using `mfilt.firsrc`.

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
`l`	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for `l`, it defaults to 2.
`num`	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When `num` is not provided as an input, `mfilt.firsrc` uses a lowpass Nyquist filter with gain equal to `l` and cutoff frequency equal to `π/max(l,m)` by default. The default length for the Nyquist filter is 24*`max(1,m)`. Therefore, each polyphase filter component has length 24.
`m`	Decimation factor for the filter. `m` specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for `m`, it defaults to 1. When `l` is unspecified as well, `m` defaults to 3.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating mfilt.firsrc objects. The next table describes each property for an mfilt.firsrc filter object.

Name	Values	Description
Arithmetic	[Double], single, fixed	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operation mode for your filter.
FilterStructure	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of mfilt object. Describes the signal flow for the filter object.
InputOffset	Integers	Contains a value derived from the number of input samples and the decimation factor — InputOffset = mod(length(nx),m) where nx is the number of input samples and m is the decimation factor.
Numerator	Vector	Vector containing the coefficients of the FIR lowpass filter used for decimation.
PersistentMemory	false, true	Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory set to false returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to true allows you to modify the States, InputOffset, and PolyphaseAccum properties.
RateChangeFactors	Positive integers. [2 3]	Specifies the interpolation and decimation factors [l m] (the rate change factors) for changing the input sample rate by nonintegral amounts.
States	Double, single, matching the filter arithmetic setting.	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the `mfilt.firsrc` filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where `hm` is a filter.

For further information about the properties of this filter or any `mfilt` object, refer to "Multirate Filter Properties" on page 6-104.

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits. [32]	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters.
`AccumWordLength`	Any integer number of bits [39]	Sets the word length used to store data in the accumulator.
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
`CoeffWordLength`	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
`FilterInternals`	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best

Name	Values	Description
		precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
`NumFracLength`	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
`OutputFracLength`	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `FilterInternals` to `SpecifyPrecision`.
`OutputWordLength`	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting `FilterInternals` to `SpecifyPrecision`.
`OverflowMode`	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
`RateChangeFactors`	Positive integers [2 3]	Specifies the interpolation and decimation factors `[l m]` (the rate change factors) for changing the input sample rate by nonintegral amounts.

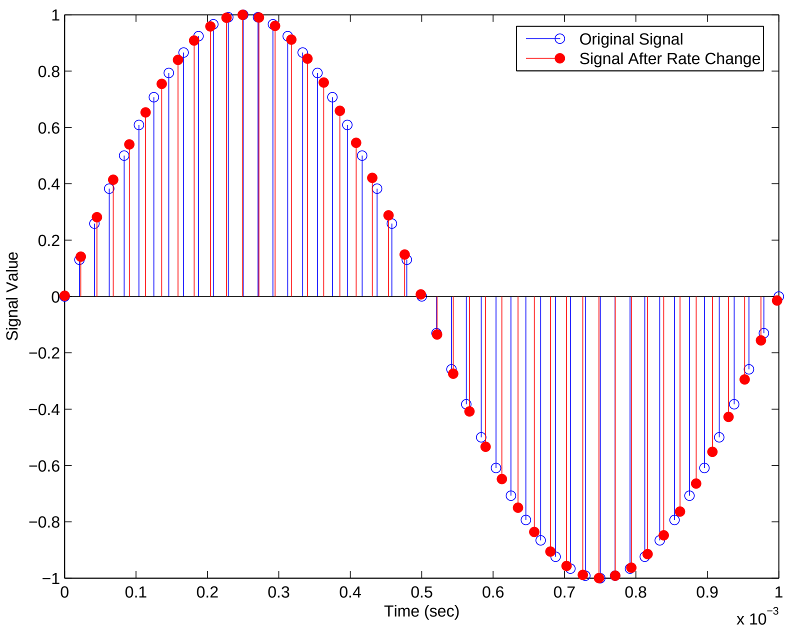
Name	Values	Description
`RoundMode`	[`convergent`], `ceil`, `fix`, `floor`, `nearest`, `round`	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `ceil` - Round toward positive infinity. • `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • `fix` - Round toward zero. • `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	`fi` object	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation. For information about the ordering of the states, refer to the filter structure section.

Examples

This is an example of a common audio rate change process — changing the sample rate of a high end audio (48 kHz) signal to the compact disc sample rate (44.1 kHz). This conversion requires a rate change factor of 0.91875, or `l` = 147 and `m` = 160.

```
l  = 147; m = 160;          % Interpolation/decimation factors.
hm = mfilt.firsrc(l,m);     % Use the default FIR filter.
fs = 48e3;                  % Original sample freq: 48 kHz.
n = 0:10239;                % 10240 samples, 0.213 seconds long.
x  = sin(2*pi*1e3/fs*n);    % Original signal, sinusoid at 1 kHz.
y = filter(hm,x);           % 9408 samples, still 0.213 seconds.
stem(n(1:49)/fs,x(1:49))    % Plot original sampled at 48 kHz.
hold on

% Plot fractionally decimated signal (44.1 kHz) in red
stem(n(1:45)/(fs*l/m),y(13:57),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value')
```

Fractional decimation provides you the flexibility to pick and choose the sample rates you want by carefully selecting `l` and `m`, the interpolation and decimation factors, that result in the final fractional decimation. The following figure shows the signal after applying the rate change filter `hm` to the original signal.



See Also

`mfilt.firsrc` | `mfilt.firinterp` | `mfilt.firdecim`

Introduced in R2011a

mfilt.firtdecim

Direct-form transposed FIR filter

Compatibility

`mfilt.firtdecim` will be removed in a future release. Use dsp.FIRDecimator instead.

Syntax

```
hm = mfilt.firtdecim(m)
hm = mfilt.firtdecim(m,num)
```

Description

`hm = mfilt.firtdecim(m)` returns a polyphase decimator `mfilt` object `hm` based on a direct-form transposed FIR structure with a decimation factor of `m`. A lowpass Nyquist filter of gain 1 and cutoff frequency of π/`m` is the default.

`hm = mfilt.firtdecim(m,num)` uses the coefficients specified by `num` for the decimation filter. `num` is a vector containing the coefficients of the transposed FIR lowpass filter used for decimation. If omitted, a lowpass Nyquist filter with gain of 1 and cutoff frequency of π/`m` is the default.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
num	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When num is not provided as an input, firtdecim uses a lowpass Nyquist filter with gain equal to l and cutoff frequency equal to π/m by default. The default length for the Nyquist filter is 24*m. Therefore, each polyphase filter component has length 24.
m	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for m it defaults to 2.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating mfilt.firtdecim objects. The next table describes each property for an mfilt.firtdecim filter object.

Name	Values	Description
Arithmetic	Double, single, fixed	Specifies the arithmetic the filter uses to process data while filtering.
DecimationFactor	Integer	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer.
FilterStructure	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of mfilt object. Also describes the signal flow for the filter object.
InputOffset	Integers	Contains a value derived from the number of input samples and the decimation factor — InputOffset = mod(length(nx),m) where

Name	Values	Description
		nx is the number of input samples that have been processed so far and m is the decimation factor.
Numerator	Vector	Vector containing the coefficients of the FIR lowpass filter used for decimation.
PersistentMemory	[false], true	Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory set to false returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to true allows you to modify the States, InputOffset, and PolyphaseAccum properties.
PolyphaseAccum	Double, single [0]	The idea behind having both PolyphaseAccum and Accum is to differentiate between the adders in the filter that work in full precision at all times (PolyphaseAccum) from the adders in the filter that the user controls and that may introduce quantization effects when FilterInternals is set to SpecifyPrecision.
States	Double, single matching the filter arithmetic setting.	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the mfilt.firtdecim filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where `hm` is a filter.

For further information about the properties of this filter or any `mfilt` object, refer to "Multirate Filter Properties" on page 6-104.

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits. [32]	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — `DenAccumFracLength` and `NumAccumFracLength` — that let you set the precision for numerator and denominator operations separately.
`AccumWordLength`	Any integer number of bits [39]	Sets the word length used to store data in the accumulator.
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
`CoeffWordLength`	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
`FilterInternals`	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.

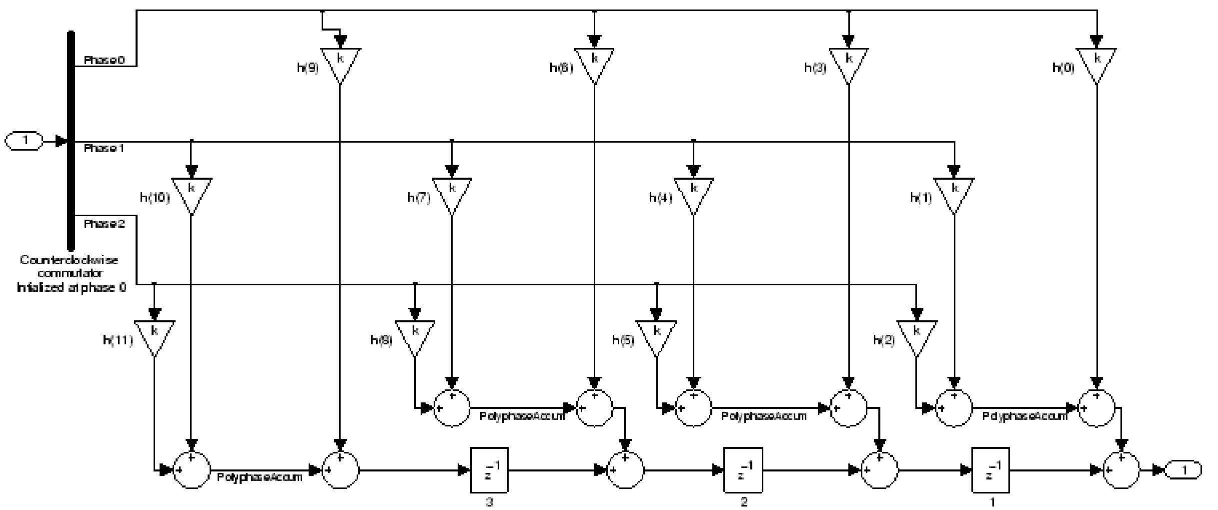
Name	Values	Description
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
`InputWordLength`	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
`NumFracLength`	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
`OutputFracLength`	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of `OutputFracLength` when you set `FilterInternals` to `SpecifyPrecision`.
`OutputWordLength`	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting `FilterInternals` to `SpecifyPrecision`.
`OverflowMode`	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
`PolyphaseAccum`	`fi` object with zeros to start	Differentiates between the adders in the filter that work in full precision at all times (`PolyphaseAccum`) and the adders in the filter that the user controls and that may introduce quantization effects when `FilterInternals` is set to `SpecifyPrecision`.

Name	Values	Description
`RoundMode`	[`convergent`], `ceil`, `fix`, `floor`, `nearest`, `round`	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `ceil` - Round toward positive infinity. • `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • `fix` - Round toward zero. • `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
`Signed`	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`States`	`fi` object	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation. For information about the ordering of the states, refer to the filter structure section.

Filter Structure

To provide sample rate changes, `mfilt.firtdecim` uses the following structure. At the input you see a commutator that operates counterclockwise, moving from position 0 to position 2, position 1, and back to position 0 as input samples enter the filter. To keep track of the position of the commutator, the `mfilt` object uses the property `InputOffset` which reports the current position of the commutator in the filter.

The following figure details the signal flow for the direct form FIR filter implemented by `mfilt.firtdecim`.



Notice the order of the states in the filter flow diagram. States 1 through 3 appear in the following diagram at each delay element. State 1 applies to the third delay element in phase 2. State 2 applies to the second delay element in phase 2. State 3 applies to the first delay element in phase 2. When you provide the states for the filter as a vector to the `States` property, the above description explains how the filter assigns the states you specify.

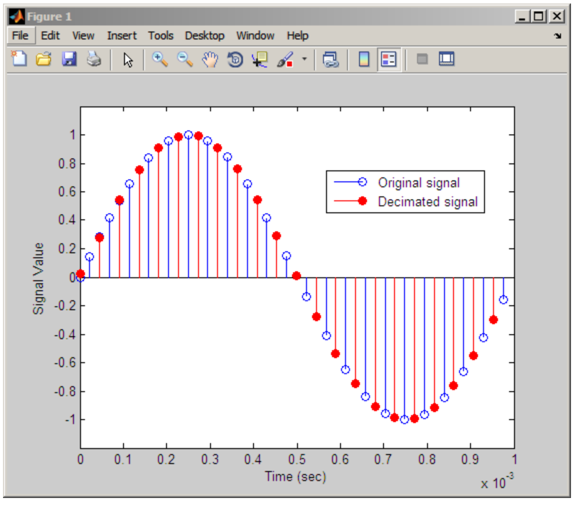
In property value form, the states for a filter `hm` are

```
hm.states=[1:3];
```

Examples

Demonstrate decimating an input signal by a factor of 2, in this case converting from 44.1 kHz down to 22.05 kHz. In the figure shown following the code, you see the results of decimating the signal.

```
m = 2;                    % Decimation factor.
hm = mfilt.firtdecim(m); % Use the default filter coeffs.
fs = 44.1e3;              % Original sample freq: 44.1 kHz.
n = 0:10239;              % 10240 samples, 0.232 second long signal
x  = sin(2*pi*1e3/fs*n); % Original signal--sinusoid at 1 kHz.
y = filter(hm,x);         % 5120 samples, 0.232 seconds.
stem(n(1:44)/fs,x(1:44)) % Plot original sampled at 44.1 kHz.
axis([0 0.001 -1.2 1.2]);
hold on                   % Plot decimated signal (22.05 kHz) in red
stem(n(1:22)/(fs/m),y(13:34),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value');
legend('Original signal','Decimated signal','location','best');
```



See Also

`mfilt.firdecim` | `mfilt.firsrc` | `mfilt.cicdecim`

Introduced in R2011a

mfilt.holdinterp

FIR hold interpolator

Compatibility

`mfilt.holdinterp` will be removed in a future release. Use dsp.CICInterpolator (with `NumSections` = 1) instead.

Syntax

```
hm = mfilt.holdinterp(l)
```

Description

`hm = mfilt.holdinterp(l)` returns the object `hm` that represents a hold interpolator with the interpolation factor `l`. To work, `l` must be an integer. When you do not include `l` in the calling syntax, it defaults to 2. To perform interpolation by noninteger amounts, use one of the fractional interpolator objects, such as `mfilt.firsrc`.

When you use this hold interpolator, each sample added to the input signal between existing samples has the value of the most recent sample from the original signal. Thus you see something like a staircase profile where the interpolated samples form a plateau between the previous and next original samples. The example demonstrates this profile clearly. Compare this to the interpolation process for other interpolators in the toolbox, such as `mfilt.linearinterp`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
`l`	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for `l` it defaults to 2.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.holdinterp` objects. The next table describes each property for an `mfilt.interp` filter object.

Name	Values	Description
`Arithmetic`	`Double`, `single`, `fixed`	Specifies the arithmetic the filter uses to process data while filtering.
`FilterStructure`	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object.
`InterpolationFactor`	Integer	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer.
`PersistentMemory`	'`false`' or '`true`'	Determines whether the filter states are restored to zero for each filtering operation.

Name	Values	Description
States	Double or single array	Filter states. states defaults to a vector of zeros that has length equal to nstates (hm). Always available, but visible in the display only when PersistentMemory is true.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the mfilt.holdinterp filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where hm is a filter.

For further information about the properties of this filter or any mfilt object, refer to "Multirate Filter Properties" on page 6-104.

Name	Values	Description
Arithmetic	Double, single, fixed	Specifies the arithmetic the filter uses to process data while filtering.
FilterStructure	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of mfilt object.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data.

Name	Values	Description
`InterpolationFactor`	Integer	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer.
`PersistentMemory`	`'false'` or `'true'`	Determine whether the filter states get restored to zero for each filtering operation
`States`	`fi` object	Contains the filter states before, during, and after filter operations. For hold interpolators, the states are always empty — hold interpolators do not have states. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation.

Filter Structure

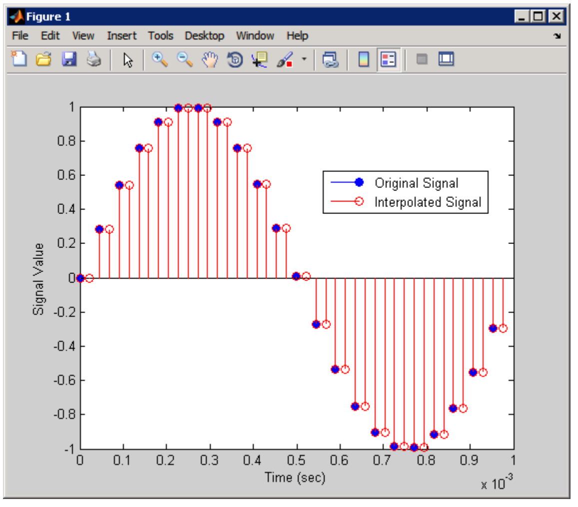
Hold interpolators do not have filter coefficients and their filter structure is trivial.

Examples

To see the effects of hold-based interpolation, interpolate an input sine wave from 22.05 to 44.1 kHz. Note that each added sample retains the value of the most recent original sample.

```
l = 2;                              % Interpolation factor
hm = mfilt.holdinterp(l);
fs = 22.05e3;                       % Original sample freq: 22.05 kHz.
n = 0:5119;                         % 5120 samples, 0.232 second long signal
x  = sin(2*pi*1e3/fs*n);            % Original signal, sinusoid at 1 kHz
y = filter(hm,x);                   % 10240 samples, still 0.232 seconds
stem(n(1:22)/fs,x(1:22),'filled')   % Plot original sampled at
                                    % 22.05 kHz
hold on                             % Plot interpolated signal (44.1 kHz)
stem(n(1:44)/(fs*l),y(1:44),'r')
legend('Original Signal','Interpolated Signal','Location','best');
xlabel('Time (sec)');ylabel('Signal Value')
```

The following figure shows clearly the step nature of the signal that comes from interpolating the signal using the hold algorithm approach. Compare the output to the linear interpolation used in `mfilt.linearinterp`.



See Also

`mfilt.linearinterp` | `mfilt.firinterp` | `mfilt.firsrc` | `mfilt.cicinterp`

Introduced in R2011a

mfilt.iirdecim

IIR decimator

Note: `mfilt.iirdecim` will be removed in a future release. Use dsp.IIRHalfbandDecimator instead.

Syntax

```
hm = mfilt.iirdecim(c1,c2,...)
```

Description

`hm = mfilt.iirdecim(c1,c2,...)` constructs an IIR decimator filter given the coefficients specified in the cell arrays `c1`, `c2`, and so on. The resulting IIR decimator is a polyphase IIR filter where each phase is a cascade allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadeallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirdecim` interprets them same way as `mfilt.cascadeallpass`.

The following exception applies to interpreting the contents of a cell array — if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and one element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadeallpass` section. This exception case occurs with quasi-linear phase IIR decimators.

Although the first example shows how to construct an IIR decimators explicitly, one usually constructs an IIR decimators filter as a result of designing an decimators, as shown in the subsequent examples.

Examples

When the coefficients are known, you can construct the IIR decimator directly using `mfilt.iirdecim`. For example, if the filter's coefficients are [0.6 0.5] for the first phase in the first stage, 0.7 for the second phase in the first stage and 0.8 for the third phase in the first stage; as well as 0.5 for the first phase in the second stage and 0.4 for the second phase in the second stage, construct the filter as shown here.

```
Hm = mfilt.iirdecim({[0.6 0.5] 0.7 0.8},{0.5 0.4})
```

Also refer to the “Quasi-Linear Phase Halfband and Dyadic Halfband Designs” section of the “IIR Polyphase Filter Design” example, iirallpassdemo example.

When the coefficients are not known, use the approach given by the following set of examples. Start by designing an elliptic halfband decimator with a decimation factor of 2. The example specifies the optional sampling frequency argument.

```
tw = 100;   % Transition width of filter.
ast = 80;   % Stopband attenuation of filter.
fs = 2000;  % Sampling frequency of signal to filter.
m = 2;      % Decimation factor.
hm = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

`hm` contains the specifications for a decimator defined by `tw`, `ast`, `m`, and `fs`.

Use the specification object `hm` to design a `mfilt.iirdecim` filter object.

```
d = design(hm,'ellip','filterstructure','iirdecim');
% Note that realizemdl requires Simulink
realizemdl(d)     % Build model of the filter.
```

Designing a linear phase decimator is similar to the previous example. In this case, design a halfband linear phase decimator with decimation factor of 2.

```
tw  = 100;  % Transition width of filter.
ast = 60;   % Stopband attenuation of filter.
fs  = 2000; % Sampling frequency of signal to filter.
m = 2;      % Decimation factor.
```

Create a specification object for the decimator.

```
hm = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

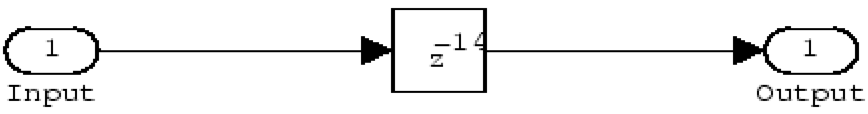
Finally, design the filter d.

```
d = design(hm,'iirlinphase','filterstructure','iirdecim');
% Note that realizemdl requires Simulink
realizemdl(d)  % Build model of the filter.
```

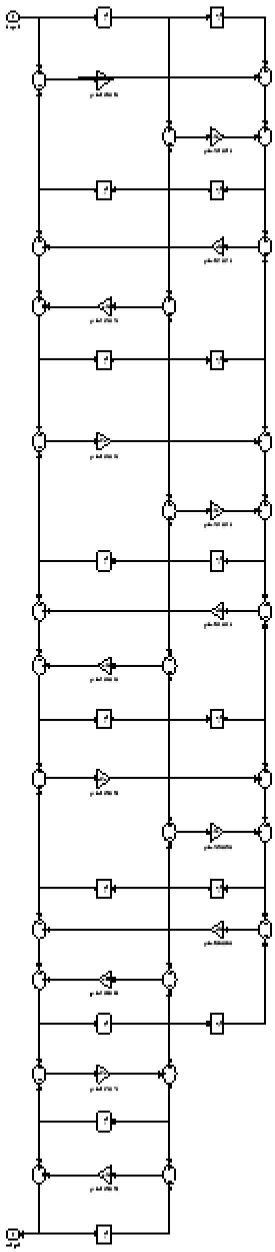
The filter implementation appears in this model, generated by `realizemdl` and Simulink.

Given the design specifications shown here

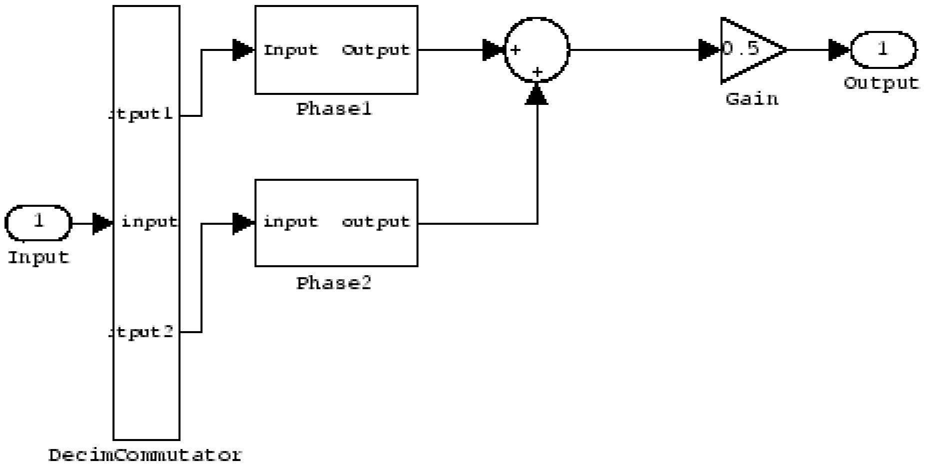
the first phase is a delay section with 0s and a 1 for coefficients and the second phase is a linear phase decimator, shown in the next models.



Phase 1 model



Phase 2 model



Overall model

For more information about Multirate Filter Constructors see the “Getting Started with Multirate Filter (MFILT) Objects” example, mfiltgettingstarteddemo.

See Also

`dfilt.cascadeallpass` | `mfilt` | `mfilt.iirinterp` | `mfilt.iirwdfdecim`

Introduced in R2011a

mfilt.iirinterp

IIR interpolator

Note: `mfilt.iirinterp` will be removed in a future release. Use dsp.IIRHalfbandInterpolator instead.

Syntax

```
hm = mfilt.iirinterp(c1,c2,...)
```

Description

`hm = mfilt.iirinterp(c1,c2,...)` constructs an IIR interpolator filter given the coefficients specified in the cell arrays C1, C2, etc.

The IIR interpolator is a polyphase IIR filter where each phase is a cascade allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadeallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirdecim` interprets them same way as `mfilt.cascadeallpass`.

The following exception applies to interpreting the contents of a cell array—if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and a unique element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadeallpass` section. This exception case occurs with quasi-linear phase IIR interpolators.

Although the first example shows how to construct an IIR interpolator explicitly, one usually constructs an IIR interpolator filter as a result of designing an interpolator, as shown in the subsequent examples.

Examples

When the coefficients are known, you can construct the IIR interpolator directly using `mfilt.iirinterp`. In the following example, a cascaded polyphase IIR interpolator filter is constructed using 2 phases for each of three stages. The coefficients are given below:

```
Phase1Sect1=0.0603;Phase1Sect2=0.4126; Phase1Sect3=0.7727;
Phase2Sect1=0.2160; Phase2Sect2=0.6044; Phase2Sect3=0.9239;
```

Next the filter is implemented by passing the above coefficients to `mfilt.iirinterp` as cell arrays, where each cell array represents a different phase.

```
Hm = mfilt.iirinterp({Phase1Sect1,Phase1Sect2,Phase1Sect3},...
{Phase2Sect1,Phase2Sect2,Phase2Sect3})

Hm =

        FilterStructure: 'IIR Polyphase Interpolator'
              Polyphase: Phase1: Section1: 0.0603
                                 Section2: 0.4126
                                 Section3: 0.7727
                         Phase2: Section1: 0.216
                                 Section2: 0.6044
                                 Section3: 0.9239
    InterpolationFactor: 2
       PersistentMemory: false
```

Also refer to the “Quasi-Linear Phase Halfband and Dyadic Halfband Designs” section of the “IIR Polyphase Filter Design” example, iirallpassdemo example.

When the coefficients are not known, use the approach given by the following set of examples. Start by designing an elliptic halfband interpolator with a interpolation factor of 2.

```
tw = 100;  % Transition width of filter.
ast = 80;   % Stopband attenuation of filter.
fs = 2000; % Sampling frequency of filter.
l  = 2;    % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

Specification object `d` stores the interpolator design specifics. With the details in `d`, design the filter, returning `hm`, an `mfilt.iirinterp` object. Use `hm` to realize the filter if you have Simulink installed.

```
hm = design(d,'ellip','filterstructure','iirinterp');
```

```
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

Designing a linear phase halfband interpolator follows the same pattern.

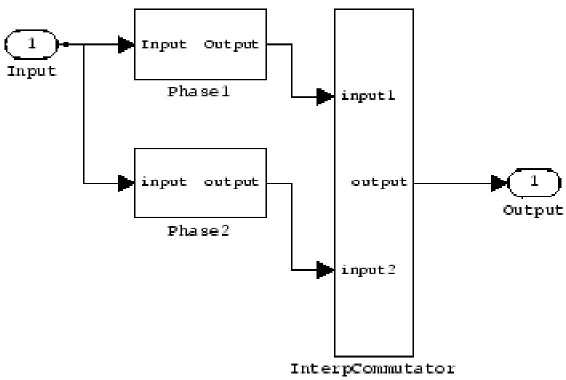
```
tw = 100;  % Transition width of filter.
ast= 60;   % Stopband attenuation of filter.
fs = 2000; % Sampling frequency of filter.
l = 2;     % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

`fdesign.interpolator` returns a specification object that stores the design features for an interpolator.

Now perform the actual design that results in an `mfilt.iirinterp` filter, `hm`.

```
hm = design(d,'iirlinphase','filterstructure','iirinterp');
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

The toolbox creates a Simulink model for `hm`, shown here. Phase1, Phase2, and InterpCommutator are all subsystem blocks.



For more information about Multirate Filter Constructors see the “Getting Started with Multirate Filter (MFILT) Objects” example, mfiltgettingstarteddemo.

See Also

`dfilt.cascadeallpass` | `mfilt` | `mfilt.iirdecim` | `mfilt.iirwdfinterp`

Introduced in R2011a

mfilt.iirwdfdecim

IIR wave digital filter decimator

Compatibility

`mfilt.iirwdfdecim` will be removed in a future release. Use dsp.IIRHalfbandDecimator instead.

Syntax

```
hm = mfilt.iirwdfdecim(c1,c2,...)
```

Description

`hm = mfilt.iirwdfdecim(c1,c2,...)` constructs an IIR wave digital decimator given the coefficients specified in the cell arrays c1, c2, and so on. The IIR decimator `hm` is a polyphase IIR filter where each phase is a cascade wave digital allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadewdfallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirwdfdecim` interprets them same way as `mfilt.cascadewdfallpass`.

The following exception applies to interpreting the contents of a cell array — if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and one element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadewdfallpass` section. This exception occurs with quasi-linear phase IIR decimators.

Usually you do not construct IIR wave digital filter decimators explicitly. Instead, you obtain an IIR wave digital filter decimator as a result of designing a halfband decimator. The first example in the following section illustrates this case.

Examples

Design an elliptic halfband decimator with a decimation factor equal to 2. Both examples use the `iirwdfdecim` filter structure (an input argument to the `design` method) to design the final decimator.

The first portion of this example generates a filter specification object `d` that stores the specifications for the decimator.

```
tw  = 100;  % Transition width of filter to design, 100 Hz.
ast = 80;   % Stopband attenuation of filter 80 dB.
fs  = 2000; % Sampling frequency of the input signal.
m   = 2;    % Decimation factor.
d = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

Now perform the actual design using `d`. Filter object `hm` is an `mfilt.iirwdfdecim` filter.

```
Hm = design(d,'ellip','FilterStructure','iirwdfdecim');
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

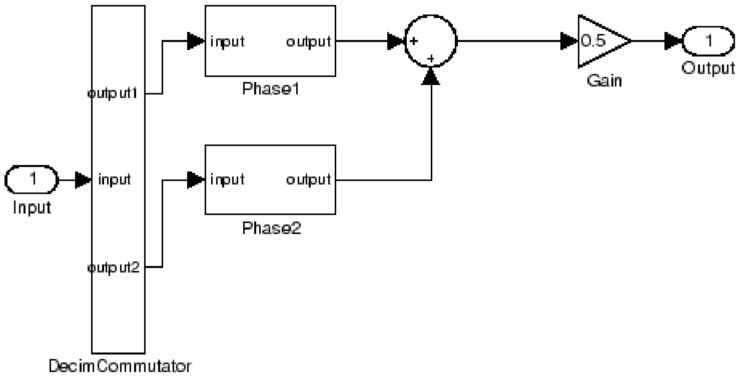
Design a linear phase halfband decimator for decimating a signal by a factor of 2.

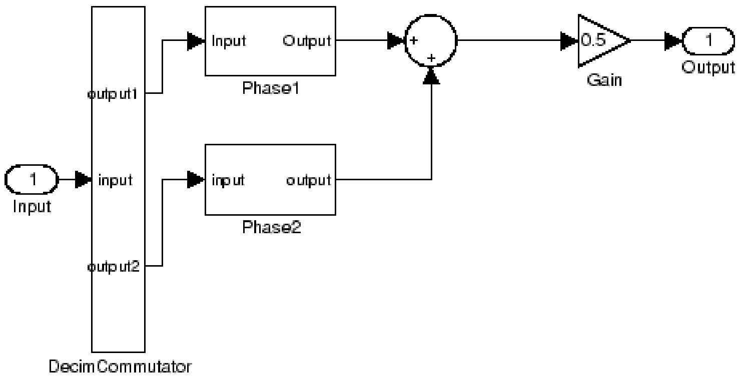
```
tw  = 100;  % Transition width of filter, 100 Hz.
ast = 60;   % Filter stopband attenuation = 80 dB
fs  = 2000; % Input signal sampling frequency.
m   = 2;    % Decimation factor.
d = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

Use `d` to design the final filter `hm`, an `mfilt.iirwdfdecim` object.

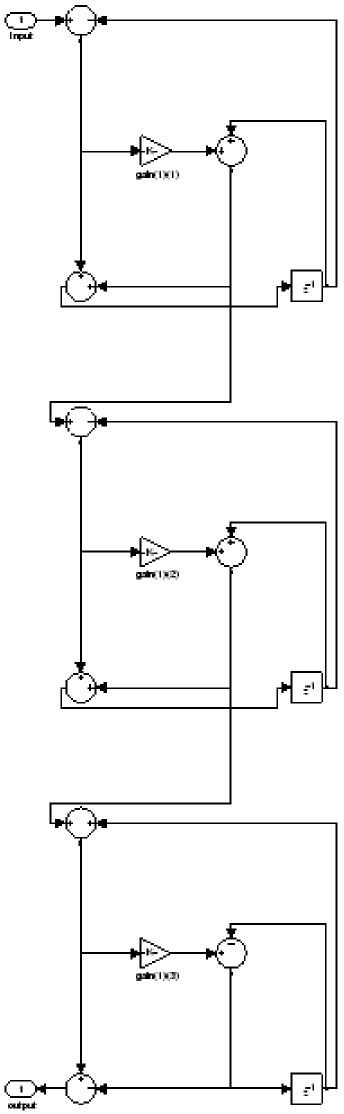
```
hm = design(d,'iirlinphase','filterstructure',...
'iirwdfdecim');
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

The models that `realizemdl` returns for each example appear below. At this level, the realizations of the filters are identical. The differences appear in the subsystem blocks Phase1 and Phase2.

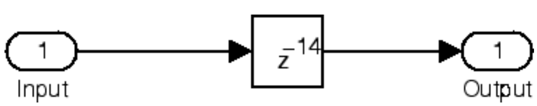




This is the Phase1 subsystem from the halfband model.



Phase1 subsystem from the linear phase model is less revealing—an allpass filter.



See Also

`dfilt.cascadewdfallpass` | `mfilt` | `mfilt.iirdecim` | `mfilt.iirwdfinterp`

Introduced in R2011a

mfilt.iirwdfinterp

IIR wave digital filter interpolator

Compatibility

`mfilt.iirwdfinterp` will be removed in a future release. Use dsp.IIRHalfbandInterpolator instead.

Syntax

```
hm = mfilt.iirwdfinterp(c1,c2,...)
```

Description

`hm = mfilt.iirwdfinterp(c1,c2,...)` constructs an IIR wave digital interpolator given the coefficients specified in the cell arrays `c1`, `c2`, and so on. The IIR interpolator `hm` is a polyphase IIR filter where each phase is a cascade wave digital allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadewdfallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirwdfinterp` interprets them same way as `mfilt.cascadewdfallpass`.

The following exception applies to interpreting the contents of a cell array — if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and one element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadewdfallpass` section. This exception occurs with quasi-linear phase IIR interpolators.

Usually you do not construct IIR wave digital filter interpolators explicitly. Rather, you obtain an IIR wave digital interpolator as a result of designing a halfband interpolator. The first example in the following section illustrates this case.

Examples

Design an elliptic halfband interpolator with interpolation factor equal to 2. At the end of the design process, hm is an IIR wave digital filter interpolator.

```
tw  = 100;  % Transition width of filter, 100 Hz.
ast = 80;   % Stopband attenuation of filter, 80 dB.
fs  = 2000; % Sampling frequency after interpolation.
l   = 2;    % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

The specification object `d` stores the interpolator design requirements. Now use `d` to design the actual filter `hm`.

```
hm = design(d,'ellip','filterstructure','iirwdfinterp');
```

If you have Simulink installed, you can realize your filter as a model built from DSP System Toolbox blocks.

```
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

For variety, design a linear phase halfband interpolator with an interpolation factor of 2.

```
tw  = 100;  % Transition width of filter, 100 Hz.
ast = 80;   % Stopband attenuation of filter, 80 dB.
fs  = 2000; % Sampling frequency after interpolation.
l   = 2;    % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

Now perform the actual design process with `d`. Filter `hm` is an IIR wave digital filter interpolator. As in the previous example, `realizemdl` returns a Simulink model of the filter if you have Simulink installed.

```
hm = design(d,'iirlinphase','filterstructure',...
'iirwdfinterp');
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

See Also

`dfilt.cascadewdfallpass` | `mfilt.iirinterp` | `mfilt.iirwdfdecim`

Introduced in R2011a

mfilt.linearinterp

Linear interpolator

Compatibility

`mfilt.linearinterp` will be removed in a future release. Use idsp.CICInterpolator (with `NumSections` = 2) instead.

Syntax

```
hm = mfilt.linearinterp(l)
```

Description

`hm = mfilt.linearinterp(l)` returns an FIR linear interpolator `hm` with an integer interpolation factor `l`. Provide `l` as a positive integer. The default value for the interpolation factor is 2 when you do not include the input argument `l`.

When you use this linear interpolator, the samples added to the input signal have values between the values of adjacent samples in the original signal. Thus you see something like a smooth profile where the interpolated samples continue a line between the previous and next original samples. The example demonstrates this smooth profile clearly. Compare this to the interpolation process for `mfilt.holdinterp`, which creates a stairstep profile.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```
- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Input Arguments

The following table describes the input argument for `mfilt.linearinterp`.

Input Argument	Description
`l`	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for `l` it defaults to 2.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.linearinterp` objects. The next table describes each property for an `mfilt.linearinterp` filter object.

Name	Values	Description
`Arithmetic`	`Double`, `single`, `fixed`	Specifies the arithmetic the filter uses to process data while filtering.
`FilterStructure`	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object.
`InterpolationFactor`	Integer	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer.
`PersistentMemory`	'`false`' or '`true`'	Determine whether the filter states get restored to zero for each filtering operation

Name	Values	Description
`States`	Double or single array	Filter states. `states` defaults to a vector of zeros that has length equal to `nstates(hm)`. Always available, but visible in the display only when `PersistentMemory` is `true`.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the `mfilt.holdinterp` filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where `hm` is a filter.

For further information about the properties of this filter or any `mfilt` object, refer to "Multirate Filter Properties" on page 6-104.

Name	Values	Description
`AccumFracLength`	Any positive or negative integer number of bits. Depends on L. [29 when L=2]	Specifies the fraction length used to interpret data output by the accumulator.
`AccumWordLength`	Any integer number of bits [33]	Sets the word length used to store data in the accumulator.
`Arithmetic`	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
`CoeffAutoScale`	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change

Name	Values	Description
		the NumFracLength property value to specify the precision used.
CoeffWordLength	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
OutputFracLength	Any positive or negative integer number of bits [29]	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilterInternals to SpecifyPrecision.
OutputWordLength	Any integer number of bits [33]	Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.

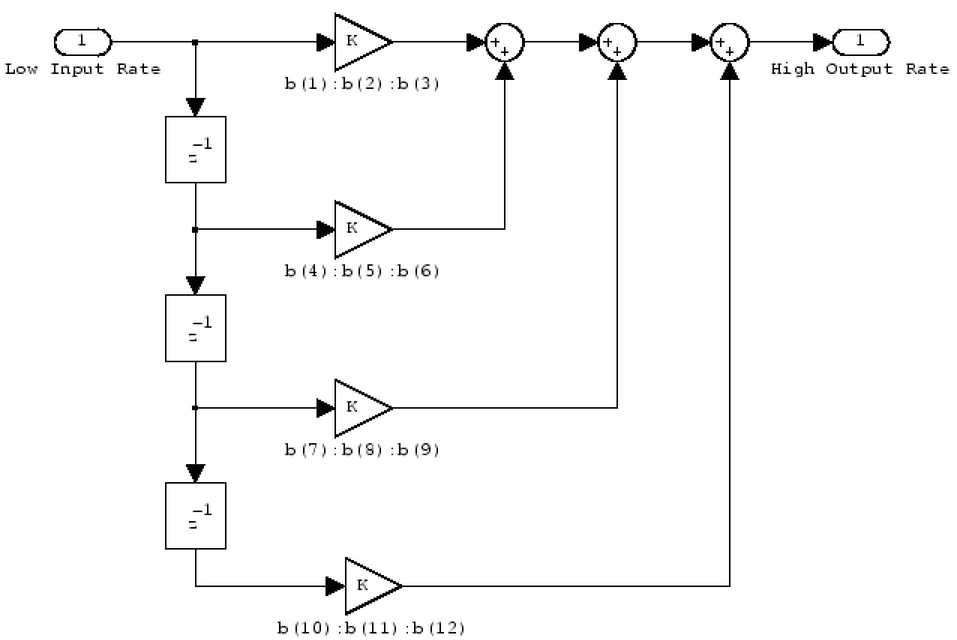
Name	Values	Description
`OverflowMode`	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.

Name	Values	Description
RoundMode	[convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.

Name	Values	Description
`States`	`fi` object	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Designer documentation. For information about the ordering of the states, refer to the filter structure in the following section.

Filter Structure

Linear interpolator structures depend on the FIR filter you use to implement the filter. By default, the structure is direct-form FIR.

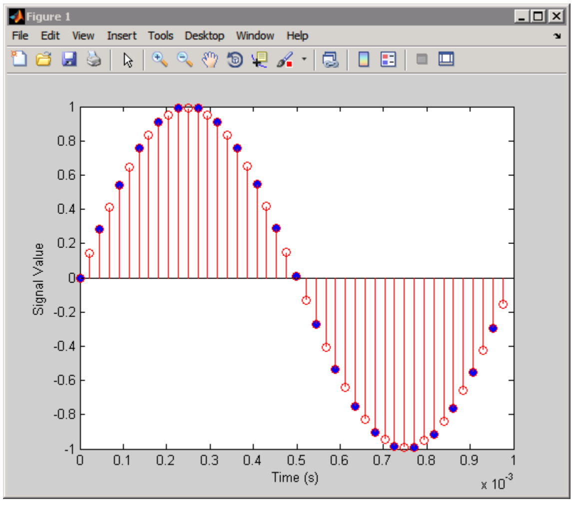


Examples

Interpolation by a factor of 2 (used to convert the input signal sampling rate from 22.05 kHz to 44.1 kHz).

```
l = 2;                          % Interpolation factor
hm = mfilt.linearinterp(l);
fs = 22.05e3;                   % Original sample freq: 22.05 kHz.
n = 0:5119;                     % 5120 samples, 0.232 second long signal
x  = sin(2*pi*1e3/fs*n);        % Original signal, sinusoid at 1 kHz
y = filter(hm,x);               % 10240 samples, still 0.232 seconds
stem(n(1:22)/fs,x(1:22),'filled') % Plot original sampled at
                                  % 22.05 kHz
hold on                         % Plot interpolated signal (44.1
                                % kHz) in red
stem(n(1:44)/(fs*l),y(2:45),'r')
xlabel('Time (s)');ylabel('Signal Value')
```

Using linear interpolation, as compared to the hold approach of `mfilt.holdinterp`, provides greater fidelity to the original signal.



See Also

`mfilt.holdinterp` | `mfilt.firinterp` | `mfilt.firsrc` | `mfilt.cicinterp`

Introduced in R2011a

midicallback

Call function handle when MIDI controls change value

Compatibility

The `midicallback` function will be removed from DSP System Toolbox in a future release. Existing instances of the function continue to run. For new code, use the `midicallback` function from Audio System Toolbox instead.

Syntax

```
oldfh = midicallback(h,newfh)
oldfh = midicallback(h,[])
fh= midicallback(h)
```

Description

`oldfh = midicallback(h,newfh)` sets `newfh` as the function handle to be called when `h` changes value, and returns the previous function handle, `oldfh`.

`oldfh = midicallback(h,[])` clears the function handle.

`fh= midicallback(h)` returns the current function handle.

Examples

Interactively Read MIDI Controls

Use the `midicallback` command with an anonymous function to interactively read MIDI controls.

```
h = midicontrols;
midicallback(h,@(h)disp(midiread(h)));
```

```
% Now move any control on the default MIDI device.
   0.6587
   0.6429
   0.6349
   0.6270
   0.6190
   0.6111
   0.6032
   0.5952
clear h
```

Input Arguments

h — Object that listens to the controls on a MIDI device

object

h is an object that listens to the controls on a MIDI device.

newfh — new function handle

function handle

The new function handle, which is set as the function handle to be called when h changes value. For information on what function handles are, see “Function Handles” (MATLAB).

Example: @myFunction

Output Arguments

oldfh — Old function handle

function handle

The function handle set by the previous call to midicallback.

Example: @myFunction

fh — Current function handle

function handle

The function handle set by the current call to midicallback.

Example: @myFunction

See Also

See Also

`midicontrols` | `midiid` | `midiread` | `midisync` | `setpref`

Introduced in R2013b

midicontrols

Open a group of MIDI controls for reading

Compatibility

The `midicontrols` function will be removed from DSP System Toolbox in a future release. Existing instances of the function continue to run. For new code, use the `midicontrols` function from Audio System Toolbox instead.

Syntax

```
h = midicontrols
h = midicontrols(ControlNumbers)
h = midicontrols(ControlNumbers,InitialValues)
h = midicontrols( ___ ,'MIDIDevice',devicename)
h = midicontrols( ___ ,'OutputMode',mode)
```

Description

`h = midicontrols` returns an object that responds to any control on the default MIDI device. Calling `midiread` with the object, returns the double scalar value of the MIDI control that recently moved after the object was created. The value is normally in the range [0 1]. See `OutputMode` for an alternative. This object can only determine a control's value if the control is moved after the `midicontrols` object is created. If `midiread` is called before the control is moved, `midiread` returns a default initial value of `0`.

`h = midicontrols(ControlNumbers)` returns an object that responds to the MIDI controls specified by `ControlNumbers`. Calling `midiread` with the object, returns a double array of the same shape as `ControlNumbers`. Use `midiid` to interactively identify the control number of individual MIDI controls.

`h = midicontrols(ControlNumbers,InitialValues)` returns an object that uses the specified `InitialValues` when controls are not moved after the object is created.

Because initial values are quantized for the underlying MIDI protocol, sometimes `midiread` returns an initial value that is slightly different from `InitialValues`.

`h = midicontrols(___,'MIDIDevice',devicename)` specifies the MIDI device to which the object responds. Use `midiid` to interactively identify the name of a specific MIDI device. If you do not specify the `'MIDIDevice'` name-value pair, the default MIDI device is used. The MATLAB preference 'midi' `'DefaultDevice'` determines the default device.

`h = midicontrols(___,'OutputMode',mode)` specifies the range of values returned by `midiread` and accepted as `InitialValues`. This name-value pair is optional, and you can insert it only at the end of the argument list.

Examples

Respond to any Control on the Default Device

Create the object, and read from it:

```
h =midicontrols
midiread(h)
```

Move one of the controls, and read the data:

```
midiread(h)
```

Respond to a Specific Control

Make the object respond to a specific control:

```
h = midicontrols(1081);
```

Use Control Numbers and an Initial Value

Return a square array, with initial value of 0.5:

```
h = midicontrols([1081 1083; 1082 1084], 0.5);
```

Set Mode to raw, and Set an Initial Value

Return a square array, with the raw initial value of 63:

```
h = midicontrols([1081 1083; 1082 1084], 63, 'OutputMode', 'rawmidi');
```

Set the Default MIDI Device

Assume your MIDI device is a Behringer BCF2000. Set the default device this way:

```
setpref midi DefaultDevice BCF2000
```

This preference persists across MATLAB sessions, so you do not need to set it again unless you want to change devices.

Use Both ControlNumbers and DeviceName

Respond to control 1001 on a Behringer BCF2000:

```
h = midicontrols(1001, 'MIDIDevice', 'BCF2000');
```

Input Arguments

`ControlNumbers` — Identifying MIDI controls

integer values

`ControlNumbers` are integer-valued double-precision numbers. Each control on the MIDI device has a specific integer assigned to it by the device manufacturer. If `ControlNumbers` is [ ], then the `midicontrols` object responds to any control on the MIDI device. As a result, `midiread` returns a double scalar.

Example: `1081`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`InitialValues` — Initial value of MIDI control

any numeric value within range

`InitialValues` must either be an array of the same size as `ControlNumbers` or a scalar. If you do not specify `InitialValues`, the default initial value is `0`. Typically, initial values must be in the range [0 1]. However, if you specify `'rawmidi'` as `OutputMode`, the `InitialValues` range is between 0 and 127. Because the initial values are quantized for the underlying MIDI protocol, sometimes `midiread` returns an initial value that is slightly different from `InitialValues`.

Example: `0.3 or [0 0.3 0.6]`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`devicename` — Name of device

character string

`devicename` is a character string assigned by the device manufacturer or the host operating system. The specified `devicename` can be a substring of the device's exact name. If you do not specify the '`MIDIDevice`' name-value pair, the default MIDI device is used. The MATLAB preference '`midi`', '`DefaultDevice`' determines the default device.

If you do not set the MATLAB preference, the host operating system chooses the default device in an unspecified way. Some systems have virtual (ie, software) MIDI devices installed. Even if you have only one hardware MIDI device attached to your system, the system may not choose it, which can cause confusion. As a best practice, use `midiid` to identify the name of the device you want. Then use `setpref` to set it as the default device.

Example: `'BCF2000 MIDI 1'`

Data Types: `char`

`mode` — Mode of output

character string

`mode` is a string and must be one of `'normalized'` or `'rawmidi'`. In normalized mode, values are in the range [0 1]. Also, initial values are quantized for the underlying MIDI protocol. In the raw MIDI mode, values are integers in the range [0 127], and the quantization of the initial values is not performed. The default of this name-value pair is '`normalized`'.

Example: `'rawmidi'`

Data Types: `char`

Output Arguments

`h` — Object that listens to the controls on a MIDI device

object

`h` is an object that listens to the controls on a MIDI device.

See Also

See Also

`midicallback` | `midiid` | `midiread` | `midisync` | `setpref`

Introduced in R2013b

midiid

Interactively identify MIDI control

Compatibility

The midiid function will be removed from DSP System Toolbox in a future release. Existing instances of the function continue to run. For new code, use the midiid function from Audio System Toolbox instead.

Syntax

```
[ctl device] = midiid
```

Description

`[ctl device] = midiid` returns the control number and device name of the MIDI control moved by the user. Call the function at the MATLAB command prompt and then move the control you want to identify on the MIDI control surface. The function detects which control you move and returns the corresponding control number and device name that specify that control.

Examples

Retrieve Control Number and Device Name

Call `midiid`.

```
[ctl,dev] = midiid;
Move the control you wish to identify; type ^C to abort.
Waiting for control message...

ctl =
1002
dev =
```

`nanoKONTROL`

Output Arguments

`ctl` — Number associated with control being moved

integer values

`ctl` is an integer-valued double-precision number. Each control on the MIDI device has a specific integer assigned to it by the device manufacturer.

Example: `1003`

Data Types: `double`

`device` — Name of device

character string

`device` is a character string assigned by the device manufacturer or the host operating system.

Example: `'nanoKontrol'`

Data Types: `char`

See Also

See Also

`midicallback` | `midicontrols` | `midiread` | `midisync` | `setpref`

Introduced in R2013b

midiread

Return most recent value of MIDI controls

Compatibility

The `midiread` function will be removed from DSP System Toolbox in a future release. Existing instances of the function continue to run. For new code, use the `midiread` function from Audio System Toolbox instead.

Syntax

```
v = midiread(h)
```

Description

`v = midiread(h)` returns the most recent value of the MIDI controls associated with `midicontrols` object, `h`. You must create `h` first before it can determine the values of its MIDI controls if they are moved. Calling `midiread` before the controls are moved, returns the initial values specified to `midicontrols`. In this case, when `h` is created. (or `0` if no initial values are specified).

Examples

Read Control Values

```
h = midicontrols;
v = midiread(h);
```

Input Arguments

h — Object that listens to the controls on a MIDI device

object

h is an object that listens to the controls on a MIDI device.

Output Arguments

v — Most recent value of MIDI controls

any numeric value

The output value depends on the OutputMode specified by midicontrols when h is created. If you specify that the OutputMode is normalized, then the midiread returns output values in the range [0 1]. Also, initial values are quantized and may be slightly different from those specified by midicontrols.

If you specify the mode as rawmidi, then midiread returns integer values in the range [0 127], and no quantization is required. If you do not specify the OutputMode, the default is normalized.

Example: 0.3 or [0 0.3 0.6]

Data Types: double | uint8

See Also

See Also

midicallback | midicontrols | midiid | midisync | setpref

Introduced in R2013b

midisync

Send values to MIDI controls to synchronize

Compatibility

The `midisync` function will be removed from DSP System Toolbox in a future release. Existing instances of the function continue to run. For new code, use the `midisync` function from Audio System Toolbox instead.

Syntax

```
midisync(h)
midisync(h,Values)
```

Description

`midisync(h)` sends the initial values specified by `midicontrols`. `h` is created by the MIDI controls associated with the `midicontrols` object, `h`. You can use `midisync` with bidirectional MIDI devices that can both send and receive messages, and move a control in response to a received message. For example, when a `midicontrols` object is first created, it is often helpful to move the MIDI control to match the initial value of the object. Many MIDI devices are not bidirectional, and calling `midisync` with a unidirectional device has no effect. `midisync` cannot tell whether a value is successfully sent to a device or even whether the device is bidirectional. Therefore, no errors or warnings are generated if sending a value fails.

`midisync(h,Values)` sends `Values` to the MIDI controls associated with the `midicontrols` object, `h`. `Values` must follow the same rules as `InitialValue` arguments of `midicontrols`.

Examples

Send a Slider Change to MIDI Control

```
midisync(h, get(slider, 'Value'))
```

Create a GUI with a Single Slider, and Synchronize it with a MIDI Control

When you move either control, the other control tracks it. The resulting value appears on the command prompt.

```
function trivialmidigui(controlnum,DEVICENAME)
    slider = uicontrol('Style','slider');
    mc = midicontrols(controlnum,'MIDIDevice',DEVICENAME);
    midisync(mc);
    set(slider,'Callback',@slidercb);
    midicallback(mc, @mccb);

    function slidercb(slider,~)
        val = get(slider,'Value');
        midisync(mc, val);
        disp(val);
    end

    function mccb(mc)
        val = midiread(mc);
        set(slider,'Value',val);
        disp(val);
    end
end
```

Input Arguments

`h` — Object that listens to the controls on a MIDI device

object

`h` is an object that listens to the controls on a MIDI device.

`Values` — Values sent to select MIDI control

any numeric value in range

`Values` must either be an array the same size as `ControlNumbers` from `midicontrols` or a scalar. If you do not specify `Values`, the default value is whatever the

InitialValues is from midicontrols. Typically, values must normally be in the range [0 1]. However, if you specify 'rawmidi' as OutputMode of midicontrols, the Values range is between 0 and 127.

Example: 0.3 or [0 0.3 0.6]

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

See Also

See Also

midicallback | midicontrols | midiid | midiread | setpref

Introduced in R2013b

minimizecoeffwl

Minimum wordlength fixed-point filter

Syntax

```
Hq = minimizecoeffwl(Hd)
Hq = minimizecoeffwl(Hd,...,'NoiseShaping',NSFlag)
Hq = inimizecoeffwl(Hd,...,'NTrials',N)
Hq = minimizecoeffwl(Hd,...'Apasstol',Apasstol)
Hq = minimizecoeffwl(Hd,...,'Astoptol',Astoptol)
Hq = minimizecoeffwl(Hd,...,'MatchrefFilter',RefFiltFlag)
```

Description

`Hq = minimizecoeffwl(Hd)` returns the minimum wordlength fixed-point filter object `Hq` that meets the design specifications of the single-stage or multistage FIR filter object `Hd`. `Hd` must be generated using `fdesign` and `design`. If `Hd` is a multistage filter object, the procedure minimizes the wordlength for each stage separately. `minimizecoeffwl` uses a stochastic noise-shaping procedure by default to minimize the wordlength. To obtain repeatable results on successive function calls, initialize the uniform random number generator `rand`.

`Hq = minimizecoeffwl(Hd,...,'NoiseShaping',NSFlag)` enables or disables the stochastic noise-shaping procedure in the minimization of the wordlength. By default `NSFlag` is `true`. Setting `NSFlag` to `false` minimizes the wordlength without using noise-shaping.

`Hq = inimizecoeffwl(Hd,...,'NTrials',N)` specifies the number of Monte Carlo trials to use in the minimization. `Hq` is the filter with the minimum wordlength among the `N` trials that meets the specifications in `Hd`. `'NTrials'` defaults to one.

`Hq = minimizecoeffwl(Hd,...'Apasstol',Apasstol)` specifies the passband ripple tolerance in dB. `'Apasstol'` defaults to `1e-4`.

`Hq = minimizecoeffwl(Hd,...,'Astoptol',Astoptol)` specifies the stopband tolerance in dB. `'Astoptol'` defaults to `1e-2`.

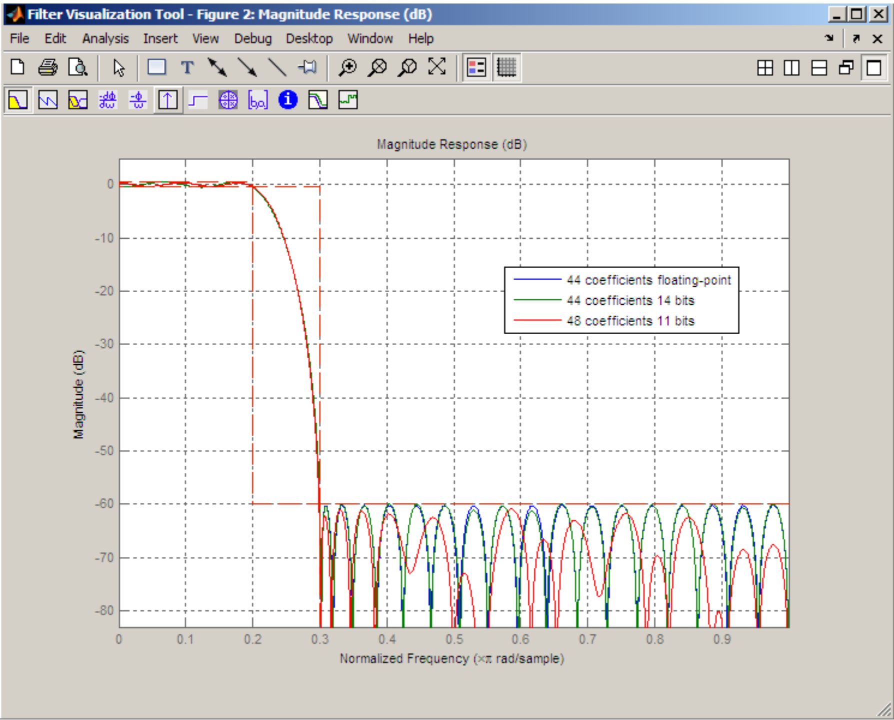
`Hq = minimizecoeffwl(Hd,...,'MatchrefFilter',RefFiltFlag)` determines whether the fixed-point filter matches the filter order and transition width of the floating-point design. Setting `'MatchRefFilter'` to `true` returns a fixed-point filter with the same order and transition width as `Hd`. The `'MatchRefFilter'` property defaults to `false` and the resulting fixed-point filter may have a different order and transition width than the floating-point design `Hd`.

You must have the Fixed-Point Designer software installed to use this function.

Examples

Minimize wordlength for lowpass FIR equiripple filter:

```
f=fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.3,1,60);
% Design filter with double-precision floating point
Hd=design(f,'equiripple');
% Find minimum wordlength fixed-point filter
% with 0.15 dB stopband tolerance
Hq=minimizecoeffwl(Hd,'MatchRefFilter',true,'Astoptol',0.15);
Hq1=minimizecoeffwl(Hd,'Astoptol',0.15);
% Hq.coeffwordlength is 14 bits.
% Hq1.coeffwordlength is 11 bits
hfvt=fvtool(Hd,Hq,Hq1,'showreference','off');
legend(hfvt,'44 coefficients floating-point',...
'44 coefficients 14 bits','48 coefficients 11 bits');
```



Tutorials

- "Fixed-Point Data Types"

See Also

`constraincoeffwl` | `measure` | `rand` | `design` | `fdesign` | `maximizestopband`

Topics

"Fixed-Point Data Types"

Introduced in R2011a

msepred

Predicted mean-squared error for adaptive filter

Compatibility

msepred will be removed in a future release. Use dsp.LMSFilter instead.

Syntax

```
[mmse,emse] = msepred(ha,x,d)
[mmse,emse,meanw,mse,tracek] = msepred(ha,x,d)
[mmse,emse,meanw,mse,tracek] = msepred(ha,x,d,m)
```

Description

[mmse,emse] = msepred(ha,x,d) predicts the steady-state values at convergence of the minimum mean-squared error (mmse) and the excess mean-squared error (emse) given the input and desired response signal sequences in x and d and the property values in the adaptfilt object ha.

[mmse,emse,meanw,mse,tracek] = msepred(ha,x,d) calculates three sequences corresponding to the analytical behavior of the LMS adaptive filter defined by ha:

- meanw — contains the sequence of coefficient vector means. The columns of matrix meanw contain predictions of the mean values of the LMS adaptive filter coefficients at each time instant. The dimensions of meanw are (size(x,1))-by-(ha.length).
- mse — contains the sequence of mean-square errors. This column vector contains predictions of the mean-square error of the LMS adaptive filter at each time instant. The length of mse is equal to size(x,1).
- tracek — contains the sequence of total coefficient error powers. This column vector contains predictions of the total coefficient error power of the LMS adaptive filter at each time instant. The length of tracek is equal to size(x,1).

[mmse,emse,meanw,mse,tracek] = msepred(ha,x,d,m) specifies an optional input argument m that is the decimation factor for computing meanw, mse, and tracek. When m > 1, msepred saves every mth predicted value of each of these sequences. When you omit the optional argument m, it defaults to one.

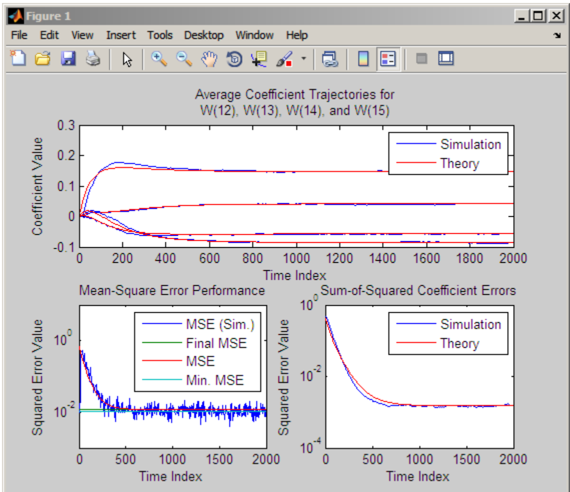
Note msepred is available for the following adaptive filters only: — adaptfilt.blms — adaptfilt.blmsfft — adaptfilt.lms — adaptfilt.nlms — adaptfilt.se Using msepred is the same for any adaptfilt object constructed by the supported filters.

Examples

Analyze and simulate a 32-coefficient adaptive filter using 25 trials of 2000 iterations each.

```
x = zeros(2000,25); d = x;     % Initialize variables
ha = fir1(31,0.5);             % FIR system to be identified
x = filter(sqrt(0.75),[1 -0.5],sign(randn(size(x))));
n = 0.1*randn(size(x));        % observation noise signal
d = filter(ha,1,x)+n;          % desired signal
l = 32;                        % Filter length
mu = 0.008;                    % LMS step size.
m  = 5;                        % Decimation factor for analysis
                               % and simulation results
ha = adaptfilt.lms(l,mu);
[mmse,emse,meanW,mse,traceK] = msepred(ha,x,d,m);
[simmse,meanWsim,Wsim,traceKsim] = msesim(ha,x,d,m);
nn = m:m:size(x,1);
subplot(2,1,1);
plot(nn,meanWsim(:,12),'b',nn,meanW(:,12),'r',nn,...
meanWsim(:,13:15),'b',nn,meanW(:,13:15),'r');
PlotTitle ={'Average Coefficient Trajectories for';...
            'W(12), W(13), W(14), and W(15)'};
title(PlotTitle);
legend('Simulation','Theory');
xlabel('Time Index'); ylabel('Coefficient Value');
subplot(2,2,3);
semilogy(nn,simmse,[0 size(x,1)],[(emse+mmse)...
(emse+mmse)],nn,mse,[0 size(x,1)],[mmse mmse]);
title('Mean-Square Error Performance');
axis([0 size(x,1) 0.001 10]);
legend('MSE (Sim.)','Final MSE','MSE','Min. MSE');
xlabel('Time Index'); ylabel('Squared Error Value');
subplot(2,2,4);
semilogy(nn,traceKsim,nn,traceK,'r');
title('Sum-of-Squared Coefficient Errors'); axis([0 size(x,1)...
0.0001 1]);
legend('Simulation','Theory');
xlabel('Time Index'); ylabel('Squared Error Value');
```

Viewing the plots in this figure you see the various error values plotted in both simulation and theory. Each subplot reveals more information about the results as the simulation converges with the theoretical performance.



See Also

`filter` | `maxstep` | `msesim`

Introduced in R2011a

msesim

Measured mean-squared error for adaptive filter

Compatibility

msesim will be removed in a future release. Use dsp.LMSFilter, dsp.BlockLMSFilter, dsp.RLSFilter, dsp.FilteredXLMSFilter, dsp.AdaptiveLatticeFilter, or dsp.LMSFilter instead.

Syntax

```
mse = msesim(ha,x,d)
[mse,meanw,w,tracek] = msesim(ha,x,d)
[mse,meanw,w,tracek] = msesim(ha,x,d,m)
```

Description

`mse = msesim(ha,x,d)` returns the sequence of mean-square errors in column vector `mse`. The vector contains estimates of the mean-square error of the adaptive filter at each time instant during adaptation. The length of `mse` is equal to `size(x,1)`. The columns of matrix `x` contain individual input signal sequences, and the columns of the matrix `d` contain corresponding desired response signal sequences.

`[mse,meanw,w,tracek] = msesim(ha,x,d)` calculates three parameters that correspond to the simulated behavior of the adaptive filter defined by `ha`:

- `meanw` — sequence of coefficient vector means. The columns of this matrix contain estimates of the mean values of the LMS adaptive filter coefficients at each time instant. The dimensions of `meanw` are (`size(x,1)`)-by-(`ha.length`).
- `w` — estimate of the final values of the adaptive filter coefficients for the algorithm corresponding to `ha`.
- `tracek` — sequence of total coefficient error powers. This column vector contains estimates of the total coefficient error power of the LMS adaptive filter at each time instant. The length of `tracek` is equal to `size(X,1)`.

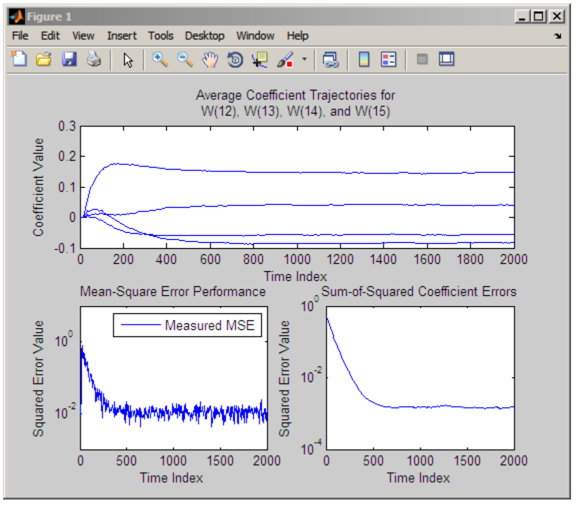
[mse,meanw,w,tracek] = msesim(ha,x,d,m) specifies an optional input argument m that is the decimation factor for computing meanw, mse, and tracek. When m > 1, msepsim saves every mth predicted value of each of these sequences. When you omit the optional argument m, it defaults to one.

Examples

Simulation of a 32-coefficient FIR filter using 25 trials, each trial having 2000 iterations of the adaptation process.

```
x = zeros(2000,25); d = x;          % Initialize variables
ha = fir1(31,0.5);                  % FIR system to be identified
x = filter(sqrt(0.75),[1 -0.5],sign(randn(size(x))));
n = 0.1*randn(size(x));             % Observation noise signal
d = filter(ha,1,x)+n;               % Desired signal
l = 32;                             % Filter length
mu = 0.008;                         % LMS Step size.
m  = 5;                             % Decimation factor for analysis
                                    % and simulation results
ha = adaptfilt.lms(l,mu);
[simmse,meanWsim,Wsim,traceKsim] = msesim(ha,x,d,m);
nn = m:m:size(x,1);
subplot(2,1,1);
plot(nn,meanWsim(:,12),'b',nn,meanWsim(:,13:15),'b');
PlotTitle ={'Average Coefficient Trajectories for';...
            'W(12), W(13), W(14), and W(15)'};
title(PlotTitle);
xlabel('Time Index'); ylabel('Coefficient Value');
subplot(2,2,3);
semilogy(nn,simmse);
title('Mean-Square Error Performance'); axis([0 size(x,1) 0.001...
10]);
legend('Measured MSE');
xlabel('Time Index'); ylabel('Squared Error Value');
subplot(2,2,4);
semilogy(nn,traceKsim);
title('Sum-of-Squared Coefficient Errors'); axis([0 size(x,1)...
0.0001 1]);
xlabel('Time Index'); ylabel('Squared Error Value');
```

Calculating the mean squared error for an adaptive filter is one measure of the performance of the adapting algorithm. In this figure, you see a variety of measures of the filter, including the error values.



See Also

filter | msepred

Introduced in R2011a

multistage

Multistage filter from specification object

Syntax

```
msFilter = design(d,'multistage','SystemObject',true)
msFilter =
design(...,'filterstructure',structure,'SystemObject',true)
msFilter = design(...,'nstages',nstages,'SystemObject',true)
msFilter = design(...,'usehalfbands',hb,'SystemObject',true)
```

Description

`msFilter = design(d,'multistage','SystemObject',true)` designs a multistage filter whose response you specified by the filter specification object `d`.

`msFilter = design(...,'filterstructure',structure,'SystemObject',true)` returns a filter with the structure specified by `structure`. Input argument `structure` is `dffir` by default and can also be one of the following options.

structure	Valid with These Responses
`firdecim`	Lowpass or Nyquist response
`firtdecim`	Lowpass or Nyquist response
`firinterp`	Lowpass or Nyquist response
`lowpass`	Default lowpass only

Multistage design applies to the default lowpass filter specification object and to decimators and interpolators that use either lowpass or Nyquist responses.

`msFilter = design(...,'nstages',nstages,'SystemObject',true)` specifies `nstages`, the number of stages to be used in the design. `nstages` must be an integer or `auto`. To allow the design algorithm to use the optimal number of stages while minimizing the cost of using the resulting filter, `nstages` is `auto` by default. When you specify an integer for `nstages`, the design algorithm minimizes the cost for the number of stages you specify.

`msFilter = design(...,'`**`usehalfbands`**`',hb,'SystemObject',true)` uses halfband filters when you set `hb` to `true`. The default value for `hb` is `false`.

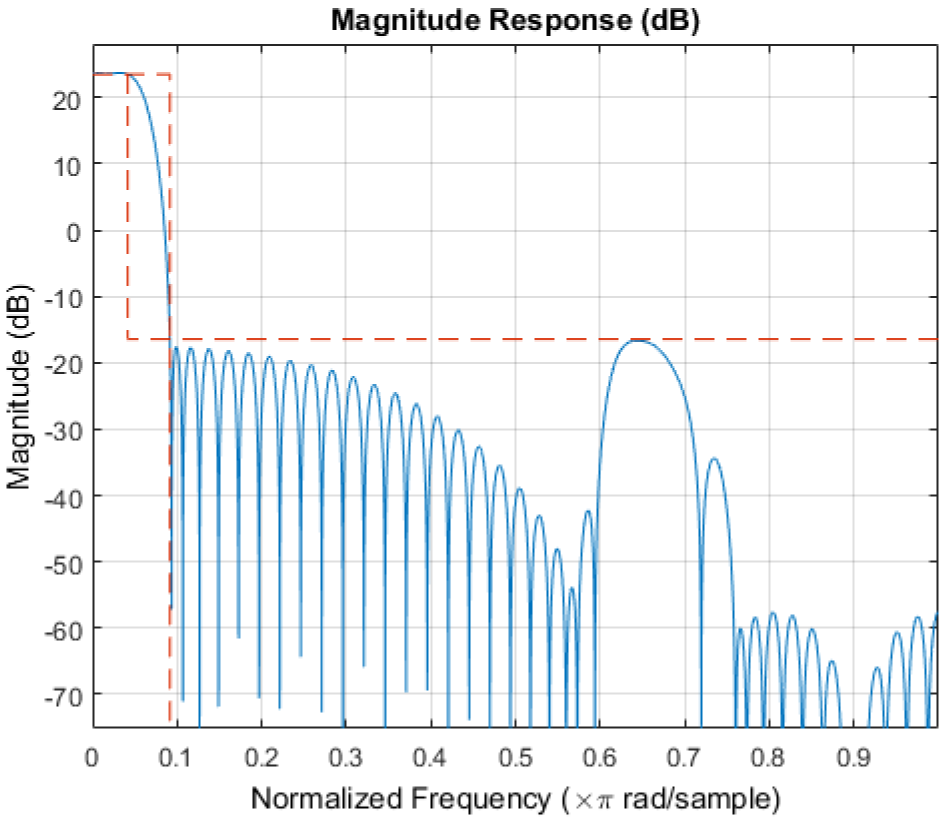
Note: To see a list of the design methods available for your filter, use `designmethods(hd)`.

Examples

Design a Multistage Interpolator

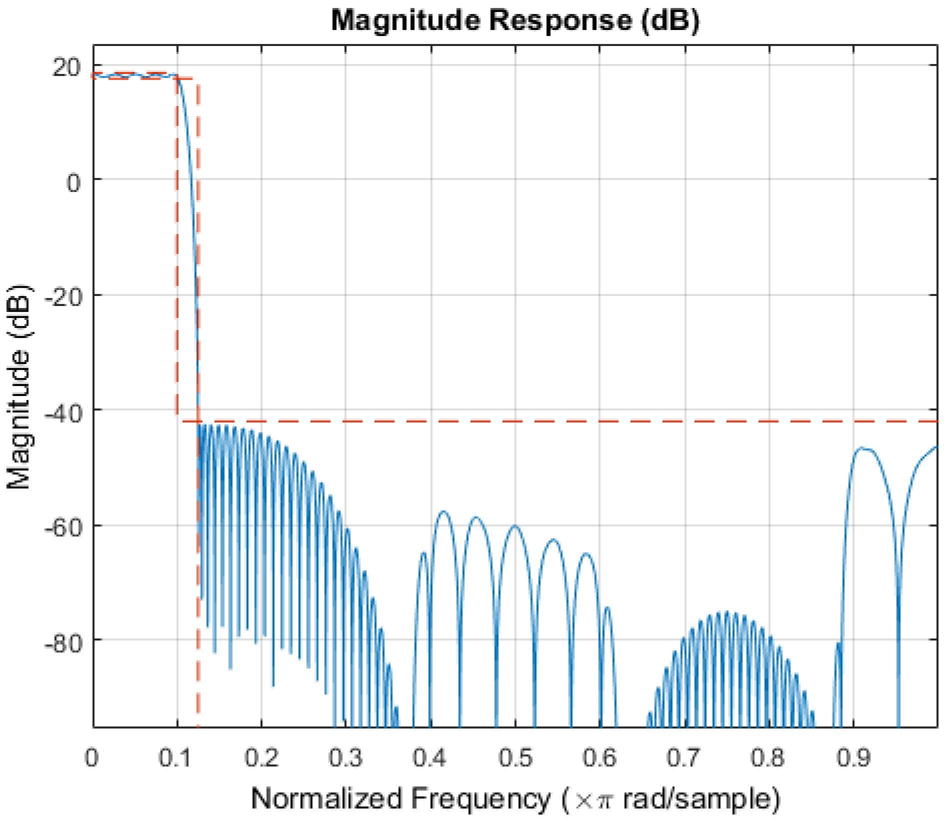
This example designs a minimum-order, multistage Nyquist interpolator.

```
l = 15;   % Interpolation factor. Also the Nyquist band.
tw = 0.05; % Normalized transition width
ast = 40;   % Minimum stopband attenuation in dB
d = fdesign.interpolator(l,'nyquist',l,'tw,ast',tw,ast);
msMinInterp = design(d,'multistage','SystemObject',true);
fvtool(msMinInterp);
```



Design a multistage lowpass interpolator with an interpolation factor of 8.

```
m = 8;
d = fdesign.interpolator(m,'lowpass');
% Use halfband filters if possible.
msInterp = design(d,'multistage','Usehalfbands',true,'SystemObject',true);
fvtool(msInterp);
```



See Also

`design` | `designopts`

Introduced in R2011a

noisepsd

Power spectral density of filter output due to roundoff noise

Syntax

```
hpsd = noisepsd(sysobj,L)
hpsd = noisepsd(sysobj,L,param1,value1,param2,value2,...)
hpsd = noisepsd(sysobj,L,opts)
noisepsd(sysobj,...)
```

Description

`hpsd = noisepsd(sysobj,L)` computes the power spectral density (PSD) at the output of filter System object, `sysobj`, occurring because of roundoff noise. This noise is produced by quantization errors within the filter. `L` is the number of trials used to compute the average. The PSD is computed from the average over the `L` trials. The more trials you specify, the better the estimate, but at the expense of longer computation time. When you do not explicitly specify `L`, the default is 10 trials.

`hpsd` is a `psd` data object. To extract the PSD vector (the data from the PSD) from `hpsd`, enter

```
get(hpsd,'data')
```

at the prompt. Plot the PSD data with `plot(hpsd)`. The average power of the output noise (the integral of the PSD) can be computed with `avgpower`, a method of `dspdata` objects:

```
avgpwr = avgpower(hpsd).
```

`hpsd = noisepsd(sysobj,L,param1,value1,param2,value2,...)` where `sysobj` is a filter System object, specifies optional parameters via propertyname/propertyvalue pairs. Valid `psd` object property values are:

Property Name	Default Value	Description and Valid Entries
`Nfft`	512	Specify the number of FFT points to use to calculate the PSD.

Property Name	Default Value	Description and Valid Entries
NormalizedFrequency	true	Determine whether to use normalized frequency. Enter a logical value of true or false. Because this property is a logical value, do not enclose the single quotation marks.
Fs	normalized	Specify the sampling frequency to use when you set NormalizedFrequency to false. Use any integer value greater than 1. Enter the value in Hz.
SpectrumType	onesided	Specify how noisepsd should generate the PSD. Options are onesided or twosided. If you choose a two-sided computation, you can also choose CenterDC = true. Otherwise, CenterDC must be false. • onesided converts the type to a spectrum that is calculated over half the Nyquist interval. All properties affected by the new frequency range are adjusted automatically. • twosided converts the type to a spectrum that is calculated over the whole Nyquist interval. All properties affected by the new frequency range are adjusted automatically.
CenterDC	false	Shift the zero-frequency component to the center of a two-sided spectrum. • When you set SpectrumType to onesided, it is changed to twosided and the data is converted to a two-sided spectrum. • Setting CenterDC to false shifts the data and the frequency values in the object so that DC is in the left edge of the spectrum. This operation does not affect the SpectrumType property setting.

Property Name	Default Value	Description and Valid Entries
`Arithmetic`	`ARITH`	Analyze the filter System object, based on the arithmetic specified in the `ARITH` input. `ARITH` can be set to `double`, `single`, or `fixed`. The analysis tool assumes a double-precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state.

Note If the spectrum data you specify is calculated over half the Nyquist interval and you do not specify a corresponding frequency vector, the default frequency vector assumes that the number of points in the whole FFT was even. Also, the plot option to convert to a whole or two-sided spectrum assumes the original whole FFT length is even.

`noisepsd` requires knowledge of the input data type. Analysis cannot be performed if the input data type is not available. If you do not specify the `Arithmetic` parameter, i.e., use the syntax `[h,w] = noisepsd(sysobj)` , then the following rules apply to this method:

- The System object state is `Unlocked` — `noisepsd` performs double-precision analysis.
- The System object state is `Locked` — `noisepsd` performs analysis based on the locked input data type.

If you do specify the `Arithmetic` parameter, i.e., use the syntax `[h,w] = noisepsd(sysobj,'Arithmetic', ARITH)`, review the following rules for this method. Which rule applies depends on the value you set for the `Arithmetic` parameter.

Value	System Object State	Rule
`ARITH = ’double’`	`Unlocked`	`noisepsd` performs double-precision analysis.
	`Locked`	`noisepsd` performs double-precision analysis.
`ARITH = ‘single’`	`Unlocked`	`noisepsd` performs single-precision analysis.
	`Locked`	`noisepsd` performs single-precision analysis.

Value	System Object State	Rule
ARITH = 'fixed'	Unlocked	noisepsd produces an error because the fixed-point input data type is unknown.
	Locked	When the input data type is double or single, then noisepsd produces an error because since the fixed-point input data type is unknown.
		When the input data is of fixed-point type, noisepsd performs analysis based on the locked input data type.

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpoleFilter
dsp.AllpassFilter
dsp.BiquadFilter
dsp.CoupledAllpassFilter
dsp.FIRFilter
dsp.HighpassFilter
dsp.IIRFilter
dsp.LowpassFilter
dsp.NotchPeakFilter
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

`hpsd = noisepsd(sysobj,L,opts)` uses an options object, `opts`, to specify the optional input arguments. This specification is not made using property-value pairs in the command. Use `opts = noisepsdopts(sysobj)` to create the object. `opts` then has the `noisepsd` settings from `sysobj`. After creating `opts`, you change the property values before calling `noisepsd`:

```
set(opts,'fs',48e3); % Set Fs to 48 kHz.
```

`noisepsd(sysobj,...)` with no output argument launches fvtool.

Examples

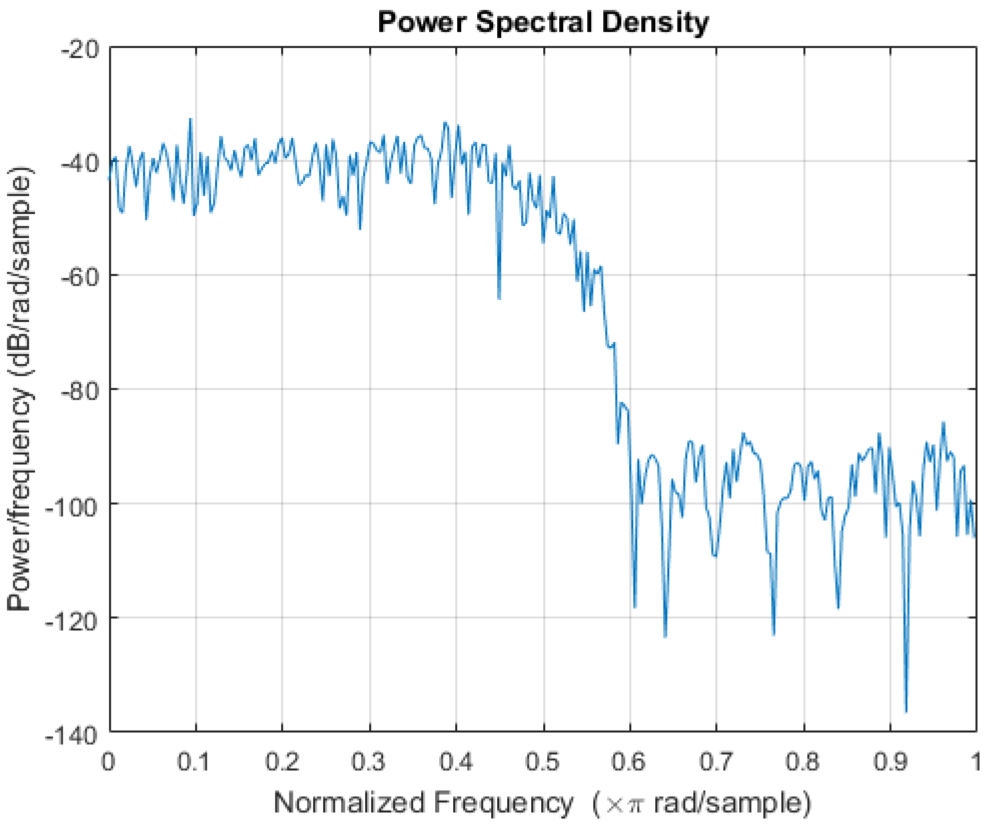
Compute the PSD of Output Noise

Compute the PSD of the output noise caused by the quantization processes in a fixed-point, direct form FIR filter. The input is of fixed-point type. `noisepsd` performs analysis based on the locked input data type.

```
b = firgr(27,[0 .4 .6 1],[1 1 0 0]);
firfilt = dsp.FIRFilter('Numerator',b); % Create the filter object.
data = fi(randn(15,16),1,16,3);
output = firfilt(data);
```

Quantize the filter to fixed-point.

```
hpsd = noisepsd(firfilt,'Arithmetic','fixed');
plot(hpsd)
```



`hpsd` looks similar to the following figure - the data resulting from the noise PSD calculation. You can review the data in hpsd.data.

References

[1] McClellan, et al., *Computer-Based Exercises for Signal Processing Using MATLAB 5*. Upper Saddle River, N.J.: Prentice-Hall, 1998.

See Also

See Also

Functions
`filter` | `noisepsdopts` | `norm` | `reorder` | `scale`

Introduced in R2011a

noisepsdopts

Options for running filter output noise PSD

Syntax

```
opts = noisepsdopts(sysobj)
```

Description

`opts = noisepsdopts(sysobj)` uses the current settings in the filter System object, `sysobj`, to create an options object, `opts`, that contains specified options for computing the output noise PSD. You can pass `opts` to the `noisepsd` method as an input argument.

Using `opts`, you can set the following properties for `noisepsd`:

Property Name	Default Value	Description and Valid Entries
`Nfft`	`512`	Specify the number of FFT points to use to calculate the PSD.
`NormalizedFrequency`	`true`	Determine whether to use normalized frequency. Enter a logical value of the logical `true` or `false`. Because this property is a logical value, do not enclose with single quotation marks.
`Fs`	`normalized`	Specify the sampling frequency to use when you set `NormalizedFrequency` to `false`. Use any integer value greater than 1. Enter the value in Hz.
`SpectrumType`	`onesided`	Specify how `noisepsd` should generate the PSD. Options are onesided or twosided. If you choose a two-sided computation, you can also choose `centerdc` = `true`. Otherwise, `centerdc` must be `false`.

Property Name	Default Value	Description and Valid Entries
		• `onesided` converts the type to a spectrum that is calculated over half the Nyquist interval. All properties affected by the new frequency range are adjusted automatically. • `twosided` converts the type to a spectrum that is calculated over the whole Nyquist interval. All properties affected by the new frequency range are adjusted automatically.
`CenterDC`	`false`	Shift the zero-frequency component to the center of a two-sided spectrum. • When you set `SpectrumType` to `onesided`, it is changed to `twosided` and the data is converted to a two-sided spectrum. • Setting `CenterDC` to `false` shifts the data and the frequency values in the object so that DC is in the left edge of the spectrum. This operation does not effect the `SpectrumType` property setting.
`Arithmetic`	`ARITH`	Analyze the filter System object, based on the arithmetic specified in the `ARITH` input. `ARITH` can be set to `double`, `single`, or `fixed`. The analysis tool assumes a double-precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state.

See Also

See Also

Functions
`noisepsd`

Introduced in R2011a

norm

P-norm of filter

Syntax

```
l = norm(hd)
l = norm(hd,pnorm)
```

Description

All of the variants of `norm` return the filter p-norm for the object in the syntax, a digital filter. When you omit the `pnorm` argument, `norm` returns the L2-norm for the object.

Note that by Parseval's theorem, the L2-norm of a filter is equal to the l2 norm. This equality is not true for the other norm variants.

For dfilt Objects

`l = norm(hd)` returns the L2-norm of a discrete-time filter.

`l = norm(hd,pnorm)` includes input argument `pnorm` that lets you specify the norm returned. `pnorm` can be either

- Frequency-domain norms specified by one of `L1`, `L2`, or `Linf`
- Discrete-time domain norms specified by one of `l1`, `l2`, or `linf`

By Parseval's theorem, the L2-norm of a filter is equal to the l2 norm. This equality is not true for the other norm variants.

IIR filters respond slightly differently to `norm`. When you compute the l2, linf, L1, and L2 norms for an IIR filter, `norm(...,L2,tol)` lets you specify the tolerance for the accuracy in the computation. For l1, l2, L2, and linf, `norm` uses the tolerance to truncate the infinite impulse response that it uses to calculate the norm. For L1, `norm` passes the tolerance to the numerical integration algorithm. Refer to Examples to see this in use. You cannot specify `Linf` for the norm and include the `tol` option.

Examples

Norm of an IIR Filter

This example shows how to compute the L2 norm of an IIR filter. A tolerance of 1e-10 is used.

```
spec = fdesign.lowpass('n,fc',5,0.4);
filter = butter(spec);
filternorm = norm(filter,'l2',1e-10)
```

```
filternorm =

    0.6336
```

See Also

reorder | scale | scalecheck

Introduced in R2011a

normalize

Normalize filter numerator or feed-forward coefficients

Syntax

```
normalize(hq)
g = normalize(hd)
```

Description

`normalize(hq)` normalizes the filter numerator coefficients for a quantized filter to have values between -1 and 1. The coefficients of `hq` change — `normalize` does not copy `hq` and return the copy. To restore the coefficients of `hq` to the original values, use `denormalize`.

Note that for lattice filters, the feed-forward coefficients stored in the property `lattice` are normalized.

`g = normalize(hd)` normalizes the numerator coefficients for the filter `hq` to between -1 and 1 and returns the gain `g` due to the normalization operation. Calling `normalize` again does not change the coefficients. `g` always returns the gain returned by the first call to normalize the filter.

Examples

Normalize the Coefficients of a Direct Form II Filter

Create a direct form II quantized filter that uses second-order sections. Then use normalize to maximize the use of the range of representable coefficients.

```
d = fdesign.lowpass('n,fp,ap,ast',8,.5,2,40);
hd = design(d,'ellip');
hd.arithmetic ='fixed';
```

Check the filter coefficients. Note that InitialSOSMatrix(3,2)>1

```
InitialSOSMatrix = hd.sosMatrix;
```

Use normalize to modify the coefficients into the range between -1 and 1. The output g contains the gains applied to each section of the SOS filter.

```
g = normalize(hd);
```

None of the numerator coefficients exceed -1 or 1.

See Also

`denormalize`

Introduced in R2011a

normalizefreq

Switch filter specification between normalized frequency and absolute frequency

Syntax

```
normalizefreq(d)
normalizefreq(d,flag)
normalizefreq(d,false,fs)
```

Description

`normalizefreq(d)` normalizes the frequency specifications in filter specifications object `d`. By default, the `NormalizedFrequency` property is set to `true` when you create a design object. You provide the design specifications in normalized frequency units. `normalizefreq` does not affect filters that already use normalized frequency.

If you use this syntax when `d` does not use normalized frequency specifications, all of the frequency specifications are normalized by `fs`/2 so they lie between 0 and 1, where `fs` is specified in the object. Included in the normalization are the filter properties that define the filter pass and stopband edge locations by frequency:

- `F3 dB` — Used by IIR filter specifications objects to describe the passband cutoff frequency
- `Fcutoff` — Used by FIR filter specifications objects to describe the passband cutoff frequency
- `Fpass` — Describes the passband edges
- `Fstop` — Describes the stopband edges

In this syntax, `normalizefreq(d)` assumes you specified `fs` when you created `d` or changed `d` to use absolute frequency specifications.

`normalizefreq(d,flag)` where `flag` is either **`true`** or **`false`**, specifies whether the `NormalizedFrequency` property value is `true` or `false` and therefore whether the filter normalizes the sampling frequency `fs` and other related frequency specifications. `fs` defaults to 1 for this syntax.

When you do not provide the input argument `flag`, it defaults to `true`. If you set `flag` to `false`, affected frequency specifications are multiplied by `fs`/2 to remove the normalization. Use this syntax to switch your filter between using normalized frequency specifications and not using normalized frequency specifications.

`normalizefreq(d,`**`false`**`,fs)` lets you specify a new sampling frequency `fs` when you set the `NormalizedFrequency` property to `false`.

Examples

Normalize the Frequency Specifications of a Filter

These examples demonstrate using `normalizefreq` in both of the major syntax applications-setting the design object frequency specifications to use absolute frequency (normalizefreq(hd,false,fs)) and resetting a design object to using normalized frequencies (normalizefreq(d)).

Construct a highpass filter specifications object by specifying the passband and stopband edges and the desired attenuations in the bands. By default, provide the frequency specifications in normalized values between 0 and 1.

```
d = fdesign.highpass(0.35, 0.45, 60, 40);
```

`Fstop` and `Fpass` are in normalized form, and the property `NormalizedFrequency` is true.

Now use normalizedfreq to convert to absolute frequency specifications, with a sampling frequency of 1000 Hz.

```
normalizefreq(d,false,1e3);
```

Both of the attenuation specifications remain the same. The passband and stopband edge definitions now appear in Hz, where the new value represents the normalized values multiplied by Fs/2, or 500 Hz.

Converting to using normalized frequencies consists of using `normalizefreq` with the design object d.

```
normalizefreq(d)
```

For bandstop, bandpass, and multiple band filter specifications objects, normalizefreq works the same way for all band edge definitions. When you do not provide the sampling frequency Fs as an input argument and you are converting to absolute frequency specifications, normalizefreq sets Fs to 1, as shown in this example.

```
d=fdesign.bandstop(0.25,0.35,0.55,0.65,50,60);
normalizefreq(d,false)
```

See Also

`fdesign.lowpass` | `fdesign.halfband` | `fdesign.highpass` | `fdesign.interpolator`

Introduced in R2011a

nstates

Number of filter states

Syntax

```
n = nstates(hd)
```

Description

Discrete-Time Filters

`n = nstates(hd)` returns the number of states `n` in the discrete-time filter `hd`. The number of states depends on the filter structure and the coefficients.

Examples

Number of States of a Filter

Determine the number of states of a direct form FIR filter.

```
FIRFilter = firls(30,[0 .1 .2 .5]*2,[1 1 0 0]);
DiscFilter = dfilt.dffir(FIRFilter);
NstateDF = nstates(DiscFilter)
```

```
NstateDF =

    30
```

Introduced in R2011a

order

Order of discrete-time filter System object

Syntax

```
n = order(sysobj)
```

Description

`n = order(sysobj)` returns the order `n` of the filter System object `sysobj`. The order depends on the filter structure and the reference double-precision floating-point coefficients.

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpassFilter
dsp.AllpoleFilter
dsp.BiquadFilter
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.CICDecimator
dsp.CICInterpolator
dsp.CoupledAllpassFilter
dsp.FarrowRateConverter
dsp.FilterCascade
dsp.FIRDecimator
dsp.FIRFilter
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator

Filter System objects
dsp.FIRInterpolator
dsp.FIRRateConverter
dsp.HighpassFilter
dsp.IIRFilter
dsp.IIRHalfbandDecimator
dsp.IIRHalfbandInterpolator
dsp.LowpassFilter
dsp.NotchPeakFilter
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

Examples

Determine the Order of a Filter

Design a compensation decimator for a CIC decimator using the `dsp.FilterCascade` object. Determine the order of the overall filter.

```
cicdecim = dsp.CICDecimator('DecimationFactor', 6, ...
    'NumSections', 6);

fs = 16e3;     % Sampling frequency of input of compensation decimator
fPass = 4e3;   % Passband frequency
fStop = 4.5e3; % Stopband frequency
ciccomp = dsp.CICCompensationDecimator(cicdecim, ...
    'DecimationFactor', 2, ...
    'PassbandFrequency', fPass, ...
    'StopbandFrequency', fStop, ...
    'SampleRate', fs);

filtchain = dsp.FilterCascade(cicdecim, ciccomp);

order(filtchain)
```

```
ans =
```

```
    648
```

Introduced in R2011a

phasedelay

Phase delay response of discrete-time filter System object

Syntax

```
[phi,w] = phasedelay(sysobj)
[phi,w] = phasedelay(sysobj,n)
[phi,w] = phasedelay(sysobj,Name,Value)
phasedelay(sysobj)
```

Description

`[phi,w] = phasedelay(sysobj)` returns the phase delay `phi` of the filter System object, `sysobj`, based on the current filter coefficients. The vector `w` contains the frequencies (in radians) at which the phase delay is evaluated.

The phase delay is evaluated at 8192 points equally spaced around the upper half of the unit circle.

`[phi,w] = phasedelay(sysobj,n)` returns the phase delay of the filter System object at `n` points equally spaced around the upper half of the unit circle.

`[phi,w] = phasedelay(sysobj,Name,Value)` returns the phase delay with additional options specified by one or more `Name,Value` pair arguments.

`phasedelay(sysobj)` plots the phase delay of the filter System object `sysobj` in the `fvtool`.

For information on more input options, refer to `phasedelay` in Signal Processing Toolbox documentation.

Examples

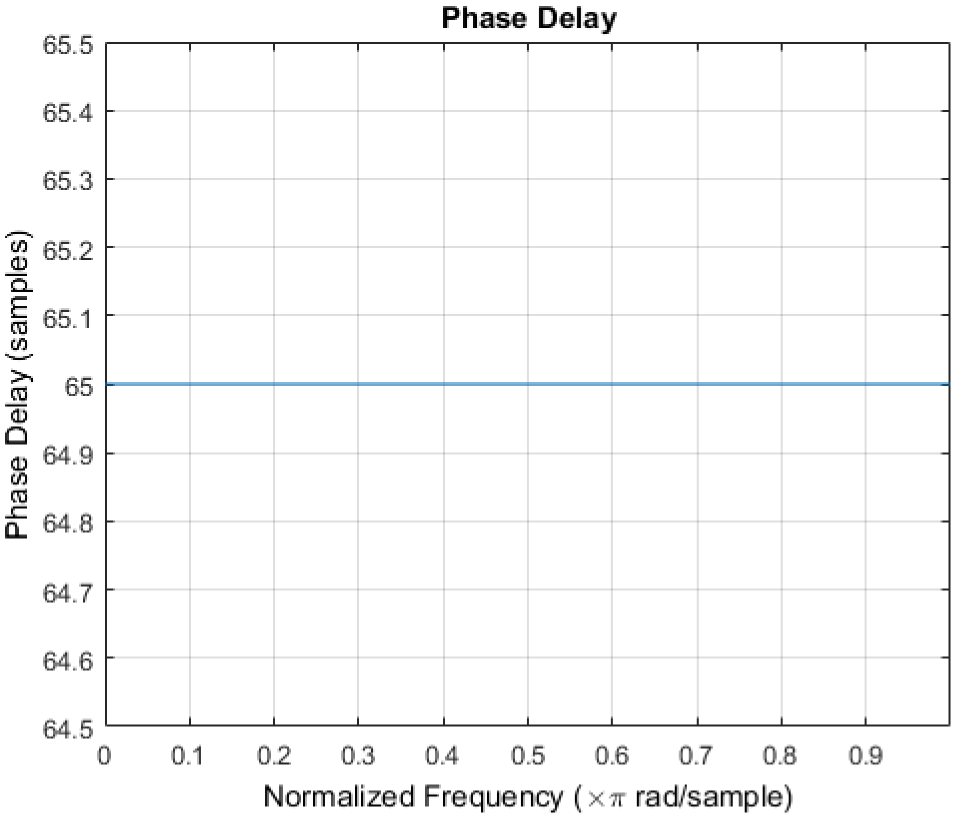
Phase Delay of a Discrete-Time Filter

```
Fs = 8000; Fcutoff = 2000;
```

```
FIRFilt = dsp.FIRFilter('Numerator', fir1(130,Fcutoff/(Fs/2)));
```

`phasedelay` computes the phase delay of the filter and displays it using fvtool

```
phasedelay(FIRFilt);
```



Input Arguments

`sysobj`

Filter System object.

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpassFilter
dsp.AllpoleFilter
dsp.BiquadFilter
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.CICDecimator
dsp.CICInterpolator
dsp.CoupledAllpassFilter
dsp.FarrowRateConverter
dsp.FilterCascade
dsp.FIRDecimator
dsp.FIRFilter
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.FIRInterpolator
dsp.FIRRateConverter
dsp.HighpassFilter
dsp.IIRFilter
dsp.IIRHalfbandDecimator
dsp.IIRHalfbandInterpolator
dsp.LowpassFilter
dsp.NotchPeakFilter
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

n

Number of samples. For an FIR filter where n is a power of two, the computation is done faster using FFTs.

Default: 8192

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Arithmetic'` — Value types:
`‘double’ | 'single' | 'fixed'`

Specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.

System Object State	Coefficient Data Type	Rule
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

phi

Phase delay vector of length n. If n is not specified, the function uses a default value of 8192.

w

Frequency vector of length n, in radians/sample. w consists of n points equally spaced around the upper half of the unit circle (from 0 to π radians/sample). If n is not specified, the function uses a default value of 8192.

Algorithms

You can provide fs, the sampling frequency, as an input as well. phasedelay uses fs to calculate the delay response and plots the response to fs/2.

See Also

See Also

Functions
freqz | freqz | fvtool | grpdelay | phasez | phasez | zerophase | zerophase | zplane

Introduced in R2011a

phasez

Unwrapped phase response for filter

Syntax

```
[phi,w] = phasez(sysobj)
[phi,w] = phasez(sysobj,n)
[phi,w] = phasez(sysobj,Name,Value)
phasez(sysobj)
```

Description

`[phi,w] = phasez(sysobj)` returns the unwrapped phase response `phi` of the filter System object, `sysobj`, based on the current filter coefficients. The vector `w` contains the frequencies (in radians) at which the function evaluates the phase response. The phase response is evaluated at 8192 points equally spaced around the upper half of the unit circle.

`[phi,w] = phasez(sysobj,n)` returns the phase response of the filter System object and the corresponding frequencies at `n` points equally spaced around the upper half of the unit circle.

`[phi,w] = phasez(sysobj,Name,Value)` returns the phase response with additional options specified by one or more `Name,Value` pair arguments.

`phasez(sysobj)` displays the phase response of the filter System object `sysobj` in the `fvtool`.

For more information about optional input arguments for `phasez`, refer to `phasez` in Signal Processing Toolbox documentation.

Examples

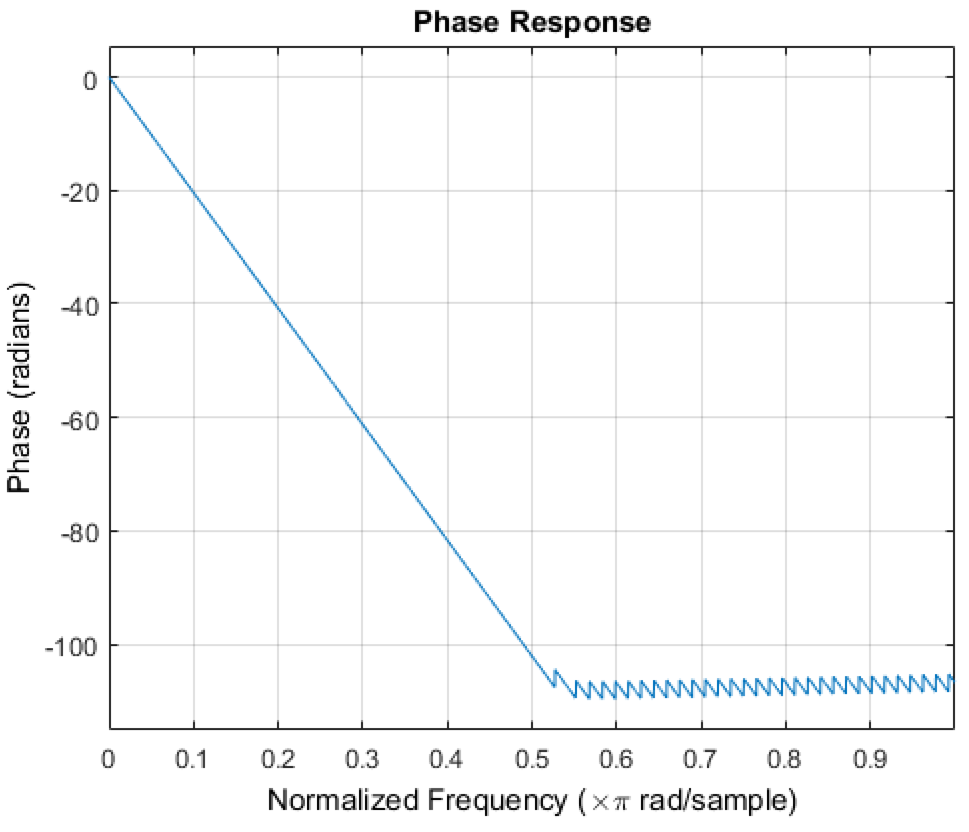
Unwrapped Phase Response of a Discrete-Time Filter

```
Fs = 8000; Fcutoff = 2000;
```

```
FIRFilt = dsp.FIRFilter('Numerator', fir1(130,Fcutoff/(Fs/2)));
```

`phasez` computes the unwrapped phase response of the filter and displays it using fvtool

```
phasez(FIRFilt);
```



Input Arguments

`sysobj`

Filter System object.

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpassFilter
dsp.AllpoleFilter
dsp.BiquadFilter
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.CICDecimator
dsp.CICInterpolator
dsp.CoupledAllpassFilter
dsp.FarrowRateConverter
dsp.FilterCascade
dsp.FIRDecimator
dsp.FIRFilter
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.FIRInterpolator
dsp.FIRRateConverter
dsp.HighpassFilter
dsp.IIRFilter
dsp.IIRHalfbandDecimator
dsp.IIRHalfbandInterpolator
dsp.LowpassFilter
dsp.NotchPeakFilter
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

n

Number of samples. For an FIR filter where `n` is a power of two, the computation is done faster using FFTs.

Default: 8192

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Arithmetic'` — Value types:
`‘double’ | 'single' | 'fixed'`

Specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.

System Object State	Coefficient Data Type	Rule
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

phi

Phase response vector of length n. If n is not specified, the function uses a default value of 8192.

w

Frequency vector of length n, in radians/sample. w consists of n points equally spaced around the upper half of the unit circle (from 0 to π radians/sample). If n is not specified, the function uses a default value of 8192.

See Also

See Also

Functions

freqz | freqz | fvtool | grpdelay | phasedelay | phasez | zerophase | zplane

Introduced in R2011a

polyphase

Polyphase decomposition of multirate filter

Syntax

```
p = polyphase(hs)
p = polyphase(hs,Name,Value)
polyphase(hs)
```

Description

`p = polyphase(hs)` returns the polyphase matrix `p` of the multirate filter System object `hs`.

`p = polyphase(hs,Name,Value)` returns the polyphase matrix `p` of the multirate filter System object `hs`.

`polyphase(hs)` launches the Filter Visualization Tool (FVTool) with all the polyphase subfilters to allow you to analyze each component subfilter individually.

Input Arguments

hs

Filter System object.

dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.FIRRateConverter
dsp.FIRDecimator
dsp.FIRInterpolator

dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.IIRHalfbandDecimator
dsp.IIRHalfbandInterpolator

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Arithmetic'` — Value types:
`'double'` | `'single'` | `'fixed'`

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`,

System Object State	Coefficient Data Type	Rule
		the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

p

Polyphase matrix `p` of the multirate filter. Each row in the matrix represents one subfilter of the multirate filter. The first row of matrix `p` represents the first subfilter, the second row the second subfilter, and so on to the last subfilter.

Examples

Polyphase Matrix of an FIR Interpolator

When you create a multirate filter that uses polyphase decomposition, polyphase lets you analyze the component filters individually by returning the components as rows in a matrix. First, create an interpolate-by-three filter.

```
hs = dsp.FIRInterpolator
```

```
hs =

  dsp.FIRInterpolator with properties:

        NumeratorSource: 'Property'
              Numerator: [1×16 double]
    InterpolationFactor: 3

  Use get to show all properties
```

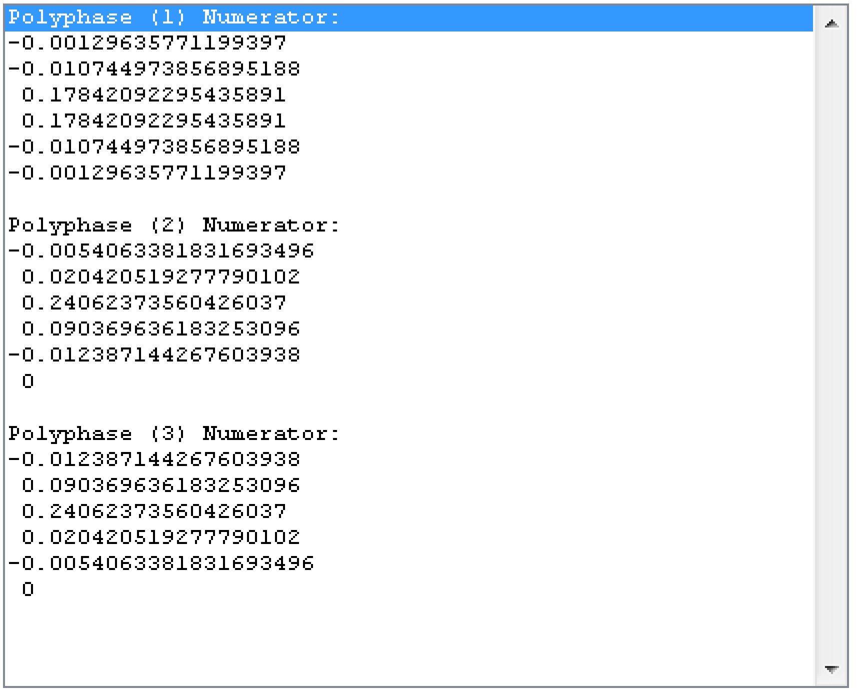
In this syntax, the matrix `p` contains all of the subfilters for `hm`, one filter per matrix row.

```
p = polyphase(hs)
```

```
p =

   -0.0013   -0.0107    0.1784    0.1784   -0.0107   -0.0013
   -0.0054    0.0204    0.2406    0.0904   -0.0124         0
   -0.0124    0.0904    0.2406    0.0204   -0.0054         0
```

Finally, using `polyphase` without an output argument opens the Filter Visualization Tool, ready for you to use the analysis capabilities of the tool to investigate the interpolator `hm`.

```
polyphase(hs)
```



The fvtool shows the coefficients of the subfilters. To see the magnitude response of the subfilters, click on the `Magnitude Response` button on the fvtool toolstrip.

See Also

dsp.FIRInterpolator

Introduced in R2011a

qreport

Most recent fixed-point filtering operation report

Syntax

```
rlog = qreport(h)
```

Description

`rlog = qreport(h)` returns the logging report stored in the filter object `h` in the object `rlog`. The ability to log features of the filtering operation is integrated in the fixed-point filter object and the `filter` method.

Each time you filter a signal with `h`, new log data overwrites the results in the filter from the previous filtering operation. To save the log from a filtering simulation, change the name of the output argument for the operation before subsequent filtering runs.

Note `qreport` requires Fixed-Point Designer software and that filter `h` is a fixed-point filter. Data logging for `fi` operations is a preference you set for each MATLAB session. To learn more about logging, `LoggingMode`, and `fi` object preferences, refer to `fipref` in the Fixed-Point Designer documentation.

Also, you cannot use `qreport` to log the filtering operations from a fixed-point Farrow filter.

Enable logging during filtering by setting `LoggingMode` to `on` for `fi` objects for your MATLAB session. Trigger logging by setting the `Arithmetic` property for `h` to `fixed`, making `h` a fixed-point filter and filtering an input signal.

Using Fixed-Point Filtering Logging

Filter operation logging with `qreport` requires some preparation in MATLAB. Complete these steps before you use `qreport`.

1. Set the fixed-point object preference for `LoggingMode` to `on` for your MATLAB session. This setting enables data logging.

   ```
   fipref('LoggingMode','on')
   ```
2. Create your fixed-point filter.
3. Filter a signal with the filter.
4. Use `qreport` to return the filtering information stored in the filter object.

`qreport` provides a way to instrument your fixed-point filters and the resulting data log offers insight into how the filter responds to a particular input data signal.

Report object `rlog` contains a filter-structure-specific list of internal signals for the filter. Each signal contains

- Minimum and maximum values that were recorded during the last simulation. Minimum and maximum values correspond to values before quantization.
- Representable numerical range of the word length and fraction length format
- Number of overflows during filtering for that signal.

Examples

View the Logging Report

This example shows how to use `qreport` to log the results of filtering a sinusoidal signal with a fixed-point direct-form FIR filter, firFilt. To use the `qreport`, set the `LoggingMode` of fixed-point objects to 'on'.

```
fipref('loggingmode','on');
hd = design(fdesign.lowpass,'equiripple');
hd.arithmetic = 'fixed';
hd.InputWordLength = 32;
fs = 1000;          % Input sampling frequency.
t = 0:1/fs:1.5;     % Signal length = 1501 samples.
x = sin(2*pi*10*t); % Amplitude = 1 sinusoid.
y = filter(hd,x);
rlog = qreport(hd)
```

```
rlog =
```

```
                              Fixed-Point Report
            ---------------------------------------------------------------------
                 Min               Max        |          Range             |
            ---------------------------------------------------------------------
       Input:             -1                 1 |        -65536        65536 |
      Output:       -1.02325           1.02325 |       -131072       131072 |
     Product:    -0.48538208        0.48538208 |        -32768        32768 |
 Accumulator:     -1.0852075         1.0852075 |       -131072       131072 |
```

See Also

See Also

Functions
`dfilt`

Introduced in R2011a

realizemdl

Simulink subsystem block for filter

Syntax

```
realizemdl(hq)
realizemdl(hq,Name,Value)
```

Description

`realizemdl(hq)` generates a model of filter `hq` in a Simulink subsystem block using sum, gain, and delay blocks from Simulink. The properties and values of `hq` define the resulting subsystem block parameters.

`realizemdl` requires Simulink. To accurately realize models of quantized filters, use Fixed-Point Designer.

`realizemdl(hq,Name,Value)` generates the model for `hq` with additional options specified by one or more `Name,Value` pair arguments. Using name-value pair arguments lets you control more fully the way the block subsystem model gets built. You can specify such details as where the block goes, what the name is, or how to optimize the block structure.

Input Arguments

hq — Filter System object
filter System object

List of filter system objects that the function supports:

Filter System objects
dsp.AllpassFilter
dsp.AllpoleFilter

Filter System objects
dsp.BiquadFilter
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.CoupledAllpassFilter
dsp.FarrowRateConverter
dsp.FilterCascade
dsp.FIRFilter
dsp.FIRInterpolator
dsp.FIRDecimator
dsp.FIRRateConverter
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.HighpassFilter
dsp.IIRFilter
dsp.IIRHalfbandDecimator
dsp.IIRHalfbandInterpolator
dsp.LowpassFilter
dsp.NotchPeakFilter

Name-Value Pair Arguments

`'Destination'` — Destination choices:
`'current'` (default) | `'new'` | `subsystemname`

Specify whether to add the block to your current Simulink model or create a new model to contain the block. If you provide the name of a current subsystem in *`subsystemname`*, `realizemdl` adds the new block to the specified subsystem.

`'Blockname'` — The name of the block (not the default) :
`'filter'` (default)

Provide the name for the new subsystem block. By default the block is named Filter.

'MapCoeffstoPorts' — Value options:
'off' (default) | 'on'

Specify whether to map the coefficients of the filter to the ports of the block.

'MapStates' — Value options:
'off' (default) | 'on'

Specify whether to apply the current filter states to the realized model. Such specification allows you to save states from a filter object you may have used or configured in a specific way. The default setting of 'off' means the states are not transferred to the model. Setting the property to 'on' preserves the current filter states in the realized model.

'OverwriteBlock' — Value options:
'off' (default) | 'on'

Specify whether to overwrite an existing block with the same name or create a new block.

'OptimizeZeros' — Value options:
'off' (default) | 'on'

Specify whether to remove zero-gain blocks.

'OptimizeOnes' — Value options:
'off' (default) | 'on'

Specify whether to replace unity-gain blocks with direct connections.

'OptimizeNegOnes' — Value options:
'off' (default) | 'on'

Specify whether to replace negative unity-gain blocks with a sign change at the nearest sum block.

'OptimizeDelayChains' — Value options:
'off' (default) | 'on'

Specify whether to replace delay chains made up of n unit delays with a single delay by n.

'CoeffNames' — Names of coefficients:
{'Num'} (default FIR) | {'Num','Den'} (default direct form IIR) | {'Num','Den','g'} (default IIR SOS) | {'Num_1,'Num_2,'Num_3'...} (default multistage) | {'K'} (default form lattice)

Specify the coefficient variable name as a cell array of character vectors. `MapCoeffsToPorts` must be set to `'on'` for this property to apply.

`'InputProcessing'` — Possible input processing options:
`'columns as channels'` (default) | `'elements as channels'`

Specify sample-based (`'elements as channels'`) or frame-based (`'columns as channels'`) processing.

`'RateOption'` — Rate options:
`'enforce single rate'` (default) | `'allow multirate'`

Specify how the block adjusts the rate at the output to accommodate the reduced number of samples. This parameter applies only when `InputProcessing` is `'columns as channels'`.

`'Arithmetic'` — Value types:
`'double'` | `'single'`

The arithmetic for System object inputs must be `'double'` or `'single'`.

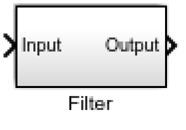
Examples

Realize Simulink Model of a Lowpass Butterworth Filter

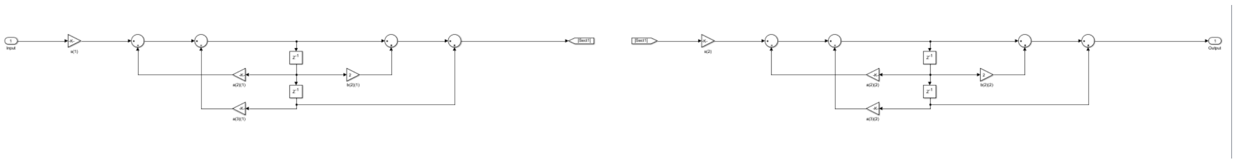
```
d = fdesign.lowpass('N,F3dB',4,0.25);
filterobject = design(d,'butter','systemobject',true);
```

Create a new model, `LPFilter.slx`, and realize the subsystem block in this model.

```
new_system('LPFilter');
realizemdl(filterobject);
```

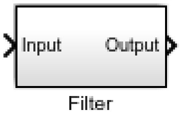


View the block diagram by clicking on the subsystem block.

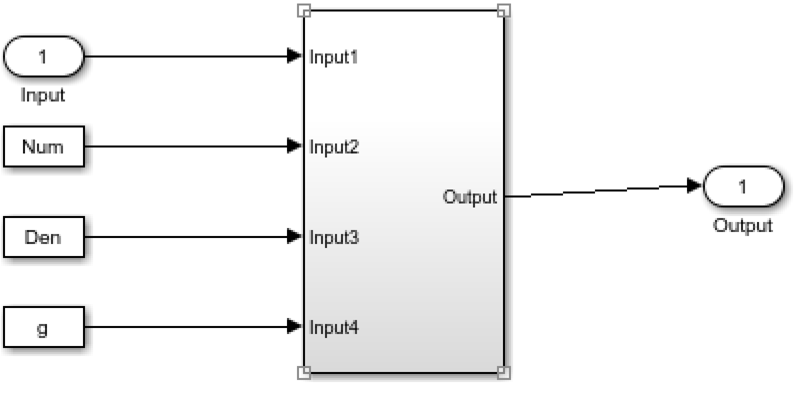


Create a new model, `LPFilterMapping.slx`, and realize the subsystem block, with coefficients mapped to ports, in this model.

```
new_system('LPFilterMapping');
realizemdl(filterobject,'MapCoeffsToPorts','on');
```



View the block diagram by clicking on the subsystem block.



In this case, the filter is an IIR filter with a direct form II second-order sections structure. Setting MapCoeffstoPorts to 'on' exports the numerator coefficients, the denominator coefficients, and the gains to the MATLAB® workspace using the default variable names Num, Den, and g. Each column of Num and Den represents one second-order section. You can modify the filter coefficients directly in the MATLAB workspace providing tunability to the realized Simulink model.

See Also

`design` | `fdesign`

Introduced in R2011a

rebuffer_delay

Number of samples of delay introduced by buffering and unbuffering operations

Syntax

```
d = rebuffer_delay(f,n,v)
d = rebuffer_delay(f,n,v,'mode')
```

Description

`d = rebuffer_delay(f,n,v)` returns the delay, in samples, introduced by the `Buffer` or `Unbuffer` block in multitasking operations.

`d = rebuffer_delay(f,n,v,'mode')` returns the delay, in samples, introduced by the Buffer or Unbuffer block in the specified tasking mode.

Input Arguments

f

Frame size of the input to the Buffer or Unbuffer block.

n

Size of the output buffer. Specify one of the following:

- The value of the **Output buffer size** parameter, if you are computing the delay introduced by a Buffer block.
- `1`, if you are computing the delay introduced by an Unbuffer block.

v

Amount of buffer overlap. Specify one of the following:

- The value of the **Buffer overlap** parameter, if you are computing the delay introduced by a Buffer block.

- `0`, if you are computing the delay introduced by an Unbuffer block.

`'mode'`

The tasking mode of the model. Specify one of the following options:

- `'singletasking'`
- `'multitasking'`

Default: `'multitasking'`

Examples

Compute the delay introduced by a Buffer block in a multitasking model:

1. Open a model containing a Buffer block. For this example, open the `ex_buffer_tut4` model by typing ex_buffer_tut4 at the MATLAB command line.
2. Double-click the Buffer block to open the block mask. Verify that you have the following settings:

   - **Output buffer size** = 3
   - **Buffer overlap** = 1
   - **Initial conditions** = 0

   Based on these settings, two of the required inputs to the `rebuffer_delay` function are as follows:

   - `n = 3`
   - `v = 1`
3. To determine the frame size of the input signal to the Buffer block, open the Signal From Workspace block mask. Verify that you have the following settings:

   - **Signal** = `sp_examples_src`
   - **Sample time** = `1`
   - **Samples per frame** = `4`

   Because **Samples per frame** = `4`, you know the `f` input to the `rebuffer_delay` function is `4`.

4 After you verify the values of all the inputs to the `rebuffer_delay` function, determine the delay that the Buffer block introduces in this multitasking model. To do so, type the following at the MATLAB command line:

```
d = rebuffer_delay(4,3,1)

d =
     8
```

Compute the delay introduced by an Unbuffer block in a multitasking model:

1 Open a model containing an Unbuffer block. For this example, open the `ex_unbuffer_ref1` model by typing ex_unbuffer_ref1 at the MATLAB command line.
2 To determine the frame size of the input to the Buffer block, open the Signal From Workspace block mask by double-clicking the block in your model. Verify that you have the following settings:

   - **Signal** = `sp_examples_src`
   - **Sample time** = `1`
   - **Samples per frame** = `3`

   Because **Samples per frame** = `3`, you know the `f` input to the `rebuffer_delay` function is `3`.
3 Use the `rebuffer_delay` function to determine the amount of delay that the Unbuffer block introduces in this multitasking model. To compute the delay introduced by the Unbuffer block, use `f = 3`, `n = 1` and `v = 0`.

```
d = rebuffer_delay(3,1,0)

d =
     3
```

Definitions

Multitasking

When you run a model in `MultiTasking` mode, Simulink processes groups of blocks with the same execution priority through each stage of simulation based on task priority.

Multitasking mode helps to create valid models of real-world multitasking systems, where sections of your model represent concurrent tasks. The **Treat each discrete rate as a separate task** parameter on the Solver (Simulink) pane of the Configuration Parameters dialog box controls this setting.

Singletasking

When you run a model in `SingleTasking` mode, Simulink processes all blocks through each stage of simulation together. The **Treat each discrete rate as a separate task** parameter on the Solver (Simulink) pane of the Configuration Parameters dialog box controls this setting.

See Also

`Buffer` | `Unbuffer`

Topics

"Buffer Delay and Initial Conditions"

Introduced before R2006a

reffilter

Reference filter for fixed-point or single-precision filter

Syntax

```
href = reffilter(hd)
```

Description

`href = reffilter(hd)` returns a new filter `href` that has the same structure as `hd`, but uses the reference coefficients and has its arithmetic property set to `double`. Note that `hd` can be either a fixed-point filter (arithmetic property set to '`fixed`', or a single-precision floating-point filter whose arithmetic property is '`single`').

`reffilter(hd)` differs from `double(hd)` in that

- the filter `href` returned by `reffilter` has the reference coefficients of `hd`.
- `double(hd)` returns the quantized coefficients of `hd` represented in double-precision.

To check the performance of your fixed-point filter, use `href` = `reffilter(hd)` to quickly have the floating-point, double-precision version of `hd` available for comparison.

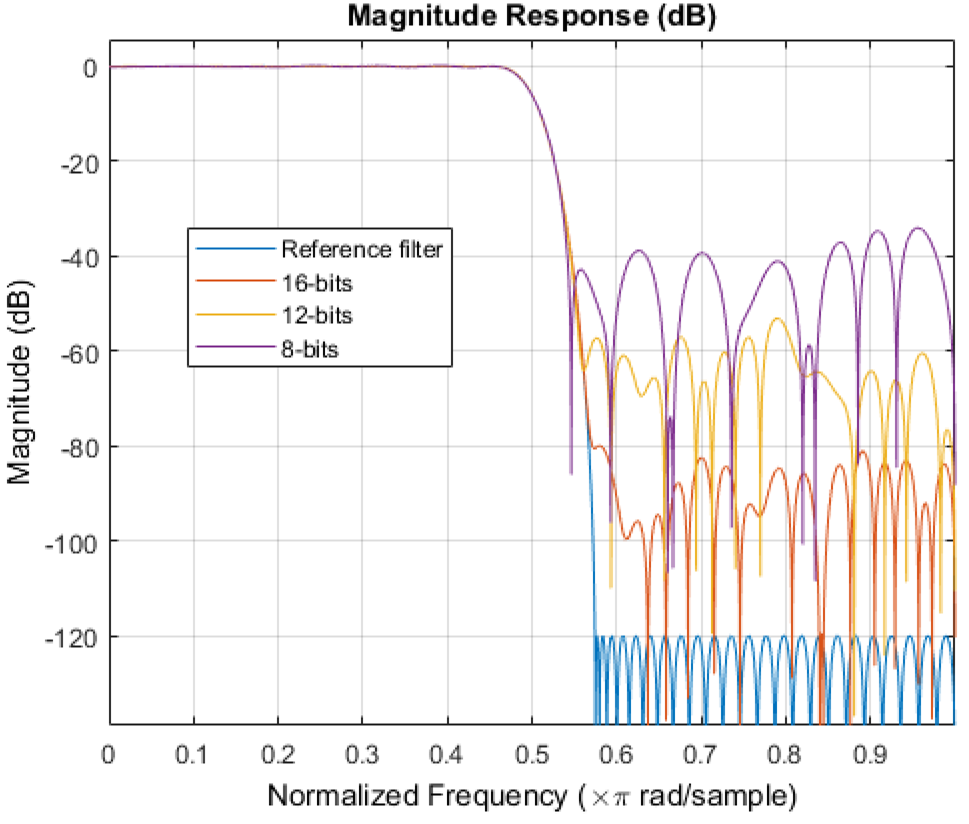
Examples

Compare Fixed-point Quantizations of a Filter

Compare several fixed-point quantizations of a filter with the same double-precision floating-point version of the filter.

```
h = dfilt.dffir(firceqrip(87,.5,[1e-3,1e-6])); % Lowpass filter.
h1 = copy(h); h2 = copy(h); % Create copies of h.
h.arithmetic = 'fixed';    % Set h to filter using fixed-point...
                           % arithmetic.
h1.arithmetic = 'fixed';   % Same for h1.
```

```
h2.arithmetic = 'fixed';  % Same for h2.
h.CoeffWordLength  = 16;  % Use 16 bits to represent the...
                          % coefficients.
h1.CoeffWordLength = 12;  % Use 12 bits to represent the...
                          % coefficients.
h2.CoeffWordLength =  8;  % Use 8 bits to represent the...
                          % coefficients.
href = reffilter(h);
hfvt = fvtool(href,h,h1,h2);
set(hfvt,'ShowReference','off'); % Reference displayed once
                                 % already.
legend(hfvt,'Reference filter','16-bits','12-bits','8-bits');
```



The fvtool shows `href`, the reference filter, and the effects of using three different word lengths to represent the coefficients.

See Also

`double`

Introduced in R2011a

reorder

Rearrange sections in SOS filter

Syntax

```
reorder(hs,order)
reorder(hs,numorder,denorder)
reorder(hs,numorder,denorder,svorder)
reorder(hs,filter_type)
reorder(hs,dir_flag)
reorder(hs,dir_flag,sv)
reorder(hs,...)
reorder(hs,...,Name,Value)
```

Description

`reorder(hs,order)` rearranges the sections of filter `hd` using the vector of indices provided in `order`.

`reorder(hs,numorder,denorder)` reorders the numerator and denominator separately using the vectors of indices in `numorder` and `denorder`.

`reorder(hs,numorder,denorder,svorder)` specifies that the scale values can be independently reordered.

`reorder(hs,filter_type)` reorders `hd` in a way suitable for the specified filter type. This reordering mode can be especially helpful for fixed-point implementations where the order of the filter sections can significantly affect your filter performance.

`reorder(hs,dir_flag)` specifies rearranges the sections according to proximity to the origin of the poles of the sections.

`reorder(hs,dir_flag,sv)` reorders scale values in addition to rearranging sections according to pole-origin proximity.

`reorder(hs,...)` rearranges the sections of the filter System object `hs` according to any of the preceding input arguments.

`reorder(hs,...,Name,Value)` rearranges the sections of the filter System object `hs` with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`hs`

`dsp.BiquadFilter` filter System object with one of the following filter structures:

Structure	Description
`df1sos`	Direct-form I filter object with second-order sections.
`df1tsos`	Direct-Form I transposed filter with second-order sections.
`df2sos`	Direct-form II filter object with second-order sections.
`df2tsos`	Direct-Form II transposed filter with second-order sections.

`order`

Vector of indices used to reorder the filter sections. `order` does not need to contain all of the indices of the filter. Omitting one or more filter section indices removes the omitted sections from the filter. You can use a logical array to remove sections from the filter, but not to reorder it.

`numorder`

Vector of indices used to reorder the numerator. `numorder` and `denorder` must be the same length.

`denorder`

Vector of indices used to reorder the numerator. `numorder` and `denorder` must be the same length.

svorder

Independent reordering of scale values. When `svorder` is not specified, the scale values are reordered with the numerator. The output scale value always remains on the end when you use the argument `numorder` to reorder the scale values.

filter_type — Different types of filters:
'auto' | 'bandpass' | 'bandstop' | 'highpass' | 'lowpass'

Filter type. The `'auto'` option and automatic ordering only apply to filters that you used `fdesign` to create. With the `'auto'` option as an input argument, `reorder` automatically rearranges the filter sections depending on the specification response type of the design.

dir_flag — Direction options:
'down' | 'up'

Pole direction flag. When `dir_flag` is `'up'`, the first filter section contains the poles closest to the origin, and the last section contains the poles closest to the unit circle. When `ir_flag` is `'down'`, the sections are ordered in the opposite direction. `reorder` always pairs zeros with the poles closest to them.

sv — Scale value options:
'poles' | 'zeros'

Reorder scale values according to poles or zeros. By default the scale values are not reordered when you use the `dir_flag` input argument.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'Arithmetic' — Value types:
‘double’ | 'single' | 'fixed'

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the CoefficientDataType property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type.

Examples

Reorder a Biquad Filter

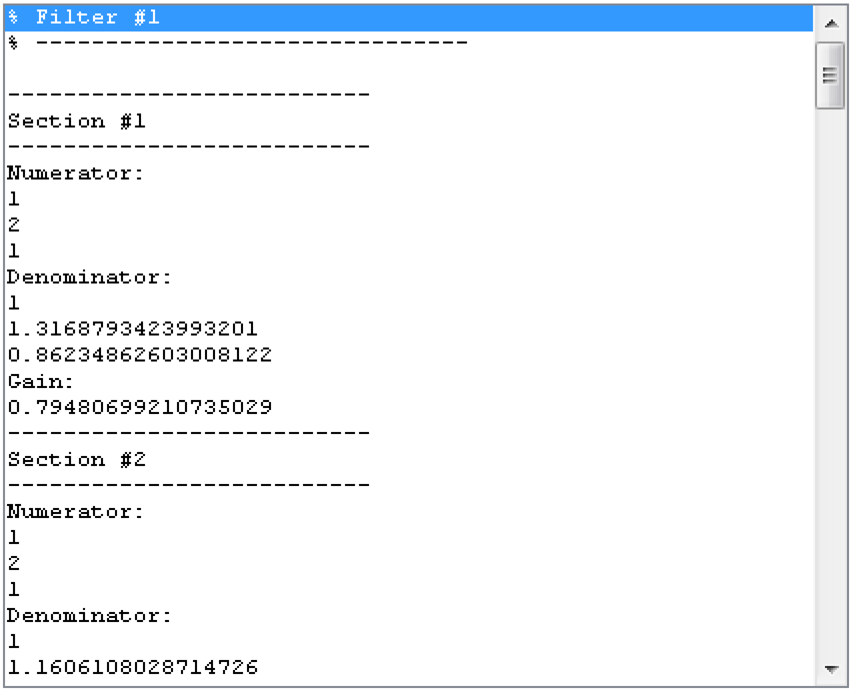
Being able to rearrange the order of the sections in a filter can be a powerful tool for controlling the filter process. This example uses reorder to change the sections of a df2sos filter. Let reorder do the reordering automatically in the first example. In the second, use reorder to specify the new order for the sections.

First use the automatic reordering option on a lowpass filter.

```
d = fdesign.lowpass('n,f3db',15,0.75);
biquad = design(d,'butter','SystemObject',true);
biquadreorder = reorder(biquad,'auto');
```

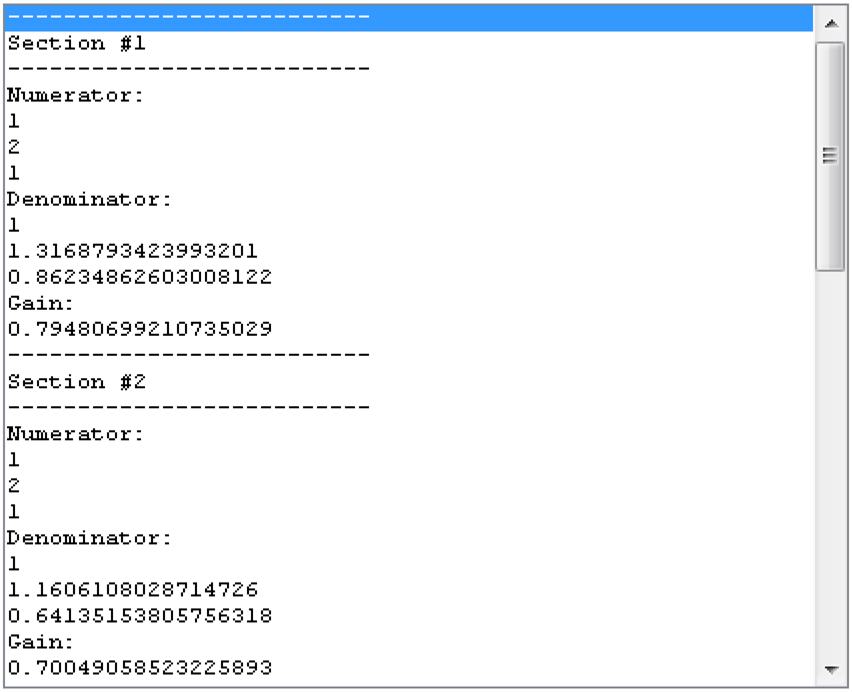
For another example of using reorder, create an SOS filter in the direct form II implementation.

```
biquad2sos = design(d,'butter', 'FilterStructure', 'df2sos',...
    'SystemObject',true);
biquad2sosreorder = reorder(biquad2sos,[1 3:7 2 8]);
fvt = fvtool(biquad2sos,biquad2sosreorder,'analysis','coefficients');
```



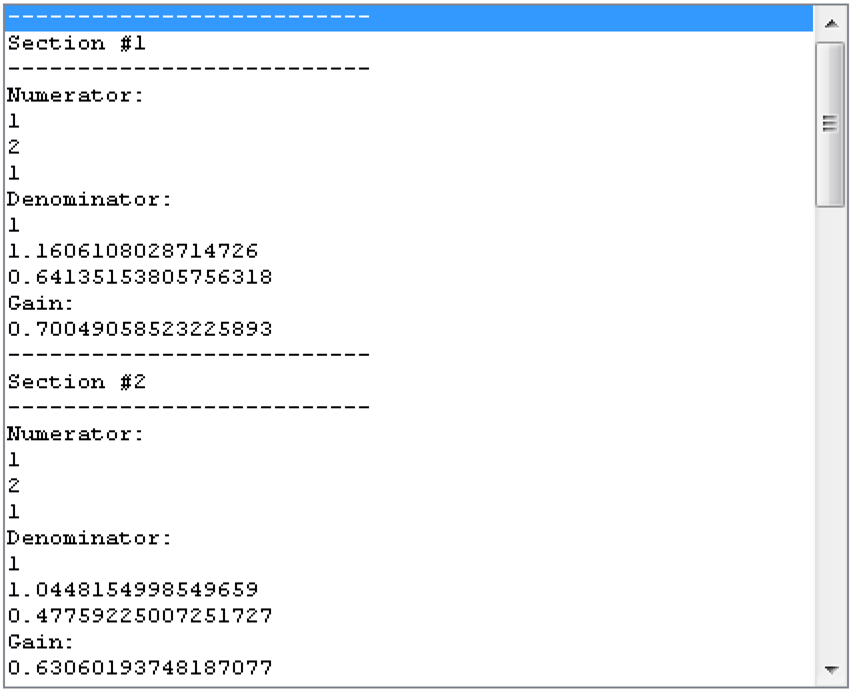
Remove the third, fourth, and seventh sections.

```
biquad2sosclone1 = clone(biquad2sos);
reorder(biquad2sosclone1, logical([1 1 0 0 1 1 0 1]));
setfilter(fvt, biquad2sosclone1);
```



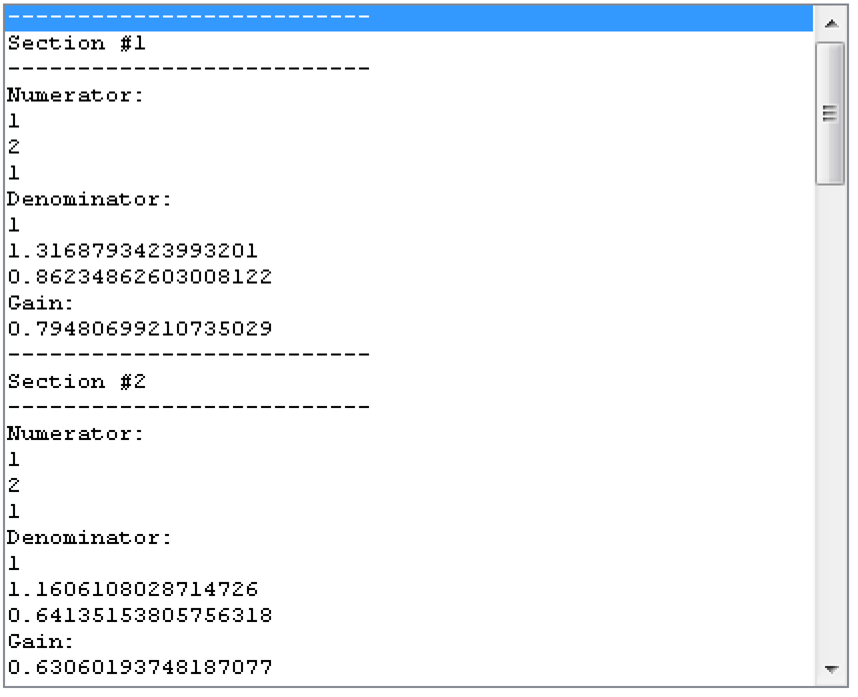
Move the first filter to the end, and remove the eighth section.

```
biquad2sosclone2 = clone(biquad2sos);
reorder(biquad2sosclone2, [2:7 1]);
setfilter(fvt, biquad2sosclone2);
```



Move the numerator and denominator independently.

```
biquad2sosclone3 = clone(biquad2sos);
reorder(biquad2sosclone3, [1 3:8 2], [1:8]);
setfilter(fvt, biquad2sosclone3);
```



References

Schlichthärle, Dietrich, *Digital Filters Basics and Design*, Springer-Verlag Berlin Heidelberg, 2000.

See Also

`cumsec` | `scale` | `scaleopts`

Introduced in R2011a

reset

Reset the internal states of a System object

Syntax

```
reset(sysobj)
```

Description

`reset(sysobj)` resets the internal states of System object, `sysobj` to their initial values. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the filter to nonzero input data, the states might be nonzero. Invoking the `step` method again without first invoking the `reset` method might produce different outputs for an identical input. System objects with no states are unaffected by `reset`.

Examples

Reset a Lowpass Filter

Create a lowpass filter with default properties.

```
LPF = dsp.LowpassFilter;
```

Create a two-channel random signal. Apply the `step` method twice on the signal.

```
x = randn(10,2);

y1 = step(LPF,x);
y2 = step(LPF,x);

no = all(y2==y1)
```

```
no =

  1×2 logical array
```

```
   0    0
```

The output is different because the internal states of LPF have changed. Use reset to reset the lowpass filter and apply step again. Verify that the output is unchanged.

```
reset(LPF)

y3 = step(LPF,x);

yes = all(y3==y1)
```

```
yes =

  1×2 logical array

   1    1
```

See Also

set

Introduced in R2011a

scale

Scale second-order sections of `dsp.BiquadFilter` System object

Syntax

```
scale(biquad)
biquadnew = scale(biquad)
scale(biquad,pnorm)
scale(biquad,pnorm,Name,Value)
scale(biquad,pnorm,opts)
scale(biquad,'Arithmetic',ARITH)
```

Description

`scale(biquad)` scales the `dsp.BiquadFilter` System object, `biquad`, using peak magnitude response scaling (L-infinity, `'Linf'`). This scaling reduces the possibility of overflows when your filter `biquad` operates in fixed-point arithmetic mode.

`biquadnew = scale(biquad)` generates a new filter System object, `biquadnew`, with scaled second-order sections. The original filter System object, `biquad`, is not changed.

`scale(biquad,pnorm)` specifies the norm used to scale the filter. Pnorm can be either a discrete-time-domain norm or a frequency-domain norm. Valid time-domain norms are `'l1'`, `'l2'`, and `'linf'`. Valid frequency-domain norms are `'L1'`, `'L2'`, and `'Linf'`. Note that L2-norm is equal to l2-norm (Parseval's theorem) but the same is not true for other norms.

The different norms can be ordered in terms of how stringent they are as follows: `'l1' >= 'Linf' >= 'L2' = 'l2' >= 'L1' >= 'linf'`.

Using the most stringent scaling, `'l1'`, the filter is the least prone to overflow, but also has the worst signal-to-noise ratio. Linf-scaling is the most commonly used scaling in practice.

`scale(biquad,pnorm,Name,Value)` specifies optional scaling parameters via by one or more `Name,Value` pair arguments.

scale(biquad,pnorm,opts) uses an options object to specify the optional scaling parameters in lieu of specifying parameter-value pairs. The opts object can be created using the scaleopts method: opts = scaleopts(biquad).

scale(biquad,'Arithmetic',ARITH) assumes that the filter arithmetic is equal to ARITH. ARITH can be set to one of 'double', 'single', or 'fixed'. The scale method assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. If 'Arithmetic' is 'double' or 'single', the default values are used for all scaling options that are not specified as an input to the scale method. If 'Arithmetic' is 'fixed', the values used for the scaling options are set according to the settings in the filter System object, biquad. However, if a scaling option is specified that differs from the settings in biquad, this option is used for scaling purposes but does not change the setting in biquad. For example, if you do not specify the 'OverflowMode' scaling option, the scale method assumes that the 'OverflowMode' is equal to the value in the OverflowAction property of the System object, biquad. If you do specify an 'OverflowMode' scaling option, then the scale method uses this overflow mode value regardless of the value in the OverflowAction property of the System object.

Input Arguments

biquad

dsp.BiquadFilter filter System object.

pnorm

Discrete-time-domain norm or a frequency-domain norm.

Valid time-domain norm values for pnorm are 'l1', 'l2', and 'linf'. Valid frequency-domain norm values are 'L1', 'L2', and 'Linf' . The 'L2' norm is equal to the 'l2' norm (by Parseval's theorem), but this equivalency does not hold for other norms — 'l1' is not the same as 'L1' and 'Linf' is not the same as 'linf'.

Filter norms can be ordered in terms of how stringent they are, as follows from most stringent to least: 'l1', 'Linf', 'l2' ('L2'), 'linf'. Using 'l1', the most stringent scaling, produces a filter that is least likely to overflow, but has the worst signal-to-noise ratio performance. The default scaling 'Linf' (default) is the most commonly used scaling norm.

opts

Scale options object. You can create the `opts` object using the `scaleopts` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'Arithmetic'

Specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.

System Object State	Coefficient Data Type	Rule
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type.

'MaxNumerator'

Maximum allowed value for numerator coefficients.

Default: 2

'MaxScaleValue'

Maximum allowed scale values. The filter applies the MaxScaleValue limit only when you set ScaleValueConstraint to a value other than unit (the default setting). Setting MaxScaleValue to any numerical value automatically changes the ScaleValueConstraint setting to none.

Default: 'Not Used'

'NumeratorConstraint'

Specifies whether and how to constrain numerator coefficient values. Possible options:

- 'none' (default)
- 'normalized'
- 'po2'
- 'unit'

'OverflowMode'

Sets the way the filter handles arithmetic overflow situations during scaling. If your device does not have guard bits available, and you are using saturation arithmetic for filtering, use 'satall' instead of 'saturate'. The default is 'wrap'.

'ScaleValueConstraint'

Specify whether to constrain the filter scale values, and how to constrain them. Choosing `'unit'` for the constraint disables the `MaxScaleValue` property setting. `'po2'` constrains the scale values to be powers of 2, while `'none'` removes any constraint on the scale values. `'unit'` is the default value.

'sosReorder'

Reorder filter sections prior to applying scaling. Possible options:

- `'auto'` (default)
- `'none'`
- `'up'`
- `'down'`
- `'lowpass'`
- `'highpass'`
- `'bandpass'`
- `'bandstop'`

Examples

Linf-norm Scaling of a Biquad Filter

Demonstrate the Linf-norm scaling of a biquad filter using the `scale` function.

```
Fs = 8000; Fcutoff = 2000;
[z,p,k] = butter(10,Fcutoff/(Fs/2)); [s,g] = zp2sos(z,p,k);
biquad = dsp.BiquadFilter('Structure', 'Direct form I', ...
    'SOSMatrix', s,'ScaleValues', g);
scale(biquad,'linf','scalevalueconstraint','none','maxscalevalue',2)
```

References

[1] Dehner, G.F. “Noise Optimized Digital Filter Design: Tutorial and Some New Aspects.” *Signal Processing*. Vol. 83, Number 8, 2003, pp. 1565–1582.

See Also

See Also

Functions

`cumsec` | `norm` | `reorder` | `scalecheck` | `scaleopts`

Introduced in R2011a

scalecheck

Check scaling of SOS filter

Syntax

```
s = scalecheck(biquad,pnorm)
s = scalecheck(biquad,pnorm,Name,Value)
```

Description

`s = scalecheck(biquad,pnorm)` checks the scaling of the filter System object `biquad`.

`s = scalecheck(biquad,pnorm,Name,Value)` checks the scaling of the filter System object `biquad` with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`biquad`

`dsp.BiquadFilter` filter System object with one of the following filter structures:

Structure	Description
`df1sos`	Direct-form I filter object with second-order sections.
`df1tsos`	Direct-Form I transposed filter with second-order sections.
`df2sos`	Direct-form II filter object with second-order sections.
`df2tsos`	Direct-Form II transposed filter with second-order sections.

pnorm — Different types of norm:
'l1' | 'l2' | 'linf' | 'L1' | 'L2' | 'Linf'

Discrete-time-domain norm or a frequency-domain norm.

Valid time-domain norm values for pnorm are 'l1', 'l2', and 'linf'. Valid frequency-domain norm values are 'L1', 'L2', and 'Linf'. The 'L2' norm is equal to the 'l2' norm (by Parseval's theorem), but this equivalency does not hold for other norms — 'l1' is not the same as 'L1' and 'Linf' is not the same as 'linf'.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

'Arithmetic' — Value type:
‘double’ | 'single' | 'fixed'

For filter System object inputs only, specify the arithmetic used during analysis. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed' , the arithmetic changes depending on the setting of the CoefficientDataType property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

System Object State	Coefficient Data Type	Rule
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type.

Output Arguments

s

Filter scaling for a given p-norm. An optimally scaled filter has partial norms equal to one. In such cases, `s` contains all ones.

For direct-form I (`df1sos`) and direct-form II transposed (`df2tsos`) filters, the function returns the p-norm of the filter computed from the filter input to the output of each second-order section. Therefore, the number of elements in `s` is one less than the number of sections in the filter. This p-norm computation does not include the trailing scale value of the filter, which you can find by entering `hd.scalevalue(end)` at the MATLAB prompt.

For direct-form II (`df2sos`) and direct-form I transposed (`df1tsos`) filters, the function returns a row vector whose elements contain the p-norm from the filter input to the input of the recursive part of each second-order section. This computation of the p-norm corresponds to the input to the multipliers in these filter structures. These inputs correspond to the locations in the signal flow where overflow should be avoided.

When hd has nontrivial scale values, that is, if any scale values are not equal to one, s is a two-row matrix, rather than a vector. The first row elements of s report the p-norm of the filter computed from the filter input to the output of each second-order section. The elements of the second row of s contain the p-norm computed from the input of the filter to the input of each scale value between the sections. For df2sos and df1tsos filter structures, the last numerator and the trailing scale value for the filter are not included when scalecheck checks the scaling.

Examples

Linf-norm scaling of a filter

This example shows how to check the Linf-norm scaling of a filter.

Design an elliptic sos filter in the direct form II structure with default specifications.

```
EllipII = design(fdesign.lowpass, 'ellip', 'FilterStructure', 'df2sos',...
    'SystemObject',true);
```

Check the scaling.

```
scalecheck(EllipII,'Linf')


ans =

    3.1678   15.0757    1.4974
    4.7360   52.6026    1.0000
```

Design an elliptic sos filter in the direct form I structure with default specifications.

```
EllipI = design(fdesign.lowpass('N,Fp,Ap,Ast',10,0.5,0.5,20), 'ellip',...
    'FilterStructure', 'df1sos','SystemObject',true);
```

Check the scaling.

```
scalecheck(EllipI,'Linf')


ans =

    1.7078    2.0807    2.6084    7.1467    1.0000
```

See Also

`norm` | `reorder` | `scale` | `scaleopts`

Introduced in R2011a

scaleopts

Create an options object for second-order section scaling

Syntax

```
opts = scaleopts(biquad)
opts = scaleopts(biquad,Name,Value)
```

Description

`opts = scaleopts(biquad)` uses the current settings in the `dsp.BiquadFilter` System object, `biquad`, to create an options object `opts` that contains specified scaling options for second-order section scaling. You can pass `opts` as an input to `scale` to apply scaling settings to a second-order filter.

`opts = scaleopts(biquad,Name,Value)` creates an options object with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`biquad`

`dsp.BiquadFilter` filter System object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Arithmetic'` — Value types:
`‘double’ | 'single' | 'fixed'`

Specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type.

Output Arguments

opts

Scaling object. The following table lists the properties of `opts`.

Parameter	Default	Description and Valid Value
`sosReorder`	`'auto'`	Reorder section prior to scaling. Valid options are `'auto'` (default), `'none'`, `'up'`, `'down'`, `'lowpass'`, `'highpass'`, `'bandpass'`, and `'bandstop'`.
`MaxNumerator`	`2`	Maximum allowed value for numerator coefficients.
`NumeratorConstraint`	`'none'`	Specifies whether and how to constrain numerator coefficient values. Options are `'none'` (default), `'unit'`, `'normalize'`, and `'po2'`.
`OverflowMode`	`'wrap'`	Sets the way the filter handles arithmetic overflow situations during scaling. Valid options are `'wrap'` (default), `'saturate'`, and `'satall'`.
`ScaleValueConstraint`	`'unit'`	Specify whether to constrain the filter scale values, and how to constrain them. Valid options are `'unit'` (default), `'none'`, and `'po2'`.
`MaxScaleValue`	`'Not used'`	Maximum allowed scale values. The filter applies the `MaxScaleValue` limit only when you set `ScaleValueConstraint` to a value other than `unit`. Setting `MaxScaleValue` to a numerical value

Parameter	Default	Description and Valid Value
		automatically changes the ScaleValueConstraint setting to none.

Examples

Options for scaling SOS filter

Create an options scaling object that contains the scaling options settings you require.

```
EllipI = design(fdesign.lowpass('N,Fp,Ap,Ast',10,0.5,0.5,20), 'ellip', 'FilterStructure
opts = scaleopts(EllipI)
```

```
EllipI =

  dsp.BiquadFilter with properties:

                     Structure: 'Direct form I'
               SOSMatrixSource: 'Property'
                     SOSMatrix: [5×6 double]
                   ScaleValues: [6×1 double]
    NumeratorInitialConditions: 0
  DenominatorInitialConditions: 0
      OptimizeUnityScaleValues: true

  Use get to show all properties


opts =

             sosReorder: 'auto'
           MaxNumerator: 2
    NumeratorConstraint: 'none'
           OverflowMode: 'wrap'
   ScaleValueConstraint: 'unit'
          MaxScaleValue: 'Not used'
```

See Also

See Also

Functions

`cumsec` | `norm` | `reorder` | `scale` | `scalecheck`

Introduced in R2011a

set2int

Configure filter for integer filtering

Syntax

```
set2int(h)
set2int(h,coeffwl)
set2int(...,inwl)
g = set2int(...)
```

Description

This section applies to discrete-time (`dfilt`) filters.

`set2int(h)` scales the filter coefficients to integer values and sets the filter coefficient and input fraction lengths to zero.

`set2int(h,coeffwl)` uses the number of bits specified by `coeffwl` as the word length it uses to represent the filter coefficients.

`set2int(...,inwl)` uses the number of bits specified by `coeffwl` as the word length it uses to represent the filter coefficients and the number of bits specified by `inwl` as the word length to represent the input data.

`g = set2int(...)` returns the gain `g` introduced into the filter by scaling the filter coefficients to integers. `g` is always calculated to be a power of 2.

Note `set2int` does not work with CIC decimators or interpolators because they do not have coefficients.

Examples

Configure an FIR Filter for Integer Filtering

Two parts comprise this example. Part 1 compares the step response of an FIR filter in both the fractional and integer filter modes. Fractional mode filtering is essentially the opposite of integer mode. Integer mode uses a filter which has coefficients represented by integers. Fractional mode filters have coefficients represented in fractional form (nonzero fraction length).

The second part of the example depends on the following - after you filter a set of data, the input data and output data cover the same range of values, unless the filter process introduces gain in the output. Converting your filter object to integer form, and then filtering a set of data, does introduce gain into the system. When the examples refer to resetting the output to the same range as the input, the examples are accounting for this added gain feature.

```
b = rcosdesign(.25,4,25,'sqrt');
hd = dfilt.dffir(b);
hd.Arithmetic = 'fixed';
hd.InputFracLength = 0; % Integer inputs.
x = ones(100,1);
yfrac = filter(hd,x); % Fractional mode output.
g = set2int(hd);      % Convert to integer coefficients.
yint = filter(hd,x);  % Integer mode output.
```

Note that `yint` and `yfrac` are `fi` objects. Later in this example, use the `fi` object properties `WordLength` and `FractionLength` to work with the output data. Now use the gain `g` to rescale the output from the integer mode filter operation. Verify that the scaled integer output is equal to the fractional output.

```
yints = double(yint)/g;
```

Verify that the scaled integer output is equal to the fractional output.

```
max(abs(yints-double(yfrac)))


ans =

     0
```

In part two, the example reinterprets the output binary data, putting the input and the output on the same scale by weighting the most significant bits in the input and output data equally.

```
WL = yint.WordLength;
FL = yint.FractionLength + log2(g);
yints2 = fi(zeros(size(yint)),true,WL,FL);
```

```
yints2.bin = yint.bin;
max(abs(double(yints2)-double(yfrac)))

ans =

     0
```

Introduced in R2011a

setspecs

Specifications for filter specification object

Syntax

```
setspecs(D,specvalue1,specvalue2,...)
setspecs(D,Specification,specvalue1,specvalue2,...)
setspecs(...Fs)
setspecs(...,MAGUNITS)
```

Description

`setspecs(D,specvalue1,specvalue2,...)` sets the specifications in filter specification object, `D`, in the same order they appear in the `Specification` property.

`setspecs(D,Specification,specvalue1,specvalue2,...)` changes the specifications for an existing filter specification object and sets values for the new `Specification` property.

`setspecs(...Fs)` specifies the sampling frequency, `Fs`, in Hz. The sampling frequency must be a scalar trailing all other specifications. Entering a sampling frequency causes all other frequency specifications to be in Hz.

`setspecs(...,MAGUNITS)` specifies the units for any magnitude specifications. `MAGUNITS` can be one of the following: `'linear'`, `'dB'`, or `'squared'`. The default is `'dB'`. The magnitude specifications are always converted and stored in dB regardless of how the units are specified.

Use `SET(D,'SPECIFICATION')` to get the list of all available specification types for the filter specification object, `D`.

Examples

Set the Filter Order and Cutoff Frequency Using `setspecs`

Construct a lowpass filter with specifications for the filter order and cutoff frequency (-6 dB). Use setspecs after construction to set the values of the filter order and cutoff frequency. Display the values in the MATLAB® command window.

```
D = fdesign.lowpass('N,Fc');
setspecs(D,10,0.2);
```

```
D.FilterOrder
```

```
ans =

    10
```

```
D.Fcutoff
```

```
ans =

    0.2000
```

Set the Specifications of a Highpass Filter Using `setspecs`

Construct a highpass filter with specifications for the numerator order, denominator order, and 3-dB frequency. Assume the sampling frequency is 1 kHz. Use setspecs to set the numerator and denominator orders to 6. Set the 3-dB frequency to 250 Hz. In order to use frequency specifications in Hz, specify the sampling frequency as a trailing scalar.

```
D = fdesign.highpass('Nb,Na,F3dB');
setspecs(D,6,6,250,1000);
```

See Also

`design` | `designmethods` | `fdesign` | `designopts`

Introduced in R2011b

sos

Convert quantized filter to second–order sections (SOS) form

Syntax

```
Hq2 = sos(Hq)
Hq2 = sos(Hq, order)
Hq2 = sos(Hq, order, scale)
```

Description

`Hq2 = sos(Hq)` returns a quantized filter `Hq2` that has second-order sections and the `dft2` structure. Use the same optional arguments used in `tf2sos`.

`Hq2 = sos(Hq, order)` specifies the order of the sections in `Hq2`, where `order` is one of the following options:

- `'down'` — to order the sections so the first section of `Hq2` contains the poles closest to the unit circle (L_∞ norm scaling)
- `'up'` — to order the sections so the first section of `Hq2` contains the poles farthest from the unit circle (L_2 norm scaling and the default)

`Hq2 = sos(Hq, order, scale)` also specifies the desired scaling of the gain and numerator coefficients of all second-order sections, where `scale` is one of the following options:

- `'none'` — to apply no scaling (default)
- `'inf'` — to apply infinity-norm scaling
- `'two'` — to apply 2-norm scaling

`Hq2` is a dsp.BiquadFilter filter System object.

Use infinity-norm scaling in conjunction with up-ordering to minimize the probability of overflow in the filter realization. Consider using 2-norm scaling in conjunction with down-ordering to minimize the peak round-off noise.

When Hq is a fixed-point filter, the filter coefficients are normalized so that the magnitude of the maximum coefficient in each section is 1. The gain of the filter is applied to the first scale value of Hq2.

sos uses the direct form II transposed (dft2) structure to implement second- order section filters.

Examples

```
[b,a]=butter(8,.5);
Hq = dfilt.df2t(b,a);
Hq.arithmetic = 'fixed';
Hq1 = sos(Hq)

Hq1 =

         FilterStructure: 'Direct-Form II Transposed, Second-Order Sections'
              Arithmetic: 'double'
               sosMatrix: [4x6 double]
             ScaleValues: [0.00927734375;1;1;1;1]
     OptimizeScaleValues: true
        PersistentMemory: false
```

See Also

convert | dfilt | tf2sos

Introduced in R2011a

specifyall

Fully specify fixed-point filter System object settings

Syntax

```
specifyall(sysobj)
specifyall(sysobj,false)
specifyall(sysobj,true)
```

Description

`specifyall(sysobj)` sets all the data type fixed-point properties of the filter System object `sysobj` to `'Custom'` so that you can easily specify all the fixed-point settings. If the object has a `FullPrecisionOverride` property, its value is set to `false`. `specifyall` is intended as a shortcut to changing all the fixed-point properties one by one.

`specifyall(sysobj,false)` sets all fixed-point properties of the filter System object `sysobj` to their default values and sets the filter to full-precision mode, if one is available.

`specifyall(sysobj,true)` is equivalent to `specifyall(sysobj)`.

Examples

Specify all Fixed-point Settings of an FIRFilter

This example demonstrates using `specifyall` to provide access to all of the fixed-point settings of an FIR filter implemented with the direct-form structure. Using `specifyall` disables all of the automatic filter scaling and reset the mode values.

```
b = fircband(12,[0 0.4 0.5 1],[1 1 0 0],[1 0.2],{'w' 'c'});
firFilter = dsp.FIRFilter('Numerator',b);
get(firFilter)
```

```
ans = 

  struct with fields:

                              Numerator: [1×13 double]
                  ReflectionCoefficients: [0.5000 0.5000]
                       InitialConditions: 0
                         NumeratorSource: 'Property'
            ReflectionCoefficientsSource: 'Property'
                               Structure: 'Direct form'
                   FullPrecisionOverride: 1
                          RoundingMethod: 'Floor'
                          OverflowAction: 'Wrap'
                    CoefficientsDataType: 'Same word length as input'
          ReflectionCoefficientsDataType: 'Same word length as input'
                CustomCoefficientsDataType: [1×1 embedded.numerictype]
    CustomReflectionCoefficientsDataType: [1×1 embedded.numerictype]
                         ProductDataType: 'Full precision'
                   CustomProductDataType: [1×1 embedded.numerictype]
                     AccumulatorDataType: 'Full precision'
               CustomAccumulatorDataType: [1×1 embedded.numerictype]
                           StateDataType: 'Same as accumulator'
                     CustomStateDataType: [1×1 embedded.numerictype]
                          OutputDataType: 'Same as accumulator'
                    CustomOutputDataType: [1×1 embedded.numerictype]
```

`specifyall` sets all the data type fixed-point properties of the FIR filter to `'Custom'`.

```
specifyall(firFilter)
get(firFilter)
```

```
ans = 

  struct with fields:

                              Numerator: [1×13 double]
                  ReflectionCoefficients: [0.5000 0.5000]
                       InitialConditions: 0
                         NumeratorSource: 'Property'
            ReflectionCoefficientsSource: 'Property'
                               Structure: 'Direct form'
                   FullPrecisionOverride: 0
                          RoundingMethod: 'Floor'
```

```
                           OverflowAction: 'Wrap'
                     CoefficientsDataType: 'Custom'
           ReflectionCoefficientsDataType: 'Custom'
               CustomCoefficientsDataType: [1×1 embedded.numerictype]
     CustomReflectionCoefficientsDataType: [1×1 embedded.numerictype]
                          ProductDataType: 'Custom'
                    CustomProductDataType: [1×1 embedded.numerictype]
                      AccumulatorDataType: 'Custom'
                CustomAccumulatorDataType: [1×1 embedded.numerictype]
                            StateDataType: 'Custom'
                      CustomStateDataType: [1×1 embedded.numerictype]
                           OutputDataType: 'Custom'
                     CustomOutputDataType: [1×1 embedded.numerictype]
```

See Also

See Also

Functions
`double` | `reffilter`

Introduced in R2011a

stepz

Step response of discrete-time filter System object

Syntax

```
[stepResp,t] = stepz(sysobj)
[stepResp,t] = stepz(sysobj,n)
[stepResp,t] = stepz(sysobj,n,fs)
[stepResp,t] = stepz(sysobj,[],fs)
[stepResp,t] = stepz(sysobj,Name,Value)
stepz(sysobj)
```

Description

`[stepResp,t] = stepz(sysobj)` computes the step response of the filter System object, `sysobj`, and returns the response in column vector `stepResp`, and a vector of times (or sample intervals) in `t`, where `t = [0 1 2 ...k-1]'`. `k` is the number of filter coefficients.

`[stepResp,t] = stepz(sysobj,n)` computes the step response at `floor(n)` 1-second intervals. The time vector `t` equals `(0:floor(n)-1)'`.

`[stepResp,t] = stepz(sysobj,n,fs)` computes the step response at `floor(n)` 1/`fs`-second intervals. The time vector `t` equals `(0:floor(n)-1)'/fs`.

`[stepResp,t] = stepz(sysobj,[],fs)` computes the step response at `k` 1/`fs`-second intervals. `k` is the number of filter coefficients. The time vector `t` equals `(0:k-1)'/fs`.

`[stepResp,t] = stepz(sysobj,Name,Value)` returns a step response with additional options specified by one or more `Name,Value` pair arguments.

`stepz(sysobj)` uses `fvtool` to plot the step response of the filter System object `sysobj`.

For more input options, refer to `stepz` in Signal Processing Toolbox documentation.

Note `stepz` works for both real and complex filters. When you omit the output arguments, `stepz` plots only the real part of the step response.

Input Arguments

`sysobj`

Filter System object.

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpassFilter
dsp.AllpoleFilter
dsp.BiquadFilter
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.CICDecimator
dsp.CICInterpolator
dsp.CoupledAllpassFilter
dsp.FarrowRateConverter
dsp.FilterCascade
dsp.FIRDecimator
dsp.FIRFilter
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.FIRInterpolator
dsp.FIRRateConverter
dsp.HighpassFilter
dsp.IIRFilter
dsp.IIRHalfbandDecimator

Filter System objects
dsp.IIRHalfbandInterpolator
dsp.LowpassFilter
dsp.NotchPeakFilter
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

n

Length of the step response vector.

Default: Number of filter coefficients

fs

Sampling frequency.

Default: 1

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'Arithmetic' — Value types:
'double' | 'single' | 'fixed'

Specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data

System Object State	Coefficient Data Type	Rule
		type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

stepResp

n-element step response vector. If n is not specified, the length of the step response vector equals the number of coefficients in the filter.

t

Time vector of length `n`, in seconds. `t` consists of `n` equally spaced points in the range `(0:floor(n)-1)'/fs`. If `n` is not specified, the function uses the number of coefficients of the filter.

See Also

See Also

`freqz` | `impz`

Introduced in R2011a

sysobj

Create filter System object from discrete-time filter

Syntax

```
hs = sysobj(hfilt)
```

Description

`hs = sysobj(hfilt)` creates a new filter System object `hs` from the `dfilt` object, `hfilt`.

The function supports a subset of `dfilt` objects. The following table lists supported filter structures for `hfilt` and the filter System object that the function creates.

Single-rate	Filter System object
Lattice AR(`dfilt.latticear`)	`dsp.AllpoleFilter`
Coupled-allpass, power-complementary lattice filter (`dfilt.calatticepc`)	dsp.CoupledAllpassFilter
Coupled-allpass, lattice filter (`dfilt.calattice`)	dsp.CoupledAllpassFilter
Cascade of discrete time filters (`dfilt.cascade`)	dsp.CoupledAllpassFilter
Direct Form I (`dfilt.df1`)	`dsp.IIRFilter`
Direct Form I transposed (`dfilt.df1t`)	`dsp.IIRFilter`
Direct Form II (`dfilt.df2`)	`dsp.IIRFilter`
Direct Form II transposed (`dfilt.df2t`)	`dsp.IIRFilter`
Direct-form FIR (`dfilt.dffir`)	dsp.FIRFilter
Direct-form FIR transposed (`dfilt.dffirt`)	dsp.FIRFilter
Direct-form symmetric FIR (`dfilt.dfsymfir`)	dsp.FIRFilter

Single-rate	Filter System object
Direct-form antisymmetric FIR (`dfilt.dfasymfir`)	dsp.FIRFilter
Discrete-time, lattice, moving-average (`dfilt.latticemamin`)	dsp.FIRFilter
Discrete-time, second-order section, direct-form I (`dfilt.df1sos`)	dsp.BiquadFilter
Discrete-time, second-order section, direct-form I transposed (`dfilt.df1tsos`)	dsp.BiquadFilter
Discrete-time, second-order section, direct-form II (`dfilt.df2sos`)	dsp.BiquadFilter
Discrete-time, second-order section, direct-form II transposed (`dfilt.df2tsos`)	dsp.BiquadFilter

Input Arguments

`hfilt`

Discrete-time filter (`dfilt`) object. The preceding table lists supported filter structures.

If `hfilt` is a discrete-time filter with the `PersistentMemory` property set to `true`, then the filter states are copied into the initial conditions properties of `hs`. Otherwise, initial conditions are ignored.

The function does not support some properties for SOS filter structures:

- If the `CastBeforeSum` property is set to `false`, the function issues a warning. `dsp.BiquadFilter` System objects always have a cast before a sum.
- If the `Signed` property is `false`, the function issues an error. `dsp.BiquadFilter` System objects do not support unsigned arithmetic.

Output Arguments

`hs`

Filter System object. The function maps almost all properties of `hfilt` into the filter System object. However, some properties are not mapped exactly:

- Filter System objects do not have a `CoeffAutoScale` property. The function specifies a word length and a fraction length regardless of whether the `CoeffAutoScale` property of `hfilt` is `true` or `false`.
- `dsp.BiquadFilter` System objects do not have a `FullPrecisionOverride` property. Full-precision values in `hfilt` are mapped to word and fraction lengths in `hs`. These settings correspond to the full-precision setting of the input data type.

Examples

Convert a discrete-time filter object to a System object

```
hfilt = dfilt.df1sos; %Direct-form I SOS
hs = sysobj(hfilt)   %Biquadratic IIR filter
```

```
hs =

  dsp.BiquadFilter with properties:

                     Structure: 'Direct form I'
               SOSMatrixSource: 'Property'
                     SOSMatrix: [1 0 0 1 0 0]
                   ScaleValues: [2×1 double]
    NumeratorInitialConditions: 0
  DenominatorInitialConditions: 0
      OptimizeUnityScaleValues: true

  Use get to show all properties
```

See Also

See Also

`block`

Introduced in R2012a

tf2ca

Transfer function to coupled allpass

Syntax

```
[d1,d2] = tf2ca(b,a)
[d1,d2] = tf2ca(b,a)
[d1,d2,beta] = tf2ca(b,a)
```

Description

`[d1,d2] = tf2ca(b,a)` where `b` is a real, symmetric vector of numerator coefficients and `a` is a real vector of denominator coefficients, corresponding to a stable digital filter, returns real vectors `d1` and `d2` containing the denominator coefficients of the allpass filters *H1(z)* and *H2(z)* such that

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[H1(z) + H2(z)]$$

representing a coupled allpass decomposition.

`[d1,d2] = tf2ca(b,a)` where `b` is a real, antisymmetric vector of numerator coefficients and `a` is a real vector of denominator coefficients, corresponding to a stable digital filter, returns real vectors `d1` and `d2` containing the denominator coefficients of the allpass filters *H1(z)* and *H2(z)* such that

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[H1(z) - H2(z)]$$

In some cases, the decomposition is not possible with real *H1(z)* and *H2(z)*. In those cases a generalized coupled allpass decomposition may be possible, as described in the following syntax.

`[d1,d2,beta] = tf2ca(b,a)` returns complex vectors `d1` and `d2` containing the denominator coefficients of the allpass filters *H1(z)* and *H2(z)*, and a complex scalar `beta`, satisfying `|beta|` = 1, such that

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)\left[\bar{\beta} \cdot H1(z) + \beta \cdot H2(z)\right]$$

representing the generalized allpass decomposition.

In the above equations, *H1(z)* and *H2(z)* are real or complex allpass IIR filters given by

$$H1(z) = \frac{fliplr(\overline{(D1(z))})}{D1(z)}, H2(1)(z) = \frac{fliplr(\overline{(D2(1)(z))})}{D2(1)(z)}$$

where *D1(z)* and *D2(z)* are polynomials whose coefficients are given by `d1` and `d2`.

Note A coupled allpass decomposition is not always possible. Nevertheless, Butterworth, Chebyshev, and Elliptic IIR filters, among others, can be factored in this manner. For details, refer to Signal Processing Toolbox User's Guide.

Examples

```
[b,a]=cheby1(9,.5,.4);
[d1,d2]=tf2ca(b,a);  % TF2CA returns denominators of the allpass.
num = 0.5*conv(fliplr(d1),d2)+0.5*conv(fliplr(d2),d1);
den = conv(d1,d2);  % Reconstruct numerator and denonimator.
MaxDiff=max([max(b-num),max(a-den)]); % Compare original and reconstructed
       % numerator and denominators.
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

`ca2tf` | `cl2tf` | `iirpowcomp` | `latc2tf` | `tf2latc`

Introduced in R2011a

tf2cl

Transfer function to coupled allpass lattice

Syntax

```
[k1,k2] = tf2cl(b,a)
[k1,k2] = tf2cl(b,a)
[k1,k2,beta] = tf2cl(b,a)
```

Description

`[k1,k2] = tf2cl(b,a)` where `b` is a real, symmetric vector of numerator coefficients and `a` is a real vector of denominator coefficients, corresponding to a stable digital filter, will perform the coupled allpass decomposition

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[H1(z) + H2(z)]$$

of a stable IIR filter *H(z)* and convert the allpass transfer functions *H1(z)* and *H2(z)* to a coupled lattice allpass structure with coefficients given in vectors `k1` and `k2`.

`[k1,k2] = tf2cl(b,a)` where `b` is a real, antisymmetric vector of numerator coefficients and `a` is a real vector of denominator coefficients, corresponding to a stable digital filter, performs the coupled allpass decomposition

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[H1(z) - H2(z)]$$

of a stable IIR filter *H(z)* and converts the allpass transfer functions *H1(z)* and *H2(z)* to a coupled lattice allpass structure with coefficients given in vectors `k1` and `k2`.

In some cases, the decomposition is not possible with real *H1(z)* and *H2(z)*. In those cases, a generalized coupled allpass decomposition may be possible, using the syntax described below.

`[k1,k2,beta] = tf2cl(b,a)` performs the generalized allpass decomposition of a stable IIR filter *H(z)* and converts the complex allpass transfer functions *H1(z)* and *H2(z)* to corresponding lattice allpass filters

$$H(z)=\frac{B(z)}{A(z)}=\left(\frac{1}{2}\right)\left[\bar{\beta}\cdot H1(z)+\beta\cdot H2(z)\right]$$

where `beta` is a complex scalar of magnitude equal to 1.

Note Coupled allpass decomposition is not always possible. Nevertheless, Butterworth, Chebyshev, and Elliptic IIR filters, among others, can be factored in this manner. For details, refer to Signal Processing Toolbox User's Guide.

Examples

```
[b,a]=cheby1(9,.5,.4);
[k1,k2]=tf2cl(b,a); % Get the reflection coeffs. for the lattices.
[num1,den1]=latc2tf(k1,'allpass'); % Convert each allpass lattice
[num2,den2]=latc2tf(k2,'allpass'); % back to transfer function.
num = 0.5*conv(num1,den2)+0.5*conv(num2,den1);
den = conv(den1,den2); % Reconstruct numerator and denonimator.
MaxDiff=max([max(b-num),max(a-den)]); % Compare original and reconstructed
       % numerator and denominators.
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constant. Expressions or variables are allowed if their values do not change.

See Also

`ca2tf` | `cl2tf` | `iirpowcomp` | `tf2ca` | `latc2tf` | `tf2latc`

Introduced in R2011a

validstructures

Structures for specification object with design method

Syntax

```
filtstruct = validstructures(D,'Systemobject',true)
filtstruct = validstructures(D,METHOD,'Systemobject',true)
```

Description

`filtstruct = validstructures(D,'Systemobject',true)` returns a structure of cell arrays, `filtstruct`, which contains a set of valid filter structures for the filter specification object, `D`. When you set `'Systemobject'` to `true`, `validstructures` returns a list of structures that support filter System objects. Each field in `filtstruct` lists a set of filter structures for the design method specified.

`filtstruct = validstructures(D,METHOD,'Systemobject',true)` returns the valid structures for the filter specification object, `D`, and the design method, `METHOD`, in a cell array of character vectors.

Examples

Valid Filter Structures

Design a default lowpass filter specification object. Return all valid design methods and structures in a structure array. Display the fieldnames to see all valid design methods. Display the valid filter structures for the `equiripple` field.

```
D = fdesign.lowpass;
filtstruct = validstructures(D,'SystemObject',true);

fn = fieldnames(filtstruct)
strs = eval(['filtstruct.' fn{5}])
```

```
fn =

  8×1 cell array
```

```
    'butter'
    'cheby1'
    'cheby2'
    'ellip'
    'equiripple'
    'ifir'
    'multistage'
    'kaiserwin'


strs =

  1×3 cell array

    'dffir'    'dffirt'    'dfsymfir'
```

Create a highpass filter of order 50 with a 3-dB frequency of 0.2. Obtain the available structures for a Butterworth design.

```
D = fdesign.highpass('N,F3dB',50,0.2);
C = validstructures(D,'butter','SystemObject',true)
```

```
C =

  1×6 cell array

  Columns 1 through 5

    'df1sos'    'df2sos'    'df1tsos'    'df2tsos'    'cascadeallpass'

  Column 6

    'cascadewdfallpass'
```

See Also

See Also

design | designmethods | designopts | fdesign

Introduced in R2009a

wdf2allpass

Wave Digital Filter to allpass coefficient transformation

Syntax

```
a = wdf2allpass(w)
A = wdf2allpass(W)
```

Description

`a = wdf2allpass(w)` accepts a vector of transformed real allpass coefficients, `w`, and returns the conventional allpass polynomial version `a`. `w` is used by allpass filter objects such as `dsp.AllpassFilter`, and `dsp.CoupledAllpassFilter`, with `Structure` set to `'Wave Digital Filter'`.

`A = wdf2allpass(W)` accepts the cell array of transformed allpass coefficient vectors `W`. Each cell of `W` contains the transformed real coefficients of a section of a cascade allpass filter. The output `A` is also a cell array, and each cell of `A` contains the conventional polynomial version of the corresponding cell of `W`. `W` is used by allpass filter objects such as dsp.AllpassFilter and dsp.CoupledAllpassFilter, with `Structure` set to `'Wave Digital Filter'`. Every cell of `W` must contain a real vector of length 1,2, or 4. When the length is 4, the second and fourth components must both be zero. `W` can be a row or column vector of cells while `A` is always returned as column.

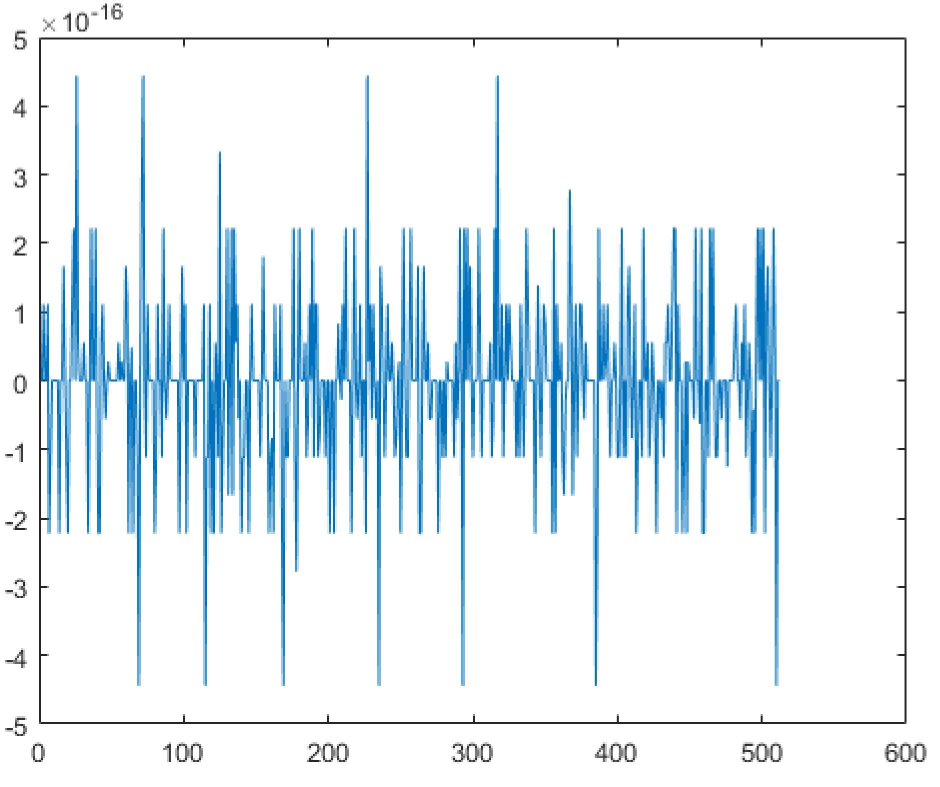
Examples

Convert Wave Digital Filter Coefficients to Allpass Polynomial Coefficients

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create a second order allpass filter with wave digital filter coefficients w = [0.5 0]. Convert these coefficients into polynomial form using `wdf2allpass`. Assign the polynomial coefficients to an allpass filter using the 'Minimum multiplier' structure. Pass a random input to both these filters and compare the outputs.

```
w = [0.5 0];
allpasswdf = dsp.AllpassFilter('Structure', 'Wave Digital Filter',...
    'WDFCoefficients', w);
a = wdf2allpass(w);
allpass = dsp.AllpassFilter('AllpassCoefficients', a);
in = randn(512, 1);
outputallpasswdf = allpasswdf(in);
outputallpass = allpass(in);
plot(outputallpasswdf-outputallpass);
```



The difference between the two outputs is very small.

Input Arguments

w — Transformed Wave Digital Filter allpass coefficients

(default) | vector of real numbers

Numeric vector of transformed Wave Digital Filter allpass coefficients, specified as a real number. w can have only length equal to 1,2, and 4. When the length is 4, the second and fourth components must both be zero. w can be a row or a column vector.

Example: `[0.3,-0.2]`

Data Types: `double` | `single`

W — Transformed Wave Digital Filter allpass coefficients

(default) | vector of cells

Cascade of allpass filter coefficients in transformed Wave Digital Filter form, specified as a cell vector. Every cell of W must contain a real vector of length 1, 2, or 4. When the length is 4, the second and fourth components must both be zero. W can be a row or a column vector of cells.

Example: `{[0.3,-0.2];0.5}`

Data Types: `double` | `single`

Output Arguments

a — allpass filter coefficients

(default) | vector of real numbers

Numeric vector of polynomial allpass coefficients, determined as a numeric row vector.

Example: `3.1`

Data Types: `double` | `single`

A — allpass filter coefficients

(default) | vector cell array

Cascade of allpass filter coefficient, determined as a column of cells, each containing a vector of length 1, 2, or 4.

Example: `{0.3 5.0 0.2}`

Data Types: `double` | `single`

Algorithms

`wdf2allpass` provides the inverse operation of `allpass2wdf`, by transforming the transformed cascade of allpass coefficients `W` into their conventional polynomial representation `A`. Please refer to the reference page for `allpass2wdf` for more details about the two representations.

`W` defines a multisection allpass filter, and `wdf2allpass` applies separately to each section, with the same transformation used in the single-section case. In this case, the numeric coefficients vector `W` can have order 1, 2, or 4.

The relations between the vector of section coefficients a and w respectively depend on the order, as follows:

$$
\begin{aligned}
&\textit{for order } 1: \\
&a_1 = w_1 \\
&\textit{for order } 2: \\
&a_1 = w_2(1 + w_1) \\
&a_2 = w_1 \\
&\textit{for order } 4: \\
&a_2 = w_3(1 + w_1) \\
&a_4 = w_1 \\
&a_1 = a_3 = 0
\end{aligned}
$$

References

[1] M. Lutovac, D. Tosic, B. Evans, *Filter Design for Signal Processing using MATLAB and Mathematica*. Prentice Hall, 2001.

See Also

See Also

allpass2wdf | ca2tf | dsp.AllpassFilter | dsp.CoupledAllpassFilter | latc2tf

Introduced in R2014a

window

FIR filter using windowed impulse response

Syntax

```
h = window(d,fcnhndl,fcnarg,'SystemObject',true)
h = window(d,win,'SystemObject',true)
```

Description

`h = window(d,fcnhndl,fcnarg,'SystemObject',true)` designs a single-rate digital filter System object using the specifications in filter specification object `d`.

`fcnhndl` is a handle to a filter design function that returns a window vector, such as the `hamming` or `blackman` functions. `fcnarg` is an optional argument that returns a window. You pass the function to `window`.

`h = window(d,win,'SystemObject',true)` designs a filter using the vector you supply in `win`. The length of vector `win` must be the same as the impulse response of the filter, which is equal to the filter order plus one.

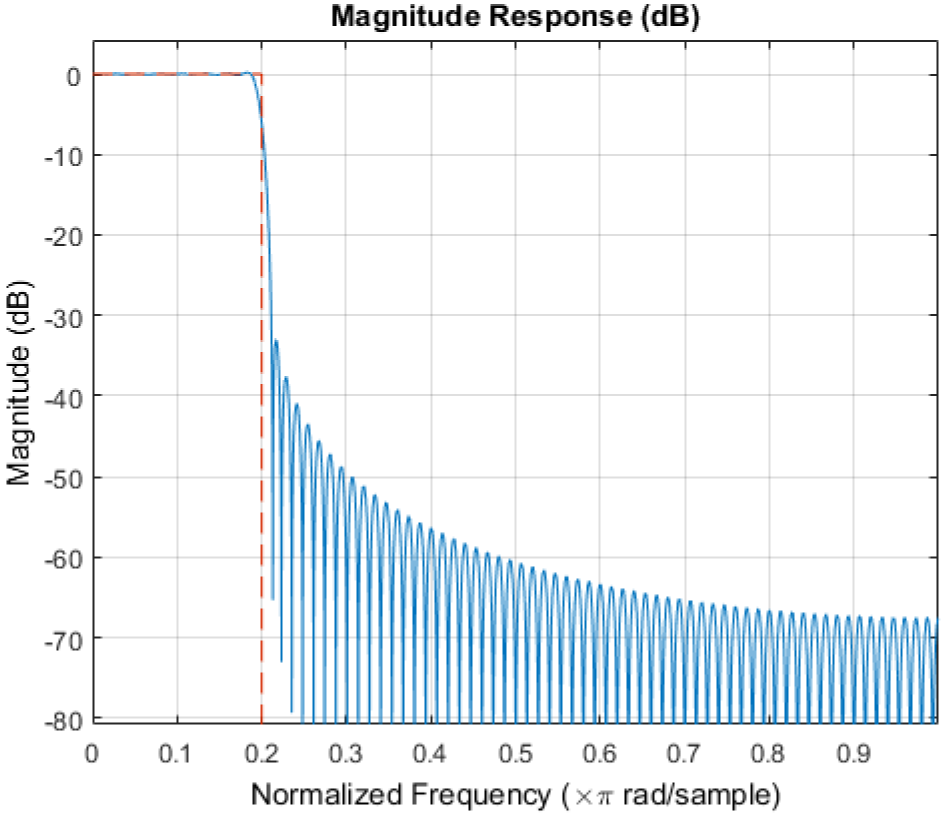
Examples

Design a Nyquist Filter Using Kaiser Window

This example designs a filter using the two design techniques of specifying a function handle and passing a window vector as an input argument.

Use a window vector provided by the kaiser window function to design a Nyquist filter. The window length must be the filter order plus one.

```
d = fdesign.nyquist(5,'n',150);
% Kaiser window with beta parameter 2.5
nyqFilter = window(d,'window',kaiser(151,2.5),'SystemObject',true);
fvtool(nyqFilter)
```



See Also

`firls` | `kaiserwin`

Introduced in R2011a

zerophase

Zero-phase response of discrete-time filter System object

Syntax

```
[zphase,w] = zerophase(sysobj)
[zphase,w] = zerophase(sysobj,n)
[zphase,w] = zerophase(sysobj,Name,Value)
zerophase(sysobj)
```

Description

`[zphase,w] = zerophase(sysobj)` returns the instantaneous zero-phase response `zphase` of the filter System object, `sysobj`, based on the current filter coefficients. The vector `w` contains the frequencies (in radians) at which the function evaluates the zero-phase response.

The zero-phase response is evaluated at 8192 points equally spaced around the upper half of the unit circle.

`[zphase,w] = zerophase(sysobj,n)` returns the instantaneous zero-phase response of the filter System object and the corresponding frequencies at `n` points equally spaced around the upper half of the unit circle.

`[zphase,w] = zerophase(sysobj,Name,Value)` returns the zero-phase response with additional options specified by one or more `Name,Value` pair arguments.

`zerophase(sysobj)` displays the zero-phase response of the filter System object `sysobj` in the `fvtool`.

For information on more input options, refer to `zerophase` in Signal Processing Toolbox documentation.

Input Arguments

sysobj

Filter System object.

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpassFilter
dsp.AllpoleFilter
dsp.BiquadFilter
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.CICDecimator
dsp.CICInterpolator
dsp.CoupledAllpassFilter
dsp.FarrowRateConverter
dsp.FilterCascade
dsp.FIRDecimator
dsp.FIRFilter
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.FIRInterpolator
dsp.FIRRateConverter
dsp.HighpassFilter
dsp.IIRFilter
dsp.IIRHalfbandDecimator
dsp.IIRHalfbandInterpolator
dsp.LowpassFilter
dsp.NotchPeakFilter

Filter System objects
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

n

Number of samples. For an FIR filter where `n` is a power of two, the computation is done faster using FFTs.

Default: 8192

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Arithmetic'` — Value type:
`‘double’ | 'single' | 'fixed'`

Specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based

System Object State	Coefficient Data Type	Rule
		on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

`zphase`

Zero phase response vector of length `n`. If `n` is not specified, the function uses a default value of `8192`.

`w`

Frequency vector of length `n`, in radians/sample. `w` consists of `n` points equally spaced around the upper half of the unit circle (from 0 to π radians/sample). If `n` is not specified, the function uses a default value of `8192`.

See Also

See Also

Functions
`freqz` | `fvtool` | `grpdelay` | `impz` | `phasez` | `zerophase` | `zplane`

Introduced in R2011a

zpkbpc2bpc

Zero-pole-gain complex bandpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkbpc2bpc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkbpc2bpc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the complex bandpass prototype by applying a first-order complex bandpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The original lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places two features of an original filter, located at frequencies W_{o1} and W_{o2}, at the required target frequency locations, W_{t1}, and W_{t2} respectively. It is assumed that W_{t2} is greater than W_{t1}. In most of the cases the features selected for the transformation are the band edges of the filter passbands. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

This transformation can also be used for transforming other types of filters; e.g., complex notch filters or resonators can be repositioned at two distinct desired frequencies at any place around the unit circle; e.g., in the adaptive system.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
```

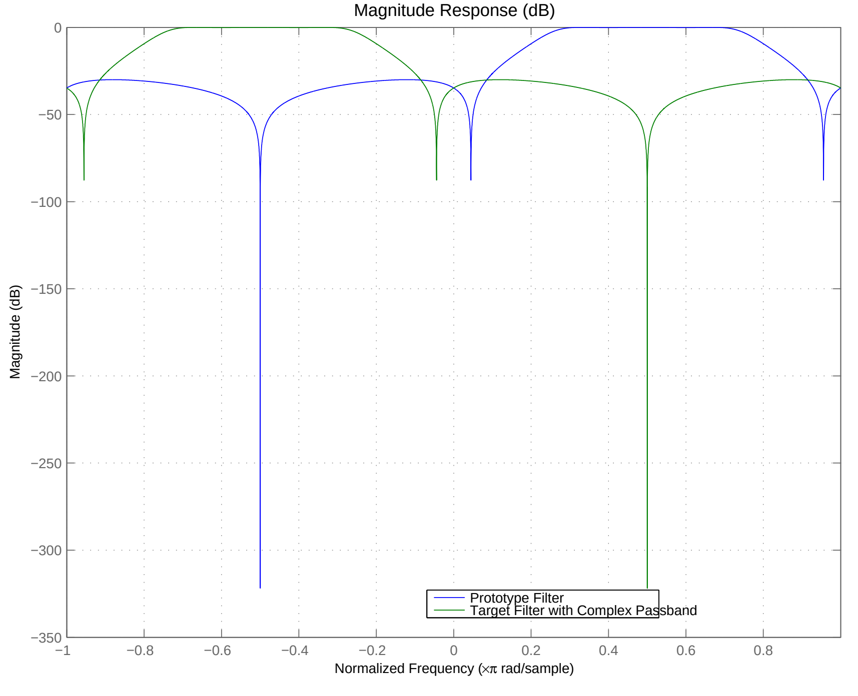
Create a complex passband from 0.25 to 0.75:

```
[b, a] = iirlp2bpc(b,a,0.5,[0.25,0.75]);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpkbpc2bpc(z,p,k,[0.25, 0.75],[-0.75, -0.25]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Comparing the filters in FVTool shows the example results. Use the features in FVTool to check the filter coefficients, or other filter analyses.



Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkftransf` | `allpassbpc2bpc` | `iirbpc2bpc`

Introduced in R2011a

zpkftransf

Zero-pole-gain frequency transformation

Syntax

```
[Z2,P2,K2] = zpkftransf(Z,P,K,AllpassNum,AllpassDen)
```

Description

`[Z2,P2,K2] = zpkftransf(Z,P,K,AllpassNum,AllpassDen)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the transformed lowpass digital filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`. If `AllpassDen` is not specified it will default to 1. If neither `AllpassNum` nor `AllpassDen` is specified, then the function returns the input filter.

Examples

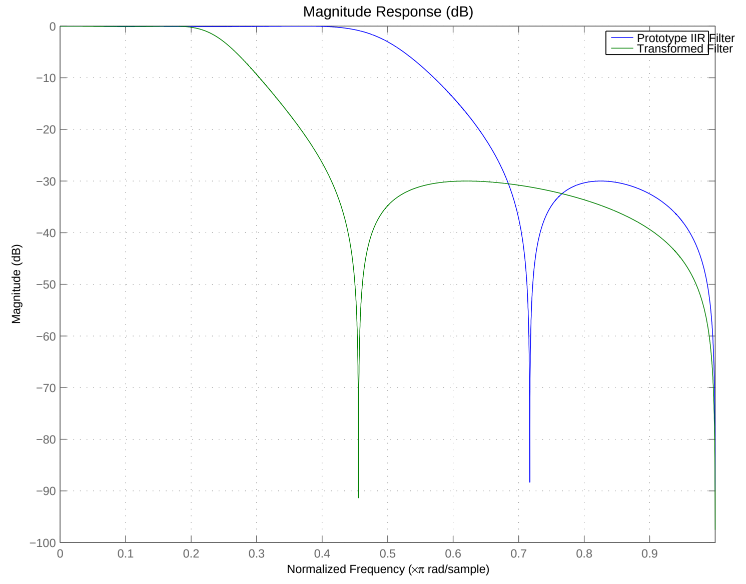
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
[AlpNum, AlpDen] = allpasslp2lp(0.5, 0.25);
[z2, p2, k2] = zpkftransf(roots(b),roots(a),b(1),AlpNum,AlpDen);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

After transforming the filter, you get the response shown in the figure, where the passband has been shifted towards zero.



Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
FTFNum	Numerator of the mapping filter
FTFDen	Denominator of the mapping filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter

See Also

`iirftransf`

Introduced in R2011a

zpklp2bp

Zero-pole-gain lowpass to bandpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bp(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bp(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandpass frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "DC Mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

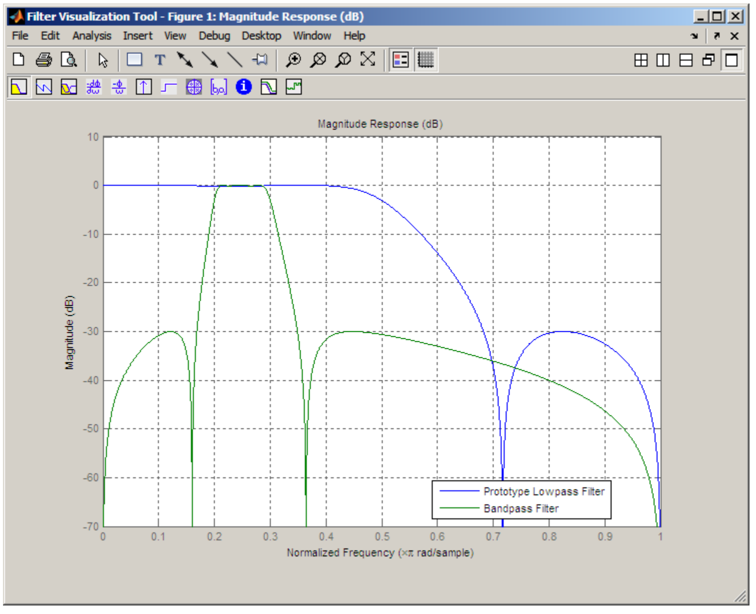
Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Real lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be easily doubled and positioned at two distinct, desired frequencies.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[B,A] = ellip(3,0.1,30,0.409);
Z = roots(B);
P = roots(A);
K = B(1);
[Z2,P2,K2] = zpklp2bp(Z,P,K, 0.5, [0.2 0.3]);
hfvt = fvtool(B,A,K2*poly(Z2),poly(P2));
legend(hfvt,'Prototype Lowpass Filter', 'Bandpass Filter');
axis([0 1 -70 10]);
```



Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter

Variable	Description
`K`	Gain factor of the prototype lowpass filter
`Wo`	Frequency value to be transformed from the prototype filter
`Wt`	Desired frequency location in the transformed target filter
`Z2`	Zeros of the target filter
`P2`	Poles of the target filter
`K2`	Gain factor of the target filter
`AllpassNum`	Numerator of the mapping filter
`AllpassDen`	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

Constantinides, A.G., "Spectral transformations for digital filters," *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

See Also

`zpkftransf` | `iirlp2bp` | `allpasslp2bp`

Introduced in R2011a

zpklp2bpc

Zero-pole-gain lowpass to complex bandpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bpc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bpc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandpass filters for radio receivers from the high-quality prototype lowpass filter.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2bpc(z, p, k, 0.5, [0.2 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

zpkftransf | iirlp2bpc | allpasslp2bpc

Introduced in R2011a

zpklp2bs

Zero-pole-gain lowpass to bandstop frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bs(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bs(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandstop frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "Nyquist Mobility," which means that the DC feature stays at DC, but the Nyquist feature moves to a location dependent on the selection of W_o and W_ts.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2bs(z, p, k, 0.5, [0.2 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

Constantinides, A.G., “Spectral transformations for digital filters,” *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, “Prototype reference transfer function parameters in the discrete-time frequency transformations,” *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, “Closed-form solutions for discrete-time elliptic transfer functions,” *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., “Design of bandpass digital filters,” *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

See Also

`zpkftransf` | `iirlp2bs` | `allpasslp2bs`

Introduced in R2011a

zpklp2bsc

Zero-pole-gain lowpass to complex bandstop frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bsc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bsc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandstop frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. Additionally the transformation swaps passbands with stopbands in the target filter.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/ resonators.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2bsc(z, p, k, 0.5, [0.2, 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

`zpkftransf` | `iirlp2bsc` | `allpasslp2bsc`

Introduced in R2011a

zpklp2hp

Zero-pole-gain lowpass to highpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2hp(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2hp(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to real highpass frequency mapping. This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency location, W_t, at the same time rotating the whole frequency response by half of the sampling frequency. Result is that the DC and Nyquist features swap places.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and the gain factor, `K`.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to highpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, or the deep minimum in the stopband, or other ones.

Lowpass to highpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without designing them again.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2hp(z, p, k, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

Constantinides, A.G., “Spectral transformations for digital filters,” *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

See Also

`zpkftransf` | `iirlp2hp` | `allpasslp2hp`

Introduced in R2011a

zpklp2lp

Zero-pole-gain lowpass to lowpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2lp(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2lp(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to real lowpass frequency mapping. This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency location, W_t.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to lowpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to lowpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without designing them again.

Examples

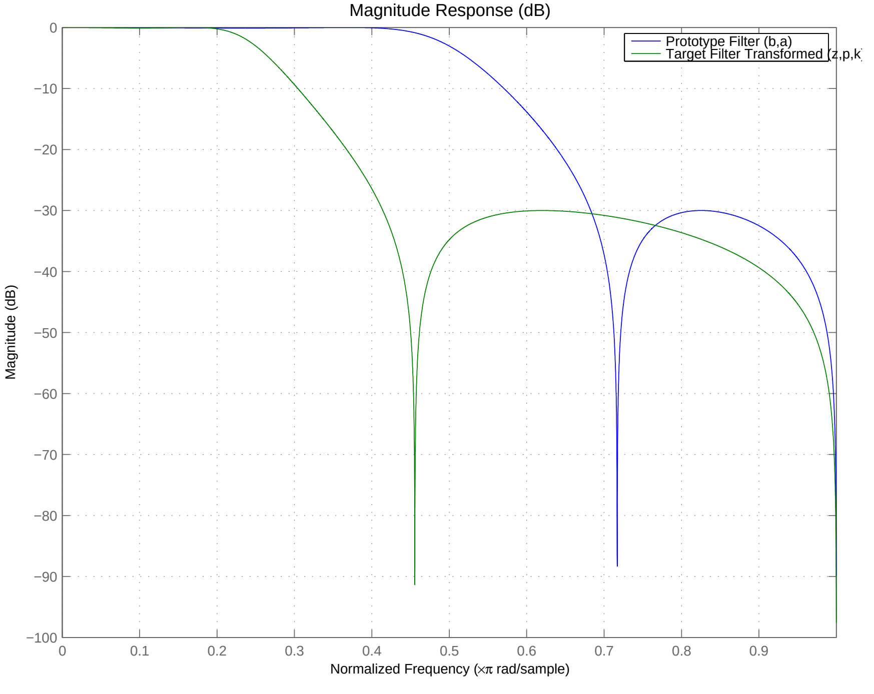
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2lp(z, p, k, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Using `zpklp2lp` creates the desired half band IIR filter with the transformed features that you specify in the transformation function. This figure shows the results.



Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter

Variable	Description
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

Constantinides, A.G., “Spectral transformations for digital filters,” *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, “Prototype reference transfer function parameters in the discrete-time frequency transformations,” *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, “Closed-form solutions for discrete-time elliptic transfer functions,” *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., “Frequency transformations for digital filters,” *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

See Also

`zpkftransf` | `iirlp2lp` | `allpasslp2lp`

Introduced in R2011a

zpklp2mb

Zero-pole-gain lowpass to M-band frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt)
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt,Pass)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying an Mth-order real lowpass to real multibandpass frequency mapping. By default the DC feature is kept at its original location.

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility". In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, $W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

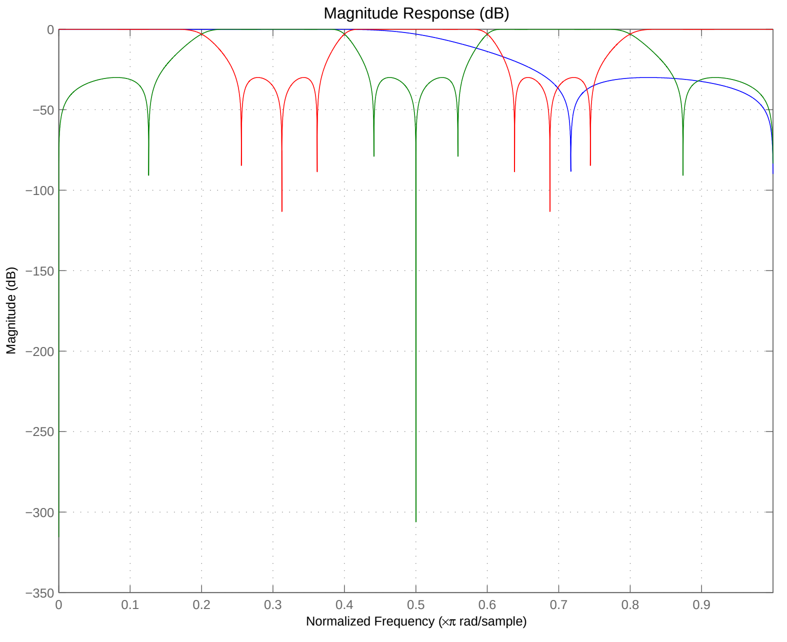
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z1,p1,k1] = zpklp2mb(z, p, k, 0.5, [2 4 6 8]/10, 'pass');
[z2,p2,k2] = zpklp2mb(z, p, k, 0.5, [2 4 6 8]/10, 'stop');
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k1*poly(z1), poly(p1), k2*poly(z2), poly(p2));
```

The resulting multiband filter that replicates features from the prototype appears in the figure shown. Note the accuracy of the replication process.



Arguments

Variable	Description
`Z`	Zeros of the prototype lowpass filter
`P`	Poles of the prototype lowpass filter
`K`	Gain factor of the prototype lowpass filter
`Wo`	Frequency value to be transformed from the prototype filter
`Wt`	Desired frequency location in the transformed target filter
`Pass`	Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `'pass'` being the default
`Z2`	Zeros of the target filter
`P2`	Poles of the target filter
`K2`	Gain factor of the target filter

Variable	Description
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

Franchitti, J.C., “All-pass filter interpolation and frequency transformation problems,” *MSc Thesis*, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.

Feyh, G., J.C. Franchitti and C.T. Mullis, “All-pass filter interpolation and frequency transformation problem,” *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

Mullis, C.T. and R.A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Massachusetts, Addison-Wesley, 1987.

Feyh, G., W.B. Jones and C.T. Mullis, *An extension of the Schur Algorithm for frequency transformations, Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

See Also

`zpkftransf` | `iirlp2mb` | `allpasslp2mb`

Introduced in R2011a

zpklp2mbc

Zero-pole-gain lowpass to complex M-band frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklpmbc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklpmbc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying an Mth-order real lowpass to complex multibandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, W_{t1},...,W_{tM}.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature, for example, the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

This transformation can also be used for transforming other types of filters; e.g., to replicate notch filters and resonators at any required location.

Examples

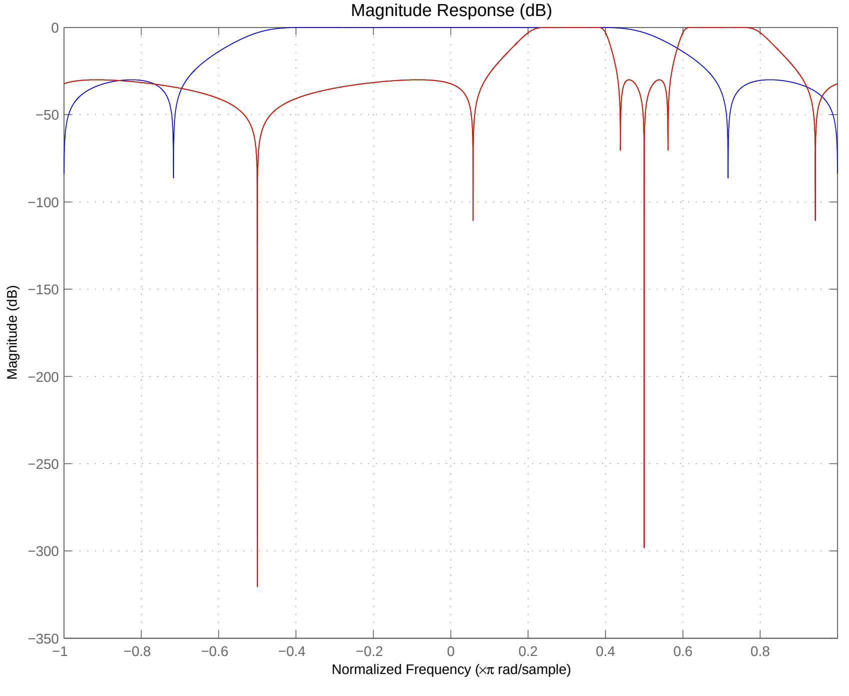
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z1,p1,k1] = zpklp2mbc(z, p, k, 0.5, [2 4 6 8]/10);
[z2,p2,k2] = zpklp2mbc(z, p, k, 0.5, [2 4 6 8]/10);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k1*poly(z1), poly(p1), k2*poly(z2), poly(p2));
```

You could review the coefficients to compare the filters, but the graphical comparison shown here is quicker and easier.



However, looking at the coefficients in FVTool shows the complex nature desired.

Arguments

Variable	Description
`Z`	Zeros of the prototype lowpass filter
`P`	Poles of the prototype lowpass filter
`K`	Gain factor of the prototype lowpass filter
`Wo`	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
`Wt`	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
`Z2`	Zeros of the target filter
`P2`	Poles of the target filter
`K2`	Gain factor of the target filter
`AllpassNum`	Numerator of the mapping filter
`AllpassDen`	Denominator of the mapping filter

See Also

`zpkftransf` | `iirlp2mbc` | `allpasslp2mbc`

Introduced in R2011a

zpklp2xc

Zero-pole-gain lowpass to complex N-point frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xc(Z,P,K,Wo,Wt)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying an Nth-order real lowpass to complex multipoint frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies W_{o1},...,W_{oN}, at the required target frequency locations, W_{t1},...,W_{tM}.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

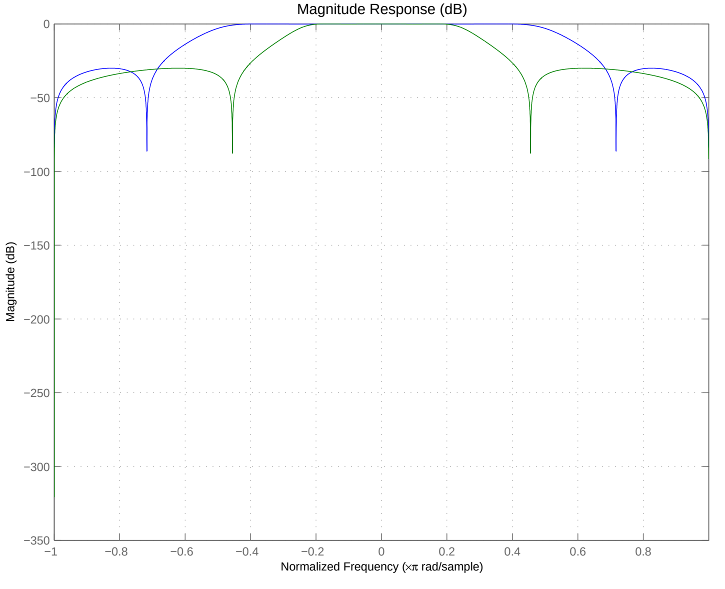
Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2xc(z, p, k, [-0.5 0.5], [-0.25 0.25]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Plotting the filters on the same axes lets you compare the results graphically, shown here.



Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency values to be transformed from the prototype filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also

`zpkftransf` | `iirlp2xc` | `allpasslp2xc`

Introduced in R2011a

zpklp2xn

Zero-pole-gain lowpass to N-point frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt)
[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt,Pass)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying an `N`th-order real lowpass to real multipoint frequency transformation, where `N` is the number of features being mapped. By default the DC feature is kept at its original location.

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility". In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

Parameter `N` also specifies the number of replicas of the prototype filter created around the unit circle after the transformation. This transformation effectively places `N` features of an original filter, located at frequencies $W_{o1},...,W_{oN}$, at the required target frequency locations, $W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter are the same in the target filter for the Nyquist mobility and are reversed for the DC mobility. For the Nyquist mobility this means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation. For DC mobility feature F_2 will precede F_1 after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2xn(z, p, k, [-0.5 0.5], [0 0.25], 'pass');
hfvt = fvtool(b, a, k2*poly(z2), poly(p2));
legend(hfvt,'Original Filter','Half-band Filter');
```

As demonstrated by the figure, the target filter has the desired response shape and values replicated from the prototype.

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Pass	Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `'pass'` being the default
Z2	Zeros of the target filter

Variable	Description
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassDen	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References

Cain, G.D., A. Krukowski and I. Kale, "High Order Transformations for Flexible IIR Filter Design," *VII European Signal Processing Conference (EUSIPCO'94)*, vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

Krukowski, A., G.D. Cain and I. Kale, "Custom designed high-order frequency transformations for IIR filters," *38th Midwest Symposium on Circuits and Systems (MWSCAS'95)*, Rio de Janeiro, Brazil, August 1995.

See Also

`zpkftransf` | `iirlp2xn` | `allpasslp2xn`

Introduced in R2011a

zpkrateup

Zero-pole-gain complex bandpass frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkrateup(Z,P,K,N)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkrateup(Z,P,K,N)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter being transformed from any prototype by applying an `N`th-order rateup frequency transformation, where `N` is the upsample ratio. Transformation creates `N` equal replicas of the prototype filter frequency response.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The original lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
% Upsample the prototype filter 4 times
[z2,p2,k2] = zpkrateup(z, p, k, 4);
% Compare prototype filter with target filter
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
N	Integer upsampling ratio
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also

`zpkrateup` | `allpassrateup` | `iirrateup`

Introduced in R2011a

zpkshift

Zero-pole-gain real shift frequency transformation

Syntax

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshift(Z,P,K,Wo,Wt)`

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshift(Z,P,K,Wo,Wt)` returns the zeros,`Z2` , poles, `P2`, and gain factor, `K2`, of the target filter transformed from the zeros, poles, and gain factor of real lowpass prototype by applying a second-order real shift frequency mapping. It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen` of the allpass mapping filter.

This transformation places one selected feature of an original filter, located at frequency W_o, at the required target frequency location, W_t. This transformation implements the "DC Mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

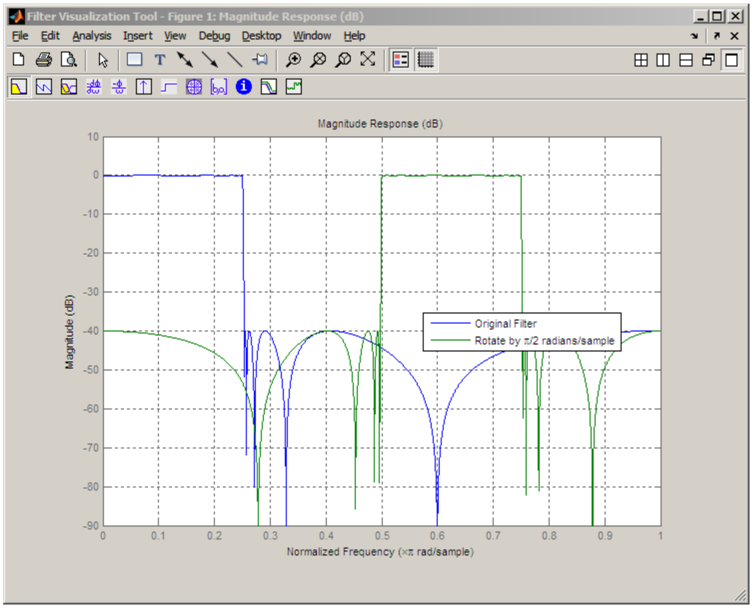
Choice of the feature subject to the real shift transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way without the need to design them again.

Examples

Rotate frequency response by π/2 radians/sample:

```
[B,A] = ellip(10,0.1,40,0.25);
% Elliptic lowpass filter with passband frequency 0.25*pi rad/sample
Z = roots(B); % get roots of numerator polynomial- filter zeros
P = roots(A); % get roots of denominator polynomial- filter poles
K = B(1);
[Z2,P2,K2] = zpkshift(Z,P,K,0.25,0.75); % shift by 0.25*pi rad/sample
Num = poly(Z2);
Den = poly(P2);
hfvt = fvtool(B,A,K2*Num,Den);
legend(hfvt,'Original Filter','Rotate by \pi/2 radians/sample');
axis([0 1 -90 10]);
```



See Also

zpkftransf | iirshift | allpassshift

Introduced in R2011a

zpkshiftc

Zero-pole-gain complex shift frequency transformation

Syntax

```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,Wo,Wt)
[Num,Den,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,0,0.5)
[Num,Den,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,0,-0.5)
```

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a first-order complex frequency shift transformation. This transformation rotates all the features of an original filter by the same amount specified by the location of the selected feature of the prototype filter, originally at W_o, placed at W_t in the target filter.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and the gain factor, `K`.

`[Num,Den,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,0,0.5)` performs the Hilbert transformation, i.e. a 90 degree counterclockwise rotation of an original filter in the frequency domain.

`[Num,Den,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,0,-0.5)` performs the inverse Hilbert transformation, i.e. a 90 degree clockwise rotation of an original filter in the frequency domain.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
```

Rotation by $\pi/4$ Radians/Sample

Rotation by -0.25:

```
[z2,p2,k2] = zpkshiftc(z, p, k, 0.5, 0.25);
hfvt = fvtool(b,a,k2*poly(z2),poly(p2));
```

Rotation by $\pi/2$ Radians/Sample

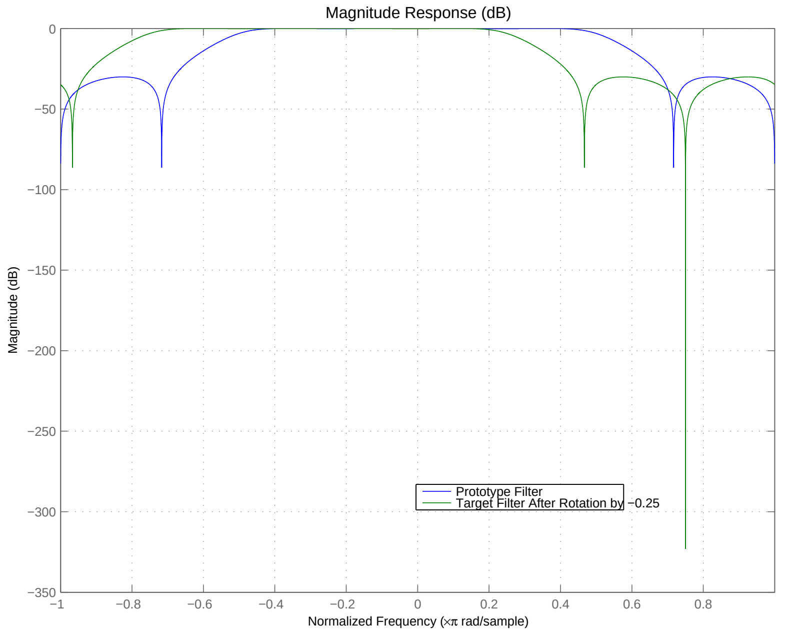
```
[z3,p3,k3] = zpkshiftc(z, p, k, 0, 0.5);
addfilter(hfvt,k3*poly(z3),poly(p3));
legend(hfvt,'Original Filter','Rotation by -\pi/4 radians/sample',...
    'Rotation by \pi/2 radians/sample');
```

Rotation by $-\pi/2$ Radians/Sample

```
[z2,p2,k2] = zpkshiftc(z, p, k, 0.5, -0.5);
fvtool(b, a, k2*poly(z2), poly(p2));
```

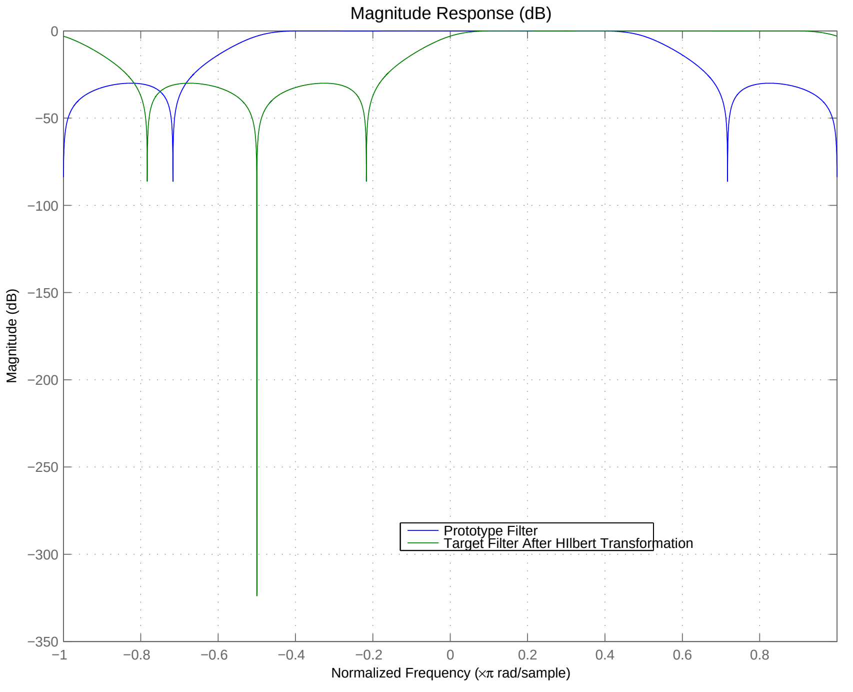
Result of Example 1

After performing the rotation, the resulting filter shows the features desired.



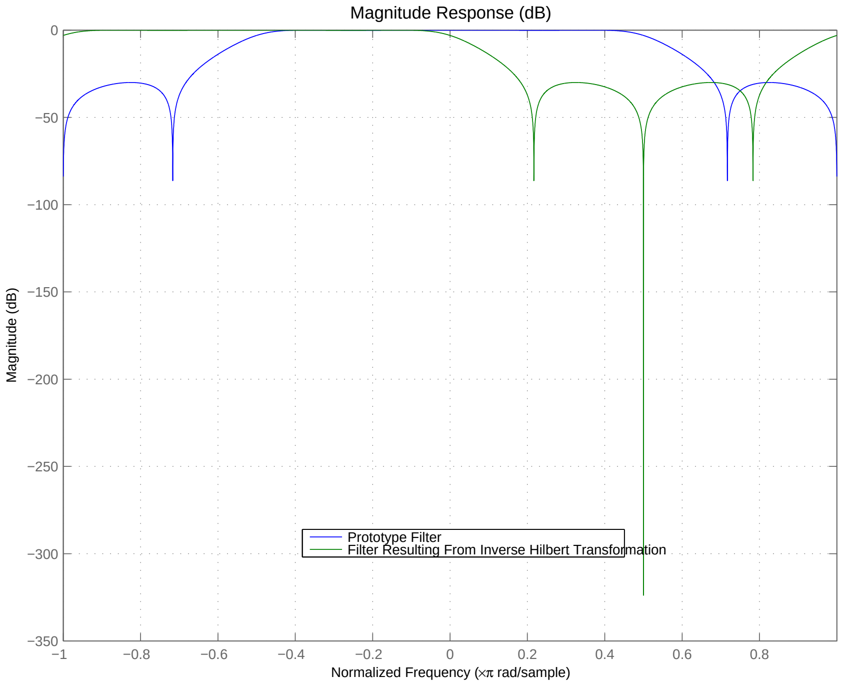
Result of Example 2

Similar to the first example, performing the Hilbert transformation generates the desired target filter, shown here.



Result of Example 3

Finally, using the inverse Hilbert transformation creates yet a third filter, as the figure shows.



Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassDen	Numerator of the mapping filter

Variable	Description
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

References

Oppenheim, A.V., R.W. Schafer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

Dutta-Roy, S.C. and B. Kumar, "On digital differentiators, Hilbert transformers, and half-band low-pass filters," *IEEE Transactions on Education*, vol. 32, pp. 314-318, August 1989.

See Also

`zpkftransf` | `iirshiftc` | `allpassshiftc`

Introduced in R2011a

zplane

Z-plane zero-pole plot for discrete-time filter System object

Syntax

```
zplane(filt)
zplane(filt,Name,Value)
[zLoc,pLoc,tLoc] = zplane(filt)
```

Description

`zplane(filt)` plots the zeros and poles of the filter System object, `filt`, with the unit circle for reference in the filter visualization tool (fvtool). Each zero is represented with a `'o'` and each pole with a `'x'` on the plot. Multiple zeros and poles are indicated by the multiplicity number shown to the upper right of the zero or pole.

When you call the `step` method on the filter System object with a fixed-point input, the filter becomes a quantized fixed-point filter, `filtQuant`. When `filtQuant` is a quantized filter, `zplane(filtQuant)` plots the poles and zeros of the quantized and unquantized filters. The symbols □ and + represent the zeros and poles of the quantized filter `filtQuant`. The plot includes the unit circle for reference.

`zplane(filt,Name,Value)` returns the pole zero plot with additional options specified by one or more `Name,Value` pair arguments.

`[zLoc,pLoc,tLoc] = zplane(filt)` returns the vector of locations for zeros, poles, and text objects. `zLoc` is the vector of locations of zeros, `pLoc` is the vector of locations of poles, and `tLoc` is the vector of locations of the axes/unit circle line and text objects which are present when there are multiple zeros or poles. In case there are no zeros or poles, `zLoc` or `pLoc` is set to the empty matrix `[]`.

Input Arguments

`filt`

A filter System object.

The following Filter System objects are supported by this analysis function:

Filter System objects
dsp.AllpassFilter
dsp.AllpoleFilter
dsp.BiquadFilter
dsp.CICCompensationDecimator
dsp.CICCompensationInterpolator
dsp.CICDecimator
dsp.CICInterpolator
dsp.CoupledAllpassFilter
dsp.FarrowRateConverter
dsp.FilterCascade
dsp.FIRDecimator
dsp.FIRFilter
dsp.FIRHalfbandDecimator
dsp.FIRHalfbandInterpolator
dsp.FIRInterpolator
dsp.FIRRateConverter
dsp.HighpassFilter
dsp.IIRFilter
dsp.IIRHalfbandDecimator
dsp.IIRHalfbandInterpolator
dsp.LowpassFilter
dsp.NotchPeakFilter
dsp.VariableBandwidthFIRFilter
dsp.VariableBandwidthIIRFilter

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Arithmetic'` — Value types:
`‘double’ | 'single' | 'fixed'`

When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based

System Object State	Coefficient Data Type	Rule
		on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

`zLoc`

Vector of locations of zeros.

`pLoc`

Vector of locations of poles.

`tLoc`

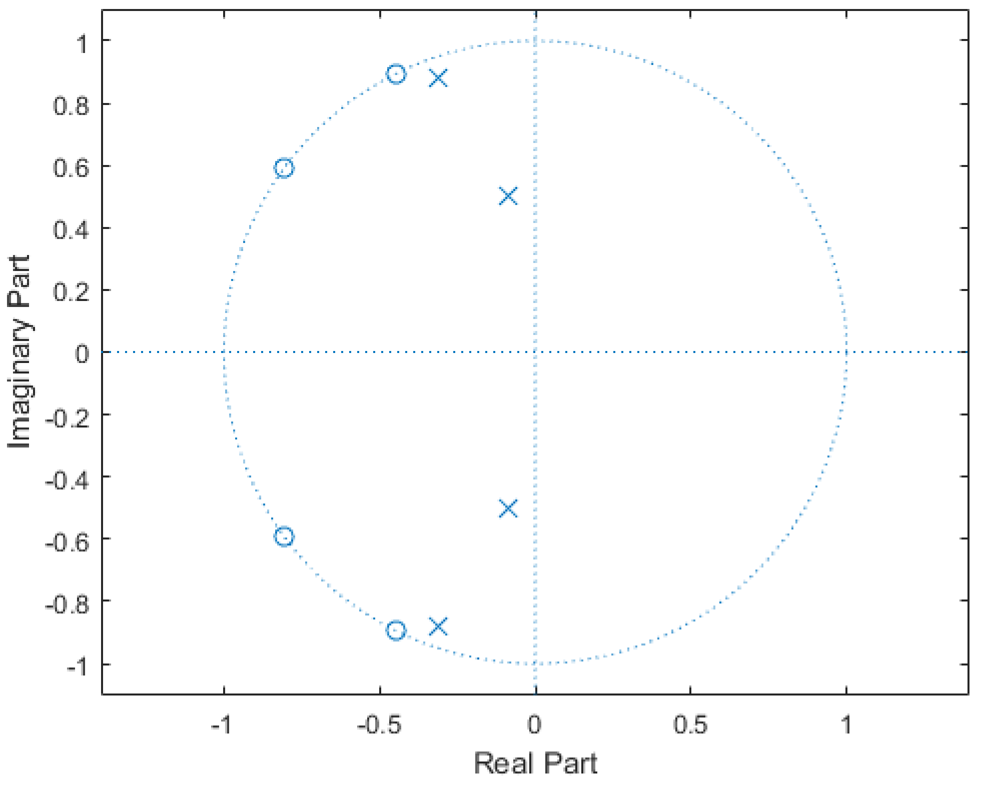
Vector of locations of axes/unit circle line and text objects.

Examples

Plot the Poles and Zeros of a Fourth-Order Filter

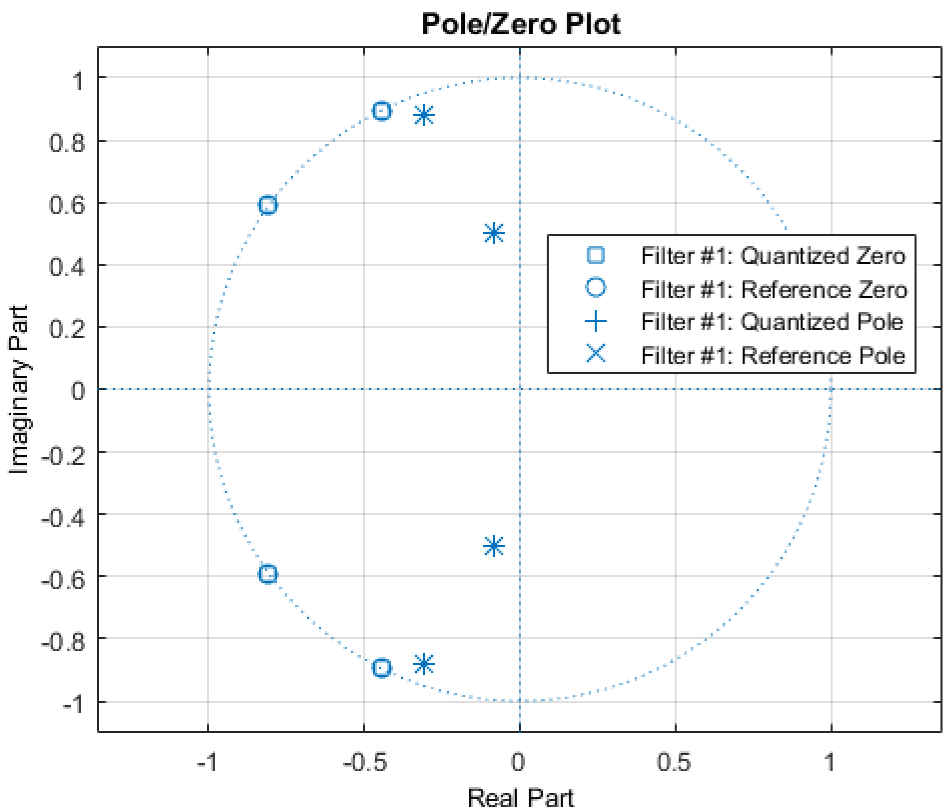
Create a Fourth-order IIR digital filter with a cutoff frequency of 0.6. Plot the poles and zeros of this filter.

```
[b,a] = ellip(4,.5,20,.6);
zplane(b,a);
```



Quantize the filter by passing a fixed-point input through the filter algorithm. Plot the quantized and unquantized poles and zeros associated with this filter.

```
iirFilt = dsp.IIRFilter('Numerator',b,'Denominator',a);
in = fi(randn(15,6),1,15,3);
out = iirFilt(in);
zplane(iirFilt);
```



See Also

See Also

Functions
`freqz` | `impz`

Introduced in R2011a

Reference for the Properties of Filter Objects

- "Fixed-Point Filter Properties" on page 6-2
- "Adaptive Filter Properties" on page 6-92
- "Multirate Filter Properties" on page 6-104

Fixed-Point Filter Properties

In this section...

"Overview of Fixed-Point Filters" on page 6-2

"Fixed-Point Objects and Filters" on page 6-2

"Summary — Fixed-Point Filter Properties" on page 6-4

"Property Details for Fixed-Point Filters" on page 6-17

Overview of Fixed-Point Filters

There is a distinction between fixed-point filters and quantized filters — quantized filters represent a superset that includes fixed-point filters.

When `dfilt` objects have their `Arithmetic` property set to `single` or `fixed`, they are quantized filters. However, after you set the `Arithmetic` property to `fixed`, the resulting filter is both quantized and fixed-point. Fixed-point filters perform arithmetic operations without allowing the binary point to move in response to the calculation — hence the name fixed-point. You can find out more about fixed-point arithmetic in your Fixed-Point Designer documentation or from the Help system.

With the `Arithmetic` property set to `single`, meaning the filter uses single-precision floating-point arithmetic, the filter allows the binary point to move during mathematical operations, such as sums or products. Therefore these filters cannot be considered fixed-point filters. But they are quantized filters.

The following sections present the properties for fixed-point filters, which includes all the properties for double-precision and single-precision floating-point filters as well.

Fixed-Point Objects and Filters

Fixed-point filters depend in part on fixed-point objects from Fixed-Point Designer software. You can see this when you display a fixed-point filter at the command prompt.

```
hd=dfilt.df2t

hd =

         FilterStructure: 'Direct-Form II Transposed'
              Arithmetic: 'double'
```

```
              Numerator: 1
            Denominator: 1
       PersistentMemory: false
                 States: [0x1 double]

set(hd,'arithmetic','fixed')
hd

hd =

        FilterStructure: 'Direct-Form II Transposed'
             Arithmetic: 'fixed'
              Numerator: 1
            Denominator: 1
       PersistentMemory: false
                 States: [1x1 embedded.fi]

        CoeffWordLength: 16
         CoeffAutoScale: true
                 Signed: true

        InputWordLength: 16
        InputFracLength: 15

       OutputWordLength: 16
       OutputFracLength: 15

        StateWordLength: 16
         StateAutoScale: true

            ProductMode: 'FullPrecision'

        AccumWordLength: 40
          CastBeforeSum: true

              RoundMode: 'convergent'
           OverflowMode: 'wrap'
```

Look at the States property, shown here

```
States: [1x1 embedded.fi]
```

The notation `embedded.fi` indicates that the states are being represented by fixed-point objects, usually called `fi` objects. If you take a closer look at the property `States`, you see how the properties of the `fi` object represent the values for the filter states.

```
hd.states

ans =

[]

              DataType: Fixed
               Scaling: BinaryPoint
                Signed: true
            WordLength: 16
        FractionLength: 15

             RoundMode: round
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

To learn more about `fi` objects (fixed-point objects) in general, refer to your Fixed-Point Designer documentation.

As inputs (data to be filtered), fixed-point filters accept both regular double-precision values and `fi` objects. Which you use depends on your needs. How your filter responds to the input data is determined by the settings of the filter properties, discussed in the next few sections.

Summary — Fixed-Point Filter Properties

Discrete-time filters in this toolbox use objects that perform the filtering and configuration of the filter. As objects, they include properties and methods that are often referred to as functions — not strictly the same as MATLAB functions but mostly so) to provide filtering capability. In discrete-time filters, or `dfilt` objects, many of the properties are dynamic, meaning they become available depending on the settings of other properties in the `dfilt` object or filter.

Dynamic Properties

When you use a `dfilt.`*`structure`* function to create a filter, MATLAB displays the filter properties in the command window in return (unless you end the command with a semicolon which suppresses the output display). Generally you see six or seven properties, ranging from the property `FilterStructure` to `PersistentMemory`. These first properties are always present in the filter. One of the most important properties is `Arithmetic`. The `Arithmetic` property controls all of the dynamic properties for a filter.

Dynamic properties become available when you change another property in the filter. For example, when you change the `Arithmetic` property value to `fixed`, the display now shows many more properties for the filter, all of them considered dynamic. Here is an example that uses a direct form II filter. First create the default filter:

```
hd=dfilt.df2

hd =

         FilterStructure: 'Direct-Form II'
              Arithmetic: 'double'
               Numerator: 1
             Denominator: 1
        PersistentMemory: false
                  States: [0x1 double]
```

With the filter `hd` in the workspace, convert the arithmetic to fixed-point. Do this by setting the property `Arithmetic` to `fixed`. Notice the display. Instead of a few properties, the filter now has many more, each one related to a particular part of the filter and its operation. Each of the now-visible properties is dynamic.

```
hd.arithmetic='fixed'

hd =

         FilterStructure: 'Direct-Form II'
              Arithmetic: 'fixed'
               Numerator: 1
             Denominator: 1
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
```

```
     OutputMode: 'AvoidOverflow'

StateWordLength: 16
StateFracLength: 15

    ProductMode: 'FullPrecision'

AccumWordLength: 40
  CastBeforeSum: true

      RoundMode: 'convergent'
   OverflowMode: 'wrap'
```

Even this list of properties is not yet complete. Changing the value of other properties such as the `ProductMode` or `CoeffAutoScale` properties may reveal even more properties that control how the filter works. Remember this feature about `dfilt` objects and dynamic properties as you review the rest of this section about properties of fixed-point filters.

An important distinction is you cannot change the value of a property unless you see the property listed in the default display for the filter. Entering the filter name at the MATLAB prompt generates the default property display for the named filter. Using `get(filtername)` does not generate the default display — it lists all of the filter properties, both those that you can change and those that are not available yet.

The following table summarizes the properties, static and dynamic, of fixed-point filters and provides a brief description of each. Full descriptions of each property, in alphabetical order, follow the table.

Property Name	Valid Values [Default Value]	Brief Description
`AccumFracLength`	Any positive or negative integer number of bits [29]	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — `DenAccumFracLength` and `NumAccumFracLength` — that let you set the precision for numerator and denominator operations separately.
`AccumWordLength`	Any positive integer number of bits [40]	Sets the word length used to store data in the accumulator/buffer.

Property Name	Valid Values [Default Value]	Brief Description
`Arithmetic`	[Double], single, fixed	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
`CastBeforeSum`	[True] or false	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
`CoeffAutoScale`	[True] or false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` and `DenFracLength` properties to specify the precision used.
`CoeffFracLength`	Any positive or negative integer number of bits [14]	Set the fraction length the filter uses to interpret coefficients. `CoeffFracLength` is not available until you set `CoeffAutoScale` to `false`. Scalar filters include this property.
`CoeffWordLength`	Any positive integer number of bits [16]	Specifies the word length to apply to filter coefficients.
`DenAccumFracLength`	Any positive or negative integer number of bits [29]	Specifies how the filter algorithm interprets the results of addition operations involving denominator coefficients.
`DenFracLength`	Any positive or negative integer number of bits [14]	Sets the fraction length the filter uses to interpret denominator coefficients. `DenFracLength` is always available, but it is read-only until you set `CoeffAutoScale` to `false`.
`Denominator`	Any filter coefficient value [1]	Holds the denominator coefficients for IIR filters.

Property Name	Valid Values [Default Value]	Brief Description
DenProdFracLength	Any positive or negative integer number of bits [29]	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value after you set ProductMode to SpecifyPrecision.
DenStateFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.
FracDelay	Any decimal value between 0 and 1 samples	Specifies the fractional delay provided by the filter, in decimal fractions of a sample.
FDAutoScale	[True] or false	Specifies whether the filter automatically chooses the proper scaling to represent the fractional delay value without overflowing. Turning this off by setting the value to false enables you to change the FDWordLength and FDFracLength properties to specify the data format applied.
FDFracLength	Any positive or negative integer number of bits [5]	Specifies the fraction length to represent the fractional delay.
FDProdFracLength	Any positive or negative integer number of bits [34]	Specifies the fraction length to represent the result of multiplying the coefficients with the fractional delay.
FDProdWordLength	Any positive or negative integer number of bits [39]	Specifies the word length to represent result of multiplying the coefficients with the fractional delay.
FDWordLength	Any positive or negative integer number of bits [6]	Specifies the word length to represent the fractional delay.
DenStateWordLength	Any positive integer number of bits [16]	Specifies the word length used to represent the states associated with denominator coefficients in the filter.

Property Name	Valid Values [Default Value]	Brief Description
`FilterInternals`	`[FullPrecision]`, `SpecifyPrecision`	Controls whether the filter sets the output word and fraction lengths, and the accumulator word and fraction lengths automatically to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` exposes the output and accumulator related properties so you can set your own word and fraction lengths for them.
`FilterStructure`	Not applicable.	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
`InputFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret data to be processed by the filter.
`InputWordLength`	Any positive integer number of bits [16]	Specifies the word length applied to represent input data.
`Ladder`	Any ladder coefficients in double-precision data type [1]	`latticearma` filters include this property to store the ladder coefficients.
`LadderAccumFrac Length`	Any positive or negative integer number of bits [29]	`latticearma` filters use this to define the fraction length applied to values output by the accumulator that stores the results of ladder computations.
`LadderFracLength`	Any positive or negative integer number of bits [14]	`latticearma` filters use ladder coefficients in the signal flow. This property determines the fraction length used to interpret the coefficients.
`Lattice`	Any lattice structure coefficients. No default value.	Stores the lattice coefficients for lattice-based filters.

Property Name	Valid Values [Default Value]	Brief Description
`LatticeAccumFracLength`	Any positive or negative integer number of bits [29]	Specifies how the accumulator outputs the results of operations on the lattice coefficients.
`LatticeFracLength`	Any positive or negative integer number of bits [15]	Specifies the fraction length applied to the lattice coefficients.
`MultiplicandFracLength`	Any positive or negative integer number of bits [15]	Sets the fraction length for values used in product operations in the filter. Direct-form I transposed (df1t) filter structures include this property.
`MultiplicandWordLength`	Any positive integer number of bits [16]	Sets the word length applied to the values input to a multiply operation (the multiplicands). The filter structure df1t includes this property.
`NumAccumFracLength`	Any positive or negative integer number of bits [29]	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients.
`Numerator`	Any double-precision filter coefficients [1]	Holds the numerator coefficient values for the filter.
`NumFracLength`	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
`NumProdFracLength`	Any positive or negative integer number of bits [29]	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. You can change the property value after you set `ProductMode` to `SpecifyPrecision`.
`NumStateFracLength`	Any positive or negative integer number of bits [15]	For IIR filters, this defines the fraction length applied to the numerator states of the filter. Specifies the fraction length used to interpret the states associated with numerator coefficients in the filter.

Property Name	Valid Values [Default Value]	Brief Description
`NumStateWordLength`	Any positive integer number of bits [16]	For IIR filters, this defines the word length applied to the numerator states of the filter. Specifies the word length used to interpret the states associated with numerator coefficients in the filter.
`OutputFracLength`	Any positive or negative integer number of bits — [15] or [12] bits depending on the filter structure	Determines how the filter interprets the filtered data. You can change the value of `OutputFracLength` after you set `OutputMode` to `SpecifyPrecision`.
`OutputMode`	[AvoidOverflow], BestPrecision, SpecifyPrecision	Sets the mode the filter uses to scale the filtered input data. You have the following choices: • `AvoidOverflow` — directs the filter to set the output data fraction length to avoid causing the data to overflow. • `BestPrecision` — directs the filter to set the output data fraction length to maximize the precision in the output data. • `SpecifyPrecision` — lets you set the fraction length used by the filtered data.
`OutputWordLength`	Any positive integer number of bits [16]	Determines the word length used for the filtered data.

Property Name	Valid Values [Default Value]	Brief Description
OverflowMode	Saturate or [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — hey maintain full precision.
ProductFracLength	Any positive or negative integer number of bits [29]	For the output from a product operation, this sets the fraction length used to interpret the numeric data. This property becomes writable (you can change the value) after you set ProductMode to SpecifyPrecision.
ProductMode	[FullPrecision], KeepLSB, KeepMSB, SpecifyPrecision	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Any positive number of bits. Default is 16 or 32 depending on the filter structure	Specifies the word length to use for the results of multiplication operations. This property becomes writable (you can change the value) after you set ProductMode to SpecifyPrecision.

Property Name	Valid Values [Default Value]	Brief Description
`PersistentMemory`	`True` or `[false]`	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `True` is the default setting.
`RoundMode`	[Convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `ceil` - Round toward positive infinity. • `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • `fix` - Round toward zero. • `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Property Name	Valid Values [Default Value]	Brief Description
`ScaleValueFracLength`	Any positive or negative integer number of bits [29]	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Available only when you disable `CoeffAutoScale` by setting it to `false`.
`ScaleValues`	[2 x 1 double] array with values of 1	Stores the scaling values for sections in SOS filters.
`Signed`	[True] or false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
`sosMatrix`	`[1 0 0 1 0 0]`	Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/section datatype]. A `[15x6 double]` SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type `double` to represent the coefficients.
`SectionInputAuto Scale`	[True] or false	Specifies whether the filter automatically chooses the proper fraction length to prevent overflow by data entering a section of an SOS filter. Setting this property to `false` enables you to change the Section`InputFracLength` property to specify the precision used. Available only for SOS filters.

Property Name	Valid Values [Default Value]	Brief Description
SectionInputFrac Length	Any positive or negative integer number of bits [29]	Section values work with SOS filters. Setting this property controls how your filter interprets the section values between sections of the filter by setting the fraction length. This applies to data entering a section. Compare to SectionOutputFracLength. Available only when you disable SectionInputAutoScale by setting it to false.
SectionInputWord Length	Any positive or negative integer number of bits [29]	Sets the word length used to represent the data moving into a section of an SOS filter.
SectionOutputAuto Scale	[True] or false	Specifies whether the filter automatically chooses the proper fraction length to prevent overflow by data leaving a section of an SOS filter. Setting this property to false enables you to change the SectionOutputFracLength property to specify the precision used.
SectionOutputFrac Length	Any positive or negative integer number of bits [29]	Section values work with SOS filters. Setting this property controls how your filter interprets the section values between sections of the filter by setting the fraction length. This applies to data leaving a section. Compare to SectionInputFracLength. Available after you disable SectionOutputAutoScale by setting it to false.
SectionOutputWord Length	Any positive or negative integer number of bits [32]	Sets the word length used to represent the data moving out of one section of an SOS filter.
StateFracLength	Any positive or negative integer number of bits [15]	Lets you set the fraction length applied to interpret the filter states.

Property Name	Valid Values [Default Value]	Brief Description
`States`	[1x1 embedded `fi`]	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Notice that the states use `fi` objects, with the associated properties from those objects. For details, refer to `filtstates` in your Signal Processing Toolbox documentation or in the Help system.
`StateWordLength`	Any positive integer number of bits [16]	Sets the word length used to represent the filter states.
`TapSumFracLength`	Any positive or negative integer number of bits [15]	Sets the fraction length used to represent the filter tap values in addition operations. This is available after you set `TapSumMode` to `false`. Symmetric and antisymmetric FIR filters include this property.
`TapSumMode`	FullPrecision, KeepLSB, [KeepMSB], SpecifyPrecision	Determines how the accumulator outputs stored that involve filter tap weights. Choose from full precision (`FullPrecision`) to prevent overflows, or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when outputting results from the accumulator. To let you set the precision (the fraction length) used by the output from the accumulator, set `FilterInternals` to `SpecifyPrecision`. Symmetric and antisymmetric FIR filters include this property.
`TapSumWordLength`	Any positive number of bits [17]	Sets the word length used to represent the filter tap weights during addition. Symmetric and antisymmetric FIR filters include this property.

Property Details for Fixed-Point Filters

When you create a fixed-point filter, you are creating a filter object (a `dfilt` object). In this manual, the terms filter, `dfilt` object, and filter object are used interchangeably. To filter data, you apply the filter object to your data set. The output of the operation is the data filtered by the filter and the filter property values.

Filter objects have properties to which you assign property values. You use these property values to assign various characteristics to the filters you create, including

- The type of arithmetic to use in filtering operations
- The structure of the filter used to implement the filter (not a property you can set or change — you select it by the `dfilt.`*`structure`* function you choose)
- The locations of quantizations and cast operations in the filter
- The data formats used in quantizing, casting, and filtering operations

Details of the properties associated with fixed-point filters are described in alphabetical order on the following pages.

AccumFracLength

Except for state-space filters, all `dfilt` objects that use fixed arithmetic have this property that defines the fraction length applied to data in the accumulator. Combined with `AccumWordLength`, `AccumFracLength` helps fully specify how the accumulator outputs data after processing addition operations. As with all fraction length properties, `AccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers.

AccumWordLength

You use `AccumWordLength` to define the data word length used in the accumulator. Set this property to a value that matches your intended hardware. For example, many digital signal processors use 40-bit accumulators, so set `AccumWordLength` to 40 in your fixed-point filter:

```
set(hq,'arithmetic','fixed');
set(hq,'AccumWordLength',40);
```

Note that `AccumWordLength` only applies to filters whose `Arithmetic` property value is `fixed`.

Arithmetic

Perhaps the most important property when you are working with `dfilt` objects, `Arithmetic` determines the type of arithmetic the filter uses, and the properties or quantizers that compose the fixed-point or quantized filter. You use strings to set the `Arithmetic` property value.

The next table shows the valid strings for the Arithmetic property. Following the table, each property string appears with more detailed information about what happens when you select the string as the value for `Arithmetic` in your `dfilt`.

Arithmetic Property String	Brief Description of Effect on the Filter
`double`	All filtering operations and coefficients use double-precision floating-point representations and math. When you use `dfilt.`*`structure`* to create a filter object, `double` is the default value for the `Arithmetic` property.
`single`	All filtering operations and coefficients use single-precision floating-point representations and math.
`fixed`	This string applies selected default values for the properties in the fixed-point filter object, including such properties as coefficient word lengths, fraction lengths, and various operating modes. Generally, the default values match those you use on many digital signal processors. Allows signed fixed data types only. Fixed-point arithmetic filters are available only when you install Fixed-Point Designer software with this toolbox.

double

When you use one of the `dfilt.`*`structure`* methods to create a filter, the `Arithmetic` property value is `double` by default. Your filter is identical to the same filter without the `Arithmetic` property, as you would create if you used Signal Processing Toolbox software.

`Double` means that the filter uses double-precision floating-point arithmetic in all operations while filtering:

- All input to the filter must be double data type. Any other data type returns an error.
- The states and output are doubles as well.

- All internal calculations are done in double math.

When you use `double` data type filter coefficients, the reference and quantized (fixed-point) filter coefficients are identical. The filter stores the reference coefficients as double data type.

single

When your filter should use single-precision floating-point arithmetic, set the `Arithmetic` property to `single` so all arithmetic in the filter processing gets restricted to single-precision data type.

- Input data must be single data type. Other data types return errors.
- The filter states and filter output use single data type.

When you choose `single`, you can provide the filter coefficients in either of two ways:

- Double data type coefficients. With `Arithmetic` set to `single`, the filter casts the double data type coefficients to single data type representation.
- Single data type. These remain unchanged by the filter.

Depending on whether you specified single or double data type coefficients, the reference coefficients for the filter are stored in the data type you provided. If you provide coefficients in double data type, the reference coefficients are double as well. Providing single data type coefficients generates single data type reference coefficients. Note that the arithmetic used by the reference filter is always double.

When you use `reffilter` to create a reference filter from the reference coefficients, the resulting filter uses double-precision versions of the reference filter coefficients.

To set the `Arithmetic` property value, create your filter, then use `set` to change the `Arithmetic` setting, as shown in this example using a direct form FIR filter.

```
b=fir1(7,0.45);

hd=dfilt.dffir(b)

hd =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'double'
               Numerator: [1x8 double]
        PersistentMemory: false
```

```
                 States: [7x1 double]

set(hd,'arithmetic','single')
hd

hd =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'single'
               Numerator: [1x8 double]
        PersistentMemory: false
                  States: [7x1 single]
```

fixed

Converting your `dfilt` object to use fixed arithmetic results in a filter structure that uses properties and property values to match how the filter would behave on digital signal processing hardware.

Note The `fixed` option for the property `Arithmetic` is available only when you install Fixed-Point Designer software as well as DSP System Toolbox software.

After you set `Arithmetic` to `fixed`, you are free to change any property value from the default value to a value that more closely matches your needs. You cannot, however, mix floating-point and fixed-point arithmetic in your filter when you select `fixed` as the `Arithmetic` property value. Choosing `fixed` restricts you to using either fixed-point or floating point throughout the filter (the data type must be homogenous). Also, all data types must be signed. `fixed` does not support unsigned data types except for unsigned coefficients when you set the property `Signed` to `false`. Mixing word and fraction lengths within the fixed object is acceptable. In short, using fixed arithmetic assumes

- fixed word length.
- fixed size and dedicated accumulator and product registers.
- the ability to do either saturation or wrap arithmetic.
- that multiple rounding modes are available.

Making these assumptions simplifies your job of creating fixed-point filters by reducing repetition in the filter construction process, such as only requiring you to enter the accumulator word size once, rather than for each step that uses the accumulator.

Default property values are a starting point in tailoring your filter to common hardware, such as choosing 40-bit word length for the accumulator, or 16-bit words for data and coefficients.

In this `dfilt` object example, `get` returns the default values for `dfilt.df1t` structures.

```
[b,a]=butter(6,0.45);
hd=dfilt.df1(b,a)

hd =

         FilterStructure: 'Direct-Form I'
              Arithmetic: 'double'
               Numerator: [1x7 double]
             Denominator: [1x7 double]
        PersistentMemory: false
                  States: Numerator:  [6x1 double]
                          Denominator:[6x1 double]


set(hd,'arithmetic','fixed')
get(hd)
        PersistentMemory: false
         FilterStructure: 'Direct-Form I'
                  States: [1x1 filtstates.dfiir]
               Numerator: [1x7 double]
             Denominator: [1x7 double]
              Arithmetic: 'fixed'
         CoeffWordLength: 16
          CoeffAutoScale: 1
                  Signed: 1
               RoundMode: 'convergent'
            OverflowMode: 'wrap'
         InputWordLength: 16
         InputFracLength: 15
             ProductMode: 'FullPrecision'
        OutputWordLength: 16
        OutputFracLength: 15
           NumFracLength: 16
           DenFracLength: 14
       ProductWordLength: 32
       NumProdFracLength: 31
       DenProdFracLength: 29
         AccumWordLength: 40
```

```
      NumAccumFracLength: 31
      DenAccumFracLength: 29
           CastBeforeSum: 1
```

Here is the default display for `hd`.

```
hd

hd =

         FilterStructure: 'Direct-Form I'
              Arithmetic: 'fixed'
               Numerator: [1x7 double]
             Denominator: [1x7 double]
        PersistentMemory: false
                  States: Numerator:  [6x1 fi]
                          Denominator:[6x1 fi]


         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
        OutputFracLength: 15

             ProductMode: 'FullPrecision'

         AccumWordLength: 40
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'
```

This second example shows the default property values for `dfilt.latticemamax` filter objects, using the coefficients from an `fir1` filter.

```
b=fir1(7,0.45)


hdlat=dfilt.latticemamax(b)
```

```
hdlat =

          FilterStructure: [1x45 char]
               Arithmetic: 'double'
                  Lattice: [1x8 double]
         PersistentMemory: false
                   States: [8x1 double]

hdlat.arithmetic='fixed'

hdlat =

          FilterStructure: [1x45 char]
               Arithmetic: 'fixed'
                  Lattice: [1x8 double]
         PersistentMemory: false
                   States: [1x1 embedded.fi]

          CoeffWordLength: 16
           CoeffAutoScale: true
                   Signed: true

          InputWordLength: 16
          InputFracLength: 15

         OutputWordLength: 16
               OutputMode: 'AvoidOverflow'

          StateWordLength: 16
          StateFracLength: 15

              ProductMode: 'FullPrecision'

          AccumWordLength: 40
            CastBeforeSum: true

                RoundMode: 'convergent'
             OverflowMode: 'wrap'
```

Unlike the `single` or `double` options for `Arithmetic`, `fixed` uses properties to define the word and fraction lengths for each portion of your filter. By changing the property value of any of the properties, you control your filter performance. Every word length and fraction length property is independent — set the one you need and the others remain unchanged, such as setting the input word length with InputWordLength, while leaving the fraction length the same.

```
d=fdesign.lowpass('n,fc',6,0.45)

d =

               Response: 'Lowpass with cutoff'
          Specification: 'N,Fc'
            Description: {2x1 cell}
    NormalizedFrequency: true
                     Fs: 'Normalized'
            FilterOrder: 6
                Fcutoff: 0.4500


designmethods(d)


Design Methods for class fdesign.lowpass:


butter

hd=butter(d)

hd =

 FilterStructure: 'Direct-Form II, Second-Order Sections'
      Arithmetic: 'double'
       sosMatrix: [3x6 double]
     ScaleValues: [4x1 double]
PersistentMemory: false
          States: [2x3 double]

hd.arithmetic='fixed'

hd =

  FilterStructure: 'Direct-Form II, Second-Order Sections'
       Arithmetic: 'fixed'
        sosMatrix: [3x6 double]
      ScaleValues: [4x1 double]
 PersistentMemory: false
```

```
           States: [1x1 embedded.fi]

   CoeffWordLength: 16
    CoeffAutoScale: true
            Signed: true

   InputWordLength: 16
   InputFracLength: 15

  SectionInputWordLength: 16
   SectionInputAutoScale: true

  SectionOutputWordLength: 16
  Section OutputAutoScale: true

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

         StateWordLength: 16
         StateFracLength: 15

             ProductMode: 'FullPrecision'

         AccumWordLength: 40
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'

hd.inputWordLength=12

hd =

 FilterStructure: 'Direct-Form II, Second-Order Sections'
      Arithmetic: 'fixed'
       sosMatrix: [3x6 double]
     ScaleValues: [4x1 double]
PersistentMemory: false
                  States: [1x1 embedded.fi]

  CoeffWordLength: 16
   CoeffAutoScale: true
           Signed: true
```

```
  InputWordLength: 12
  InputFracLength: 15

  SectionInputWordLength: 16
   SectionInputAutoScale: true

 SectionOutputWordLength: 16
  SectionOutputAutoScale: true

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

         StateWordLength: 16
         StateFracLength: 15

             ProductMode: 'FullPrecision'

         AccumWordLength: 40
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'
```

Notice that the properties for the lattice filter `hdlat` and direct-form II filter `hd` are different, as befits their differing filter structures. Also, some properties are common to both objects, such as `RoundMode` and `PersistentMemory` and behave the same way in both objects.

Notes About Fraction Length, Word Length, and Precision

Word length and fraction length combine to make the format for a fixed-point number, where word length is the number of bits used to represent the value and fraction length specifies, in bits, the location of the binary point in the fixed-point representation. Therein lies a problem — fraction length, which you specify in bits, can be larger than the word length, or a negative number of bits. This section explains how that idea works and how you might use it.

Unfortunately fraction length is somewhat misnamed (although it continues to be used in this User's Guide and elsewhere for historical reasons).

Fraction length defined as the number of fractional bits (bits to the right of the binary point) is true only when the fraction length is positive and less than or equal to the word length. In MATLAB format notation you can use [word length fraction length]. For example, for the format [16 16], the second 16 (the fraction length) is the number of fractional bits or bits to the right of the binary point. In this example, all 16 bits are to the right of the binary point.

But it is also possible to have fixed-point formats of [16 18] or [16 -45]. In these cases the fraction length can no longer be the number of bits to the right of the binary point since the format says the word length is 16 — there cannot be 18 fraction length bits on the right. And how can there be a negative number of bits for the fraction length, such as [16 -45]?

A better way to think about fixed-point format [word length fraction length] and what it means is that the representation of a fixed-point number is a weighted sum of powers of two driven by the fraction length, or the two's complement representation of the fixed-point number.

Consider the format [B L], where the fraction length L can be positive, negative, 0, greater than B (the word length) or less than B. (B and L are always integers and B is always positive.)

Given a binary string b(1) b(2) b(3) ... b(B), to determine the two's-complement value of the string in the format described by [B L], use the value of the individual bits in the binary string in the following formula, where b(1) is the first binary bit (and most significant bit, MSB), b(2) is the second, and on up to b(B).

The decimal numeric value that those bits represent is given by

```
value =-b(1)*2^(B-L-1)+b(2)*2^(B-L-2)+b(3)*2^(B-L-3)+...+ b(B)*2^(-L)
```

L, the fraction length, represents the negative of the weight of the last, or least significant bit (LSB). L is also the step size or the precision provided by a given fraction length.

Precision

Here is how precision works.

When all of the bits of a binary string are zero except for the LSB (which is therefore equal to one), the value represented by the bit string is given by $2^{(-L)}$. If L is negative, for example L=-16, the value is 2^{16}. The smallest step between numbers that can be represented in a format where L=-16 is given by 1 x 2^{16} (the rightmost term in the formula above), which is 65536. Note the precision does not depend on the word length.

Take a look at another example. When the word length set to 8 bits, the decimal value 12 is represented in binary by 00001100. That 12 is the decimal equivalent of 00001100 tells you that you are using [8 0] data format representation — the word length is 8 bits and fraction length 0 bits, and the step size or precision (the smallest difference between two adjacent values in the format [8,0], is 2^0=1.

Suppose you plan to keep only the upper 5 bits and discard the other three. The resulting precision after removing the right-most three bits comes from the weight of the lowest remaining bit, the fifth bit from the left, which is 2^3=8, so the format would be [5,-3].

Note that in this format the step size is 8, I cannot represent numbers that are between multiples of 8.

In MATLAB, with Fixed-Point Designer software installed:

```
x=8;
q=quantizer([8,0]); % Word length = 8, fraction length = 0
xq=quantize(q,x);
binxq=num2bin(q,xq);
q1=quantizer([5 -3]); % Word length = 5, fraction length = -3
xq1 = quantize(q1,xq);
binxq1=num2bin(q1,xq1);
binxq

binxq =

00001000

binxq1

binxq1 =

00001
```

But notice that in [5,-3] format, 00001 is the two's complement representation for 8, not for 1; `q = quantizer([8 0])` and `q1 = quantizer([5 -3])` are not the same. They cover the about the same range — `range(q)>range(q1)` — but their quantization step is different — `eps(q)`= 8, and `eps(q1)=1`.

Look at one more example. When you construct a quantizer `q`

```
q = quantizer([a,b])
```

the first element in `[a,b]` is `a`, the word length used for quantization. The second element in the expression, `b`, is related to the quantization step — the numerical difference between the two closest values that the quantizer can represent. This is also related to the weight given to the LSB. Note that `2^(-b)` = `eps(q)`.

Now construct two quantizers, q1 and q2. Let q1 use the format [32,0] and let q2 use the format [16, -16].

```
q1 = quantizer([32,0])
q2 = quantizer([16,-16])
```

Quantizers q1 and q2 cover the same range, but q2 has less precision. It covers the range in steps of 2^{16}, while q covers the range in steps of 1.

This lost precision is due to (or can be used to model) throwing out 16 least-significant bits.

An important point to understand is that in dfilt objects and filtering you control which bits are carried from the sum and product operations in the filter to the filter output by setting the format for the output from the sum or product operation.

For instance, if you use [16 0] as the output format for a 32-bit result from a sum operation when the original format is [32 0], you take the lower 16 bits from the result. If you use [16 -16], you take the higher 16 bits of the original 32 bits. You could even take 16 bits somewhere in between the 32 bits by choosing something like [16 -8], but you probably do not want to do that.

Filter scaling is directly implicated in the format and precision for a filter. When you know the filter input and output formats, as well as the filter internal formats, you can scale the inputs or outputs to stay within the format ranges.

Notice that overflows or saturation might occur at the filter input, filter output, or within the filter itself, such as during add or multiply or accumulate operations. Improper scaling at any point in the filter can result in numerical errors that dramatically change the performance of your fixed-point filter implementation.

CastBeforeSum

Setting the CastBeforeSum property determines how the filter handles the input values to sum operations in the filter. After you set your filter Arithmetic property value to fixed, you have the option of using CastBeforeSum to control the data type of some inputs (addends) to summations in your filter. To determine which addends reflect the CastBeforeSum property setting, refer to the reference page for the signal flow diagram for the filter structure.

CastBeforeSum specifies whether to cast selected addends to summations in the filter to the output format from the addition operation before performing the addition. When you specify `true` for the property value, the results of the affected sum operations match most closely the results found on most digital signal processors. Performing the cast operation before the summation adds one or two additional quantization operations that can add error sources to your filter results.

Specifying `CastBeforeSum` to be `false` prevents the addends from being cast to the output format before the addition operation. Choose this setting to get the most accurate results from summations without considering the hardware your filter might use.

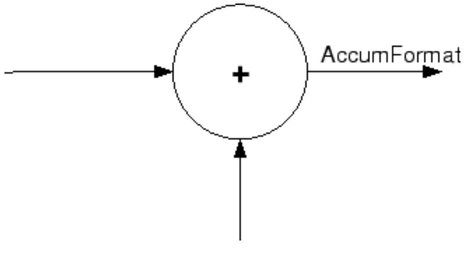
Notice that the output format for every sum operation reflects the value of the output property specified in the filter structure diagram. Which input property is referenced by `CastBeforeSum` depends on the structure.

Property Value	Description
`false`	Configures filter summation operations to retain the addends in the format carried from the previous operation.
`true`	Configures filter summation operations to convert the input format of the addends to match the summation output format before performing the summation operation. Usually this generates results from the summation that more closely match those found from digital signal processors

Another point — with `CastBeforeSum` set to `false`, the filter realization process inserts an intermediate data type format to hold temporarily the full precision sum of the inputs. A separate Convert block performs the process of casting the addition result to the accumulator format. This intermediate data format occurs because the Sum block in Simulink always casts input (addends) to the output data type.

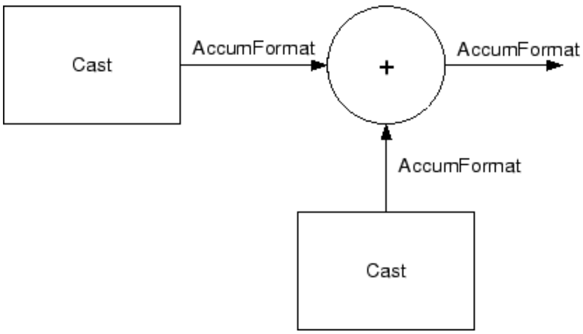
Diagrams of CastBeforeSum Settings

When `CastBeforeSum` is `false`, sum elements in filter signal flow diagrams look like this:



showing that the input data to the sum operations (the addends) retain their format word length and fraction length from previous operations. The addition process uses the existing input formats and then casts the output to the format defined by `AccumFormat`. Thus the output data has the word length and fraction length defined by `AccumWordLength` and `AccumFracLength`.

When `CastBeforeSum` is `true`, sum elements in filter signal flow diagrams look like this:



showing that the input data gets recast to the accumulator format word length and fraction length (AccumFormat) before the sum operation occurs. The data output by the addition operation has the word length and fraction length defined by `AccumWordLength` and `AccumFracLength`.

CoeffAutoScale

How the filter represents the filter coefficients depends on the property value of `CoeffAutoScale`. When you create a `dfilt` object, you use coefficients in double-precision format. Converting the `dfilt` object to fixed-point arithmetic forces the coefficients into a fixed-point representation. The representation the filter uses depends on whether the value of `CoeffAutoScale` is `true` or `false`.

- `CoeffAutoScale` = `true` means the filter chooses the fraction length to maintain the value of the coefficients as close to the double-precision values as possible. When you change the word length applied to the coefficients, the filter object changes the fraction length to try to accommodate the change. `true` is the default setting.
- `CoeffAutoScale` = `false` removes the automatic scaling of the fraction length for the coefficients and exposes the property that controls the coefficient fraction length

so you can change it. For example, if the filter is a direct form FIR filter, setting `CoeffAutoScale` = `false` exposes the `NumFracLength` property that specifies the fraction length applied to numerator coefficients. If the filter is an IIR filter, setting `CoeffAutoScale` = `false` exposes both the `NumFracLength` and `DenFracLength` properties.

Here is an example of using `CoeffAutoScale` with a direct form filter.

```
hd2=dfilt.dffir([0.3 0.6 0.3])

hd2 =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'double'
               Numerator: [0.3000 0.6000 0.3000]
        PersistentMemory: false
                  States: [2x1 double]

hd2.arithmetic='fixed'

hd2 =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'fixed'
               Numerator: [0.3000 0.6000 0.3000]
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

             ProductMode: 'FullPrecision'

         AccumWordLength: 40
           CastBeforeSum: true

               RoundMode: 'convergent'
```

```
             OverflowMode: 'wrap'
```

To this point, the filter coefficients retain the original values from when you created the filter as shown in the `Numerator` property. Now change the `CoeffAutoScale` property value from `true` to `false`.

```
hd2.coeffautoScale=false

hd2 =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'fixed'
               Numerator: [0.3000 0.6000 0.3000]
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: false
           NumFracLength: 15
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

             ProductMode: 'FullPrecision'

         AccumWordLength: 40
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'
```

With the `NumFracLength` property now available, change the word length to 5 bits.

Notice the coefficient values. Setting `CoeffAutoScale` to `false` removes the automatic fraction length adjustment and the filter coefficients cannot be represented by the current format of [5 15] — a word length of 5 bits, fraction length of 15 bits.

```
hd2.coeffwordlength=5

hd2 =
```

```
         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'fixed'
               Numerator: [4.5776e-004 4.5776e-004 4.5776e-004]
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 5
          CoeffAutoScale: false
           NumFracLength: 15
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

             ProductMode: 'FullPrecision'

         AccumWordLength: 40
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'
```

Restoring `CoeffAutoScale` to `true` goes some way to fixing the coefficient values. Automatically scaling the coefficient fraction length results in setting the fraction length to 4 bits. You can check this with `get(hd2)` as shown below.

```
hd2.coeffautoScale=true

hd2 =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'fixed'
               Numerator: [0.3125 0.6250 0.3125]
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 5
          CoeffAutoScale: true
                  Signed: true
```

```
          InputWordLength: 16
          InputFracLength: 15

         OutputWordLength: 16
               OutputMode: 'AvoidOverflow'

              ProductMode: 'FullPrecision'

          AccumWordLength: 40
            CastBeforeSum: true

                RoundMode: 'convergent'
             OverflowMode: 'wrap'

get(hd2)
        PersistentMemory: false
FilterStructure: 'Direct-Form FIR'
                  States: [1x1 embedded.fi]
               Numerator: [0.3125 0.6250 0.3125]
              Arithmetic: 'fixed'
         CoeffWordLength: 5
          CoeffAutoScale: 1
                  Signed: 1
               RoundMode: 'convergent'
            OverflowMode: 'wrap'
         InputWordLength: 16
         InputFracLength: 15
        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'
             ProductMode: 'FullPrecision'
           NumFracLength: 4
        OutputFracLength: 12
       ProductWordLength: 21
       ProductFracLength: 19
         AccumWordLength: 40
         AccumFracLength: 19
           CastBeforeSum: 1
```

Clearly five bits is not enough to represent the coefficients accurately.

CoeffFracLength

Fixed-point scalar filters that you create using `dfilt.scalar` use this property to define the fraction length applied to the scalar filter coefficients. Like the coefficient-fractionlength-related properties for the FIR, lattice, and IIR filters, `CoeffFracLength` is not displayed for scalar filters until you set `CoeffAutoScale` to `false`. Once you change the automatic scaling you can set the fraction length for the coefficients to any value you require.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 14 bits, with the `CoeffWordlength` of 16 bits.

CoeffWordLength

One primary consideration in developing filters for hardware is the length of a data word. `CoeffWordLength` defines the word length for these data storage and arithmetic locations:

- Numerator and denominator filter coefficients
- Tap sum in `dfilt.dfsymfir` and `dfilt.dfasymfir` filter objects
- Section input, multiplicand, and state values in direct-form SOS filter objects such as `dfilt.df1t` and `dfilt.df2`
- Scale values in second-order filters
- Lattice and ladder coefficients in lattice filter objects, such as `dfilt.latticearma` and `dfilt.latticemamax`
- Gain in `dfilt.scalar`

Setting this property value controls the word length for the data listed. In most cases, the data words in this list have separate fraction length properties to define the associated fraction lengths.

Any positive, integer word length works here, limited by the machine you use to develop your filter and the hardware you use to deploy your filter.

DenAccumFracLength

Filter structures `df1`, `df1t`, `df2`, and `df2t` that use `fixed` arithmetic have this property that defines the fraction length applied to denominator coefficients in the accumulator. In combination with `AccumWordLength`, the properties fully specify how the accumulator outputs data stored there.

As with all fraction length properties, `DenAccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers.

To be able to change the property value for this property, you set `FilterInternals` to `SpecifyPrecision`.

DenFracLength

Property `DenFracLength` contains the value that specifies the fraction length for the denominator coefficients for your filter. `DenFracLength` specifies the fraction length used to interpret the data stored in `C`. Used in combination with `CoeffWordLength`, these two properties define the interpretation of the coefficients stored in the vector that contains the denominator coefficients.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 15 bits, with the `CoeffWordLength` of 16 bits.

Denominator

The denominator coefficients for your IIR filter, taken from the prototype you start with, are stored in this property. Generally this is a 1-by-N array of data in double format, where N is the length of the filter.

All IIR filter objects include `Denominator`, except the lattice-based filters which store their coefficients in the `Lattice` property, and second-order section filters, such as `dfilt.df1tsos`, which use the `SosMatrix` property to hold the coefficients for the sections.

DenProdFracLength

A property of all of the direct form IIR `dfilt` objects, except the ones that implement second-order sections, `DenProdFracLength` specifies the fraction length applied to data output from product operations that the filter performs on denominator coefficients.

Looking at the signal flow diagram for the `dfilt.df1t` filter, for example, you see that denominators and numerators are handled separately. When you set `ProductMode` to `SpecifyPrecision`, you can change the `DenProdFracLength` setting manually. Otherwise, for multiplication operations that use the denominator coefficients, the filter sets the fraction length as defined by the `ProductMode` setting.

DenStateFracLength

When you look at the flow diagram for the `dfilt.df1sos` filter object, the states associated with denominator coefficient operations take the fraction length from this property. In combination with the `DenStateWordLength` property, these properties fully specify how the filter interprets the states.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 15 bits, with the `DenStateWordLength` of 16 bits.

DenStateWordLength

When you look at the flow diagram for the `dfilt.df1sos` filter object, the states associated with the denominator coefficient operations take the data format from this property and the `DenStateFracLength` property. In combination, these properties fully specify how the filter interprets the state it uses.

By default, the value is 16 bits, with the `DenStateFracLength` of 15 bits.

FilterInternals

Similar to the FilterInternals pane in FDATool, this property controls whether the filter sets the output word and fraction lengths automatically, and the accumulator word and fraction lengths automatically as well, to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. Setting `FilterInternals` to `SpecifyPrecision` exposes the output and accumulator related properties so you can set your own word and fraction lengths for them. Note that

FilterStructure

Every `dfilt` object has a `FilterStructure` property. This is a read-only property containing a string that declares the structure of the filter object you created.

When you construct filter objects, the `FilterStructure` property value is returned containing one of the strings shown in the following table. Property `FilterStructure` indicates the filter architecture and comes from the constructor you use to create the filter.

After you create a filter object, you cannot change the `FilterStructure` property value. To make filters that use different structures, you construct new filters using the appropriate methods, or use `convert` to switch to a new structure.

Default value

Since this depends on the constructor you use and the constructor includes the filter structure definition, there is no default value. When you try to create a filter without specifying a structure, MATLAB returns an error.

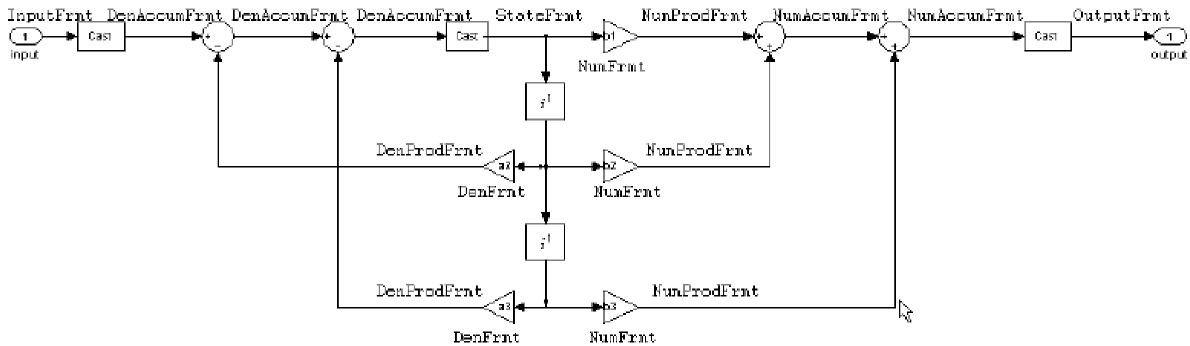
Filter Constructor Name	FilterStructure Property String and Filter Type
`'dfilt.df1'`	Direct form I
`'dfilt.df1sos'`	Direct form I filter implemented using second-order sections
`'dfilt.df1t'`	Direct form I transposed
`'dfilt.df2'`	Direct form II
`'dfilt.df2sos'`	Direct form II filter implemented using second order sections
`'dfilt.df2t'`	Direct form II transposed
`'dfilt.dfasymfir'`	Antisymmetric finite impulse response (FIR). Even and odd forms.
`'dfilt.dffir'`	Direct form FIR
`'dfilt.dffirt'`	Direct form FIR transposed
`'dfilt.latticeallpass'`	Lattice allpass
`'dfilt.latticear'`	Lattice autoregressive (AR)
`'dfilt.latticemamin'`	Lattice moving average (MA) minimum phase
`'dfilt.latticemamax'`	Lattice moving average (MA) maximum phase
`'dfilt.latticearma'`	Lattice ARMA
`'dfilt.dfsymfir'`	Symmetric FIR. Even and odd forms
`'dfilt.scalar'`	Scalar

Filter Structures with Quantizations Shown in Place

To help you understand how and where the quantizations occur in filter structures in this toolbox, the figure below shows the structure for a Direct Form II filter, including the quantizations (fixed-point formats) that compose part of the fixed-point filter. You see that one or more quantization processes, specified by the *format label, accompany each filter element, such as a delay, product, or summation element. The input to or output from each element reflects the result of applying the associated quantization as defined by the word length and fraction length format. Wherever a particular filter element appears in a filter structure, recall the quantization process that accompanies the element as it appears in this figure. Each filter reference page, such as the `dfilt.df2` reference page, includes the signal flow diagram showing the formatting elements that define the quantizations that occur throughout the filter flow.

For example, a product quantization, either numerator or denominator, follows every product (gain) element and a sum quantization, also either numerator or denominator, follows each sum element. The figure shows the `Arithmetic` property value set to `fixed`.

df2 IIR Filter Structure Including the Formatting Objects, with Arithmetic Property Value fixed



When your `df2` filter uses the `Arithmetic` property set to `fixed`, the filter structure contains the formatting features shown in the diagram. The formats included in the structure are fixed-point objects that include properties to set various word and fraction length formats. For example, the `NumFormat` or `DenFormat` in the fixed-point arithmetic filter set the properties for quantizing numerator or denominator coefficients according to word and fraction length settings.

When the leading denominator coefficient a(1) in your filter is not 1, choose it to be a power of two so that a shift replaces the multiply that would otherwise be used.

Fixed-Point Arithmetic Filter Structures

You choose among several filter structures when you create fixed-point filters. You can also specify filters with single or multiple cascaded sections of the same type. Because quantization is a nonlinear process, different fixed-point filter structures produce different results.

To specify the filter structure, you select the appropriate `dfilt.`*`structure`* method to construct your filter. Refer to the function reference information for `dfilt` and `set` for details on setting property values for quantized filters.

The figures in the following subsections of this section serve as aids to help you determine how to enter your filter coefficients for each filter structure. Each subsection contains an example for constructing a filter of the given structure.

Scale factors for the input and output for the filters do not appear in the block diagrams. The default filter structures do not include, nor assume, the scale factors. For filter scaling information, refer to `scale` in the Help system.

About the Filter Structure Diagrams

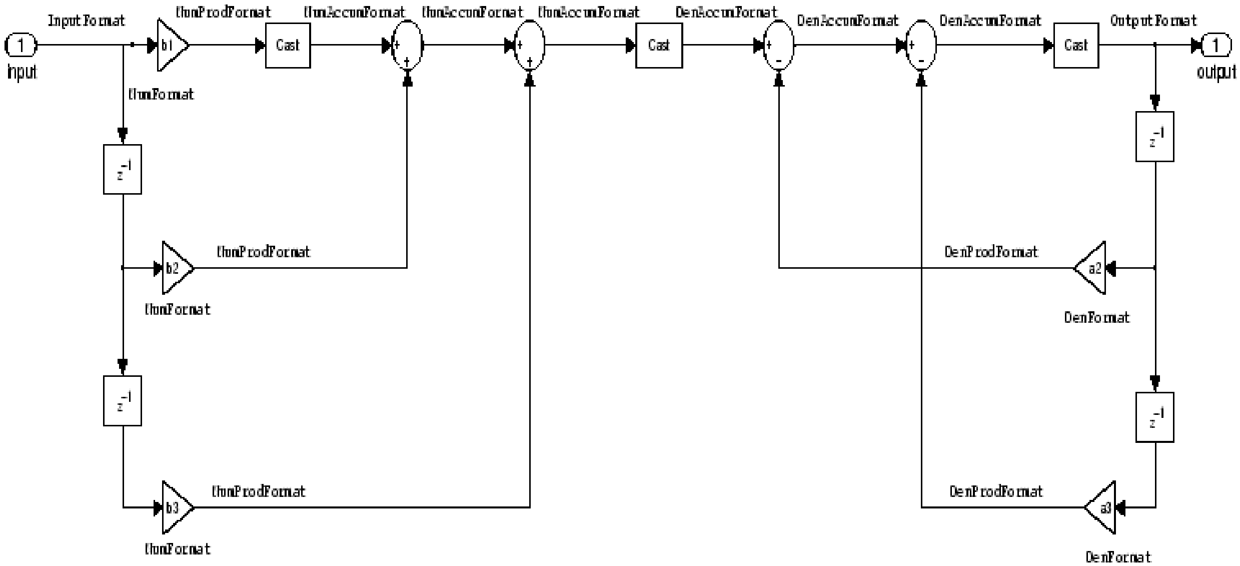
In the diagrams that accompany the following filter structure descriptions, you see the active operators that define the filter, such as sums and gains, and the formatting features that control the processing in the filter. Notice also that the coefficients are labeled in the figure. This tells you the order in which the filter processes the coefficients.

While the meaning of the block elements is straightforward, the labels for the formats that form part of the filter are less clear. Each figure includes text in the form *`labelFormat`* that represents the existence of a formatting feature at that point in the structure. The *`Format`* stands for formatting object and the *`label`* specifies the data that the formatting object affects.

For example, in the `dfilt.df2` filter shown above, the entries `InputFormat` and `OutputFormat` are the formats applied, that is the word length and fraction length, to the filter input and output data. For example, filter properties like `OutputWordLength` and `InputWordLength` specify values that control filter operations at the input and output points in the structure and are represented by the formatting objects `InputFormat` and `OutputFormat` shown in the filter structure diagrams.

Direct Form I Filter Structure

The following figure depicts the *direct form I* filter structure that directly realizes a transfer function with a second-order numerator and denominator. The numerator coefficients are numbered $b(i)$, i =1, 2, 3; the denominator coefficients are numbered $a(i)$, i = 1, 2, 3; and the states (used for initial and final state values in filtering) are labeled $z(i)$. In the figure, the `Arithmetic` property is set to `fixed`.



Example — Specifying a Direct Form I Filter

You can specify a second-order direct form I structure for a quantized filter `hq` with the following code.

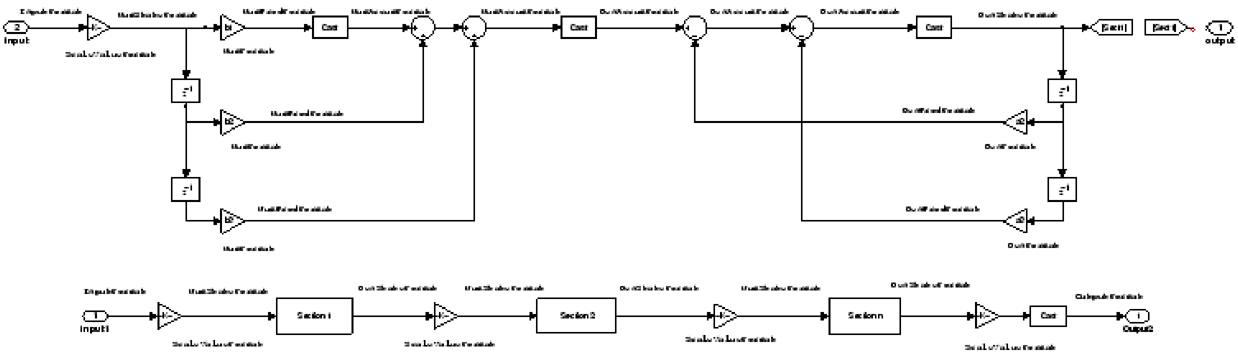
```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df1(b,a);
```

To create the fixed-point filter, set the `Arithmetic` property to `fixed` as shown here.

```
set(hq,'arithmetic','fixed');
```

Direct Form I Filter Structure With Second-Order Sections

The following figure depicts a *direct form I* filter structure that directly realizes a transfer function with a second-order numerator and denominator and second-order sections. The numerator coefficients are numbered $b(i)$, i =1, 2, 3; the denominator coefficients are numbered $a(i)$, i = 1, 2, 3; and the states (used for initial and final state values in filtering) are labeled $z(i)$. In the figure, the `Arithmetic` property is set to `fixed` to place the filter in fixed-point mode.



Example — Specifying a Direct Form I Filter with Second-Order Sections

You can specify an eighth-order direct form I structure for a quantized filter `hq` with the following code.

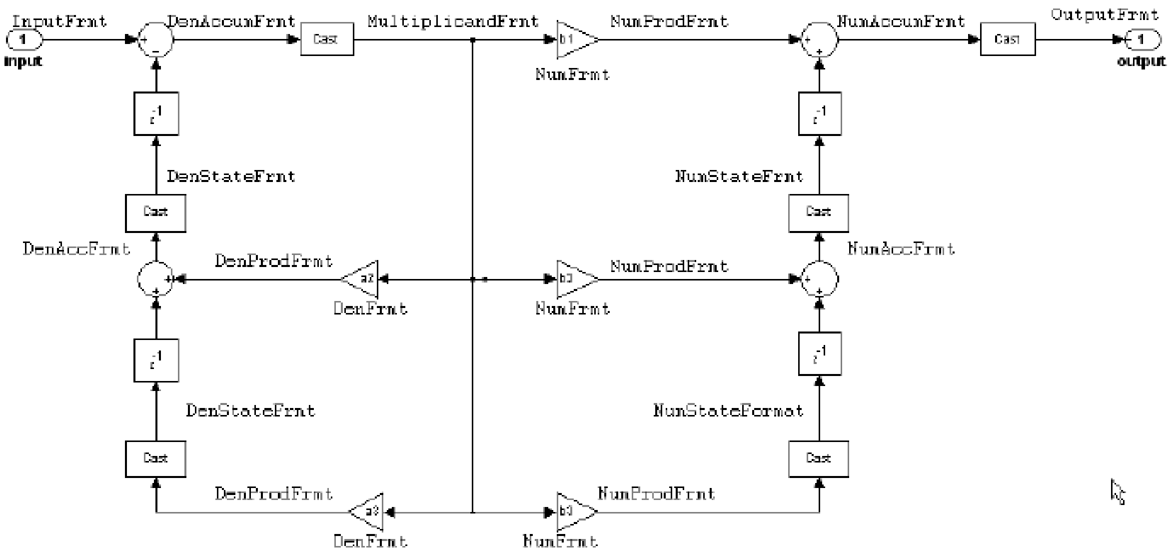
```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df1sos(b,a);
```

To create the fixed-point filter, set the `Arithmetic` property to `fixed`, as shown here.

```
set(hq,'arithmetic','fixed');
```

Direct Form I Transposed Filter Structure

The next signal flow diagram depicts a *direct form I transposed* filter structure that directly realizes a transfer function with a second-order numerator and denominator. The numerator coefficients are $b(i)$, i = 1, 2, 3; the denominator coefficients are $a(i)$, i = 1, 2, 3; and the states (used for initial and final state values in filtering) are labeled $z(i)$. With the `Arithmetic` property value set to `fixed`, the figure shows the filter with the properties indicated.



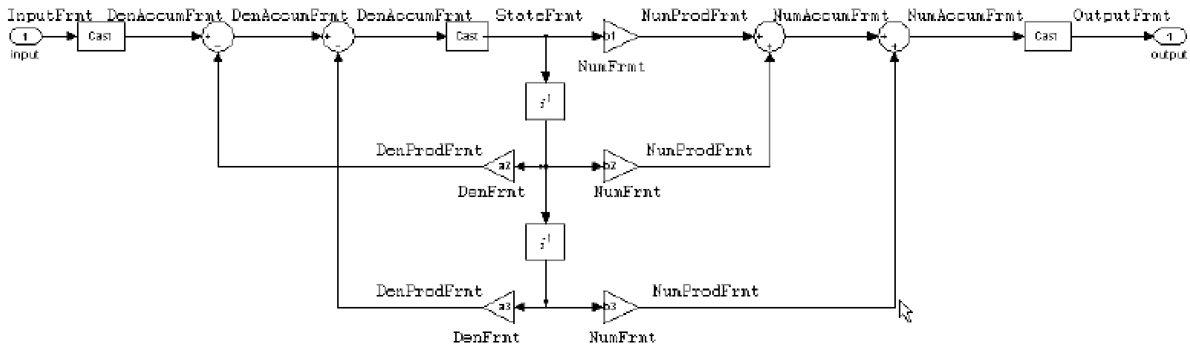
Example — Specifying a Direct Form I Transposed Filter

You can specify a second-order direct form I transposed filter structure for a quantized filter `hq` with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df1t(b,a);
set(hq,'arithmetic','fixed');
```

Direct Form II Filter Structure

The following graphic depicts a *direct form II* filter structure that directly realizes a transfer function with a second-order numerator and denominator. In the figure, the `Arithmetic` property value is `fixed`. Numerator coefficients are named $b(i)$; denominator coefficients are named $a(i)$, $i = 1, 2, 3$; and the states (used for initial and final state values in filtering) are named $z(i)$.



Use the method `dfilt.df2` to construct a quantized filter whose `FilterStructure` property is `Direct-Form II`.

Example — Specifying a Direct Form II Filter

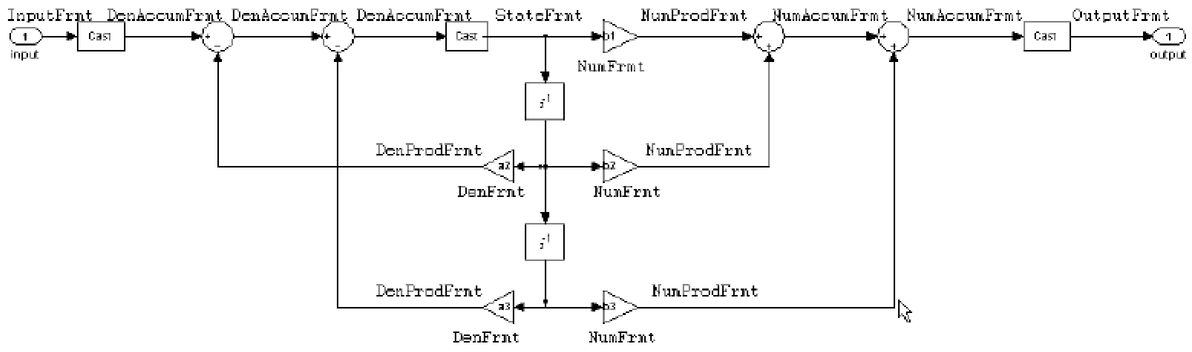
You can specify a second-order direct form II filter structure for a quantized filter `hq` with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df2(b,a);
hq.arithmetic = 'fixed'
```

To convert your initial double-precision filter `hq` to a quantized or fixed-point filter, set the `Arithmetic` property to `fixed`, as shown.

Direct Form II Filter Structure With Second-Order Sections

The following figure depicts *direct form II* filter structure using second-order sections that directly realizes a transfer function with a second-order numerator and denominator sections. In the figure, the `Arithmetic` property value is `fixed`. Numerator coefficients are labeled $b(i)$; denominator coefficients are labeled $a(i)$, $i = 1, 2, 3$; and the states (used for initial and final state values in filtering) are labeled $z(i)$.



Use the method `dfilt.df2sos` to construct a quantized filter whose `FilterStructure` property is `Direct-Form II`.

Example — Specifying a Direct Form II Filter with Second-Order Sections

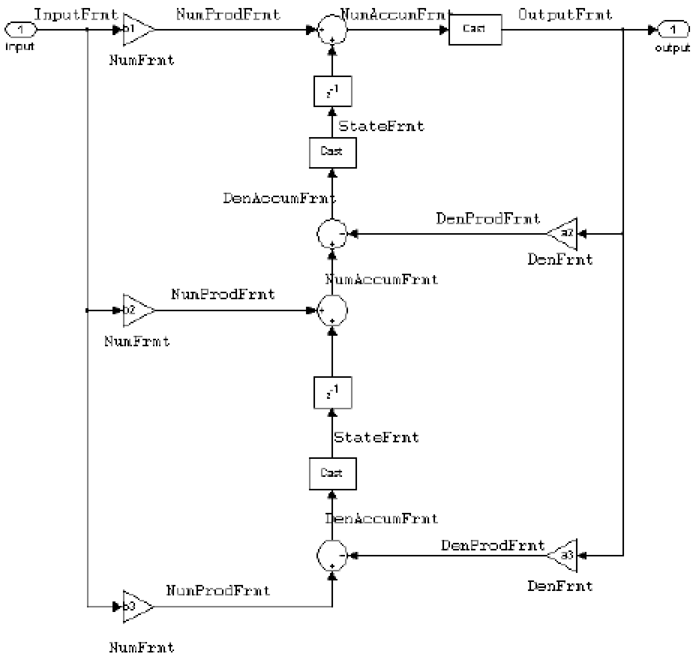
You can specify a tenth-order direct form II filter structure that uses second-order sections for a quantized filter `hq` with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df2sos(b,a);
hq.arithmetic = 'fixed'
```

To convert your prototype double-precision filter `hq` to a fixed-point filter, set the `Arithmetic` property to `fixed`, as shown.

Direct Form II Transposed Filter Structure

The following figure depicts the *direct form II transposed* filter structure that directly realizes transfer functions with a second-order numerator and denominator. The numerator coefficients are labeled $b(i)$, the denominator coefficients are labeled $a(i)$, $i = 1$, 2, 3, and the states (used for initial and final state values in filtering) are labeled $z(i)$. In the first figure, the `Arithmetic` property value is `fixed`.



Use the constructor `dfilt.df2t` to specify the value of the `FilterStructure` property for a filter with this structure that you can convert to fixed-point filtering.

Example — Specifying a Direct Form II Transposed Filter

Specifying or constructing a second-order direct form II transposed filter for a fixed-point filter `hq` starts with the following code to define the coefficients and construct the filter.

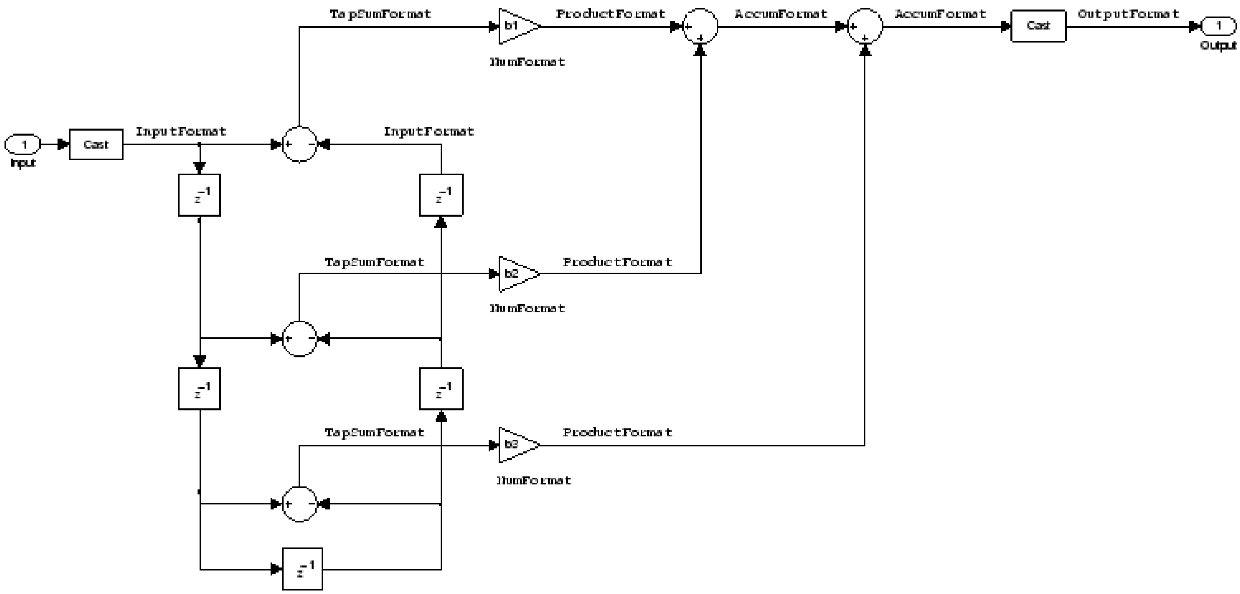
```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df2t(b,a);
```

Now create the fixed-point filtering version of the filter from `hd`, which is floating point.

```
hq = set(hd,'arithmetic','fixed');
```

Direct Form Antisymmetric FIR Filter Structure (Any Order)

The following figure depicts a *direct form antisymmetric FIR* filter structure that directly realizes a second-order antisymmetric FIR filter. The filter coefficients are labeled *b(i)*, and the initial and final state values in filtering are labeled *z*(*i*). This structure reflects the `Arithmetic` property set to `fixed`.



Use the method `dfilt.dfasymfir` to construct the filter, and then set the `Arithmetic` property to `fixed` to convert to a fixed-point filter with this structure.

Example — Specifying an Odd-Order Direct Form Antisymmetric FIR Filter

Specify a fifth-order direct form antisymmetric FIR filter structure for a fixed-point filter `hq` with the following code.

```
b = [-0.008 0.06 -0.44 0.44 -0.06 0.008];
hq = dfilt.dfasymfir(b);
set(hq,'arithmetic','fixed')

hq

hq =

         FilterStructure: 'Direct-Form Antisymmetric FIR'
              Arithmetic: 'fixed'
               Numerator: [-0.0080 0.0600 -0.4400 0.4400 -0.0600 0.0080]
        PersistentMemory: false
                  States: [1x1 fi object]

         CoeffWordLength: 16
          CoeffAutoScale: true
```

```
             Signed: true

    InputWordLength: 16
    InputFracLength: 15

   OutputWordLength: 16
         OutputMode: 'AvoidOverflow'

         TapSumMode: 'KeepMSB'
   TapSumWordLength: 17

        ProductMode: 'FullPrecision'

    AccumWordLength: 40

      CastBeforeSum: true
          RoundMode: 'convergent'
       OverflowMode: 'wrap'

    InheritSettings: false
```

Example — Specifying an Even-Order Direct Form Antisymmetric FIR Filter

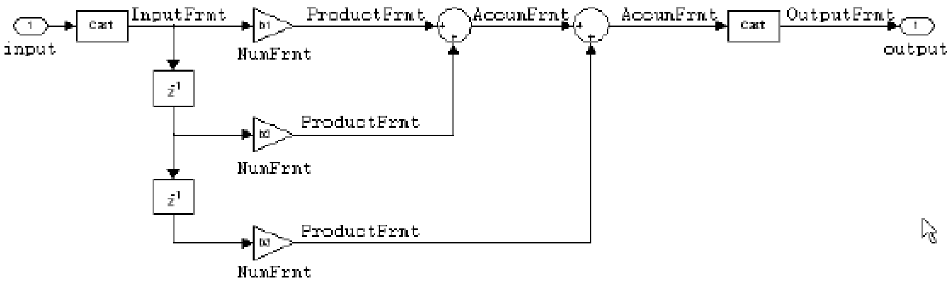
You can specify a fourth-order direct form antisymmetric FIR filter structure for a fixed-point filter hq with the following code.

```
b = [-0.01 0.1 0.0 -0.1 0.01];
hq = dfilt.dfasymfir(b);
hq.arithmetic='fixed'

hq =

         FilterStructure: 'Direct-Form Antisymmetric FIR'
              Arithmetic: 'fixed'
               Numerator: [-0.0100 0.1000 0 -0.1000 0.0100]
        PersistentMemory: false
                  States: [1x1 fi object]

         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

              TapSumMode: 'KeepMSB'
        TapSumWordLength: 17
```

```
       ProductMode: 'FullPrecision'

    AccumWordLength: 40

      CastBeforeSum: true
          RoundMode: 'convergent'
       OverflowMode: 'wrap'

    InheritSettings: false
```

Direct Form Finite Impulse Response (FIR) Filter Structure

In the next figure, you see the signal flow graph for a *direct form finite impulse response (FIR)* filter structure that directly realizes a second-order FIR filter. The filter coefficients are $b(i)$, $i = 1, 2, 3$, and the states (used for initial and final state values in filtering) are $z(i)$. To generate the figure, set the `Arithmetic` property to `fixed` after you create your prototype filter in double-precision arithmetic.



Use the `dfilt.dffir` method to generate a filter that uses this structure.

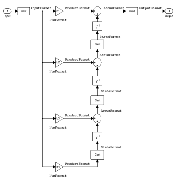
Example — Specifying a Direct Form FIR Filter

You can specify a second-order direct form FIR filter structure for a fixed-point filter `hq` with the following code.

```
b = [0.05 0.9 0.05];
hd = dfilt.dffir(b);
hq = set(hd,'arithmetic','fixed');
```

Direct Form FIR Transposed Filter Structure

This figure uses the filter coefficients labeled *b*(*i*), *i* = 1, 2, 3, and states (used for initial and final state values in filtering) are labeled *z*(*i*). These depict a *direct form finite impulse response (FIR) transposed* filter structure that directly realizes a second-order FIR filter.



With the `Arithmetic` property set to `fixed`, your filter matches the figure. Using the method `dfilt.dffirt` returns a double-precision filter that you convert to a fixed-point filter.

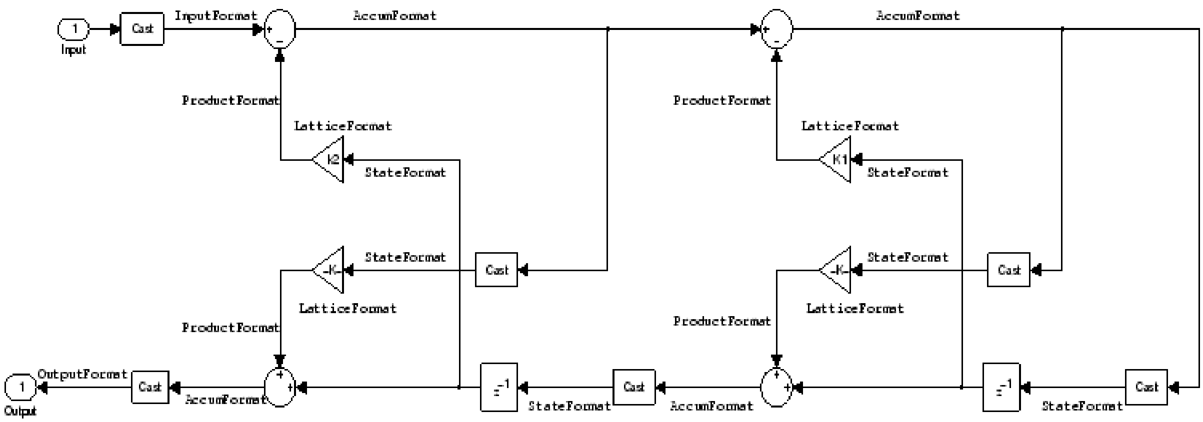
Example — Specifying a Direct Form FIR Transposed Filter

You can specify a second-order direct form FIR transposed filter structure for a fixed-point filter `hq` with the following code.

```
b = [0.05 0.9 0.05];
hd=dfilt.dffirt(b);
hq = copy(hd);
hq.arithmetic = 'fixed';
```

Lattice Allpass Filter Structure

The following figure depicts the *lattice allpass* filter structure. The pictured structure directly realizes third-order lattice allpass filters using fixed-point arithmetic. The filter reflection coefficients are labeled *k1*(*i*), *i* = 1, 2, 3. The states (used for initial and final state values in filtering) are labeled *z*(*i*).



To create a quantized filter that uses the lattice allpass structure shown in the figure, use the `dfilt.latticeallpass` method and set the `Arithmetic` property to `fixed`.

Example — Specifying a Lattice Allpass Filter

You can create a third-order lattice allpass filter structure for a quantized filter `hq` with the following code.

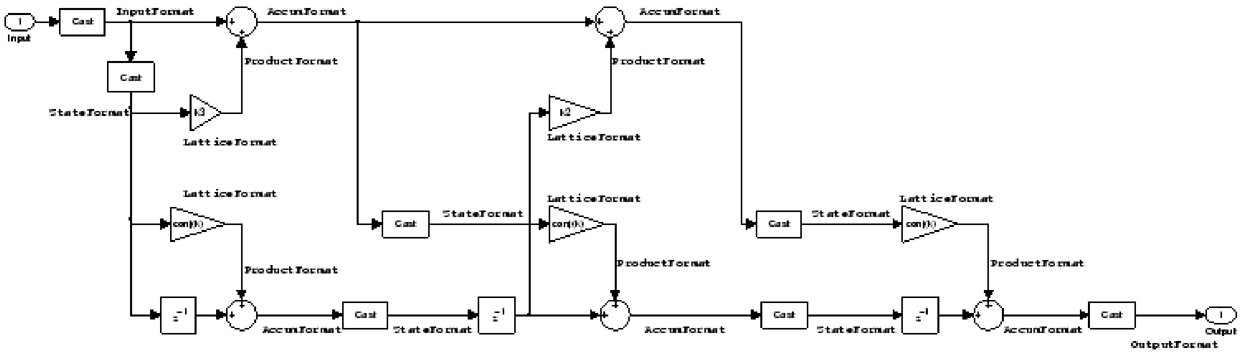
```
k = [.66 .7 .44];
hd=dfilt.latticeallpass(k);
set(hq,'arithmetic','fixed');
```

Lattice Moving Average Maximum Phase Filter Structure

In the next figure you see a *lattice moving average maximum phase* filter structure. This signal flow diagram directly realizes a third-order lattice moving average (MA) filter with the following phase form depending on the initial transfer function:

- When you start with a minimum phase transfer function, the upper branch of the resulting lattice structure returns a minimum phase filter. The lower branch returns a maximum phase filter.
- When your transfer function is neither minimum phase nor maximum phase, the lattice moving average maximum phase structure will not be maximum phase.
- When you start with a maximum phase filter, the resulting lattice filter is maximum phase also.

The filter reflection coefficients are labeled $k(i)$, $i = 1, 2, 3$. The states (used for initial and final state values in filtering) are labeled $z(i)$. In the figure, we set the `Arithmetic` property to `fixed` to reveal the fixed-point arithmetic format features that control such options as word length and fraction length.



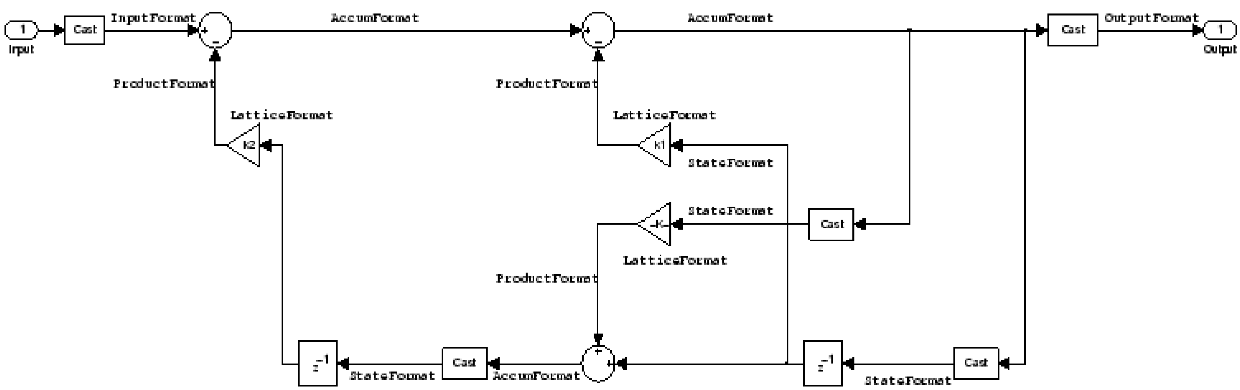
Example — Constructing a Lattice Moving Average Maximum Phase Filter

Constructing a fourth-order lattice MA maximum phase filter structure for a quantized filter `hq` begins with the following code.

```
k = [.66 .7 .44 .33];
hd=dfilt.latticemamax(k);
```

Lattice Autoregressive (AR) Filter Structure

The method `dfilt.latticear` directly realizes lattice autoregressive filters in the toolbox. The following figure depicts the third-order *lattice autoregressive (AR)* filter structure — with the `Arithmetic` property equal to `fixed`. The filter reflection coefficients are labeled $k(i)$, $i = 1, 2, 3$, and the states (used for initial and final state values in filtering) are labeled $z(i)$.



Example — Specifying a Lattice AR Filter

You can specify a third-order lattice AR filter structure for a quantized filter hq with the following code.

```
k = [.66 .7 .44];
hd=dfilt.latticear(k);
hq.arithmetic = 'custom';
```

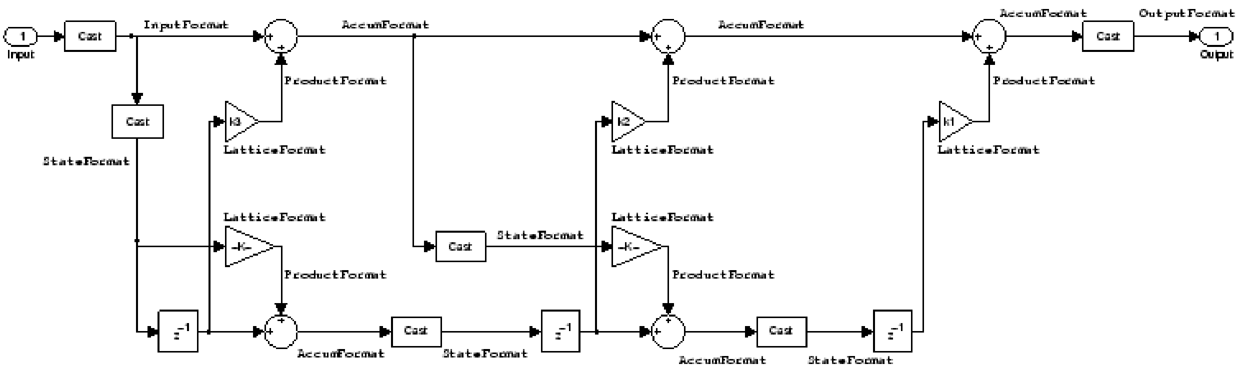
Lattice Moving Average (MA) Filter Structure for Minimum Phase

The following figures depict *lattice moving average (MA)* filter structures that directly realize third-order lattice MA filters for minimum phase. The filter reflection coefficients are labeled $k(i)$, (i). = 1, 2, 3, and the states (used for initial and final state values in filtering) are labeled $z(i)$. Setting the `Arithmetic` property of the filter to `fixed` results in a fixed-point filter that matches the figure.

This signal flow diagram directly realizes a third-order lattice moving average (MA) filter with the following phase form depending on the initial transfer function:

- When you start with a minimum phase transfer function, the upper branch of the resulting lattice structure returns a minimum phase filter. The lower branch returns a minimum phase filter.
- When your transfer function is neither minimum phase nor maximum phase, the lattice moving average minimum phase structure will not be minimum phase.
- When you start with a minimum phase filter, the resulting lattice filter is minimum phase also.

The filter reflection coefficients are labeled $k((i).)$, i = 1, 2, 3. The states (used for initial and final state values in filtering) are labeled $z((i).)$. This figure shows the filter structure when the`Arithmetic` property is set to `fixed` to reveal the fixed-point arithmetic format features that control such options as word length and fraction length.



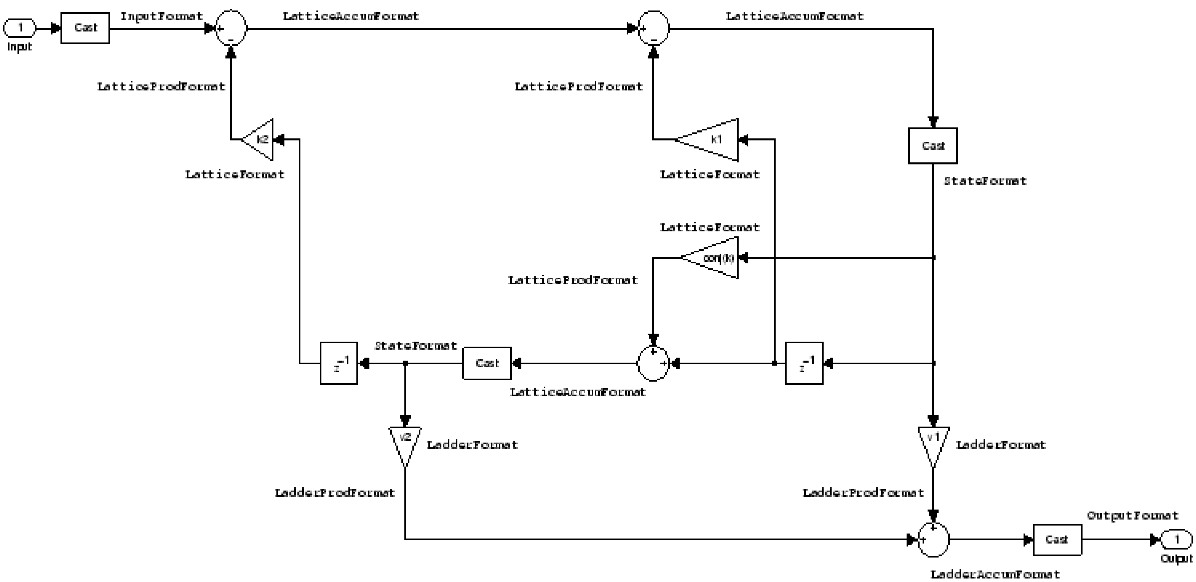
Example — Specifying a Minimum Phase Lattice MA Filter

You can specify a third-order lattice MA filter structure for minimum phase applications using variations of the following code.

```
k = [.66 .7 .44];
hd=dfilt.latticemamin(k);
set(hq,'arithmetic','fixed');
```

Lattice Autoregressive Moving Average (ARMA) Filter Structure

The figure below depicts a *lattice autoregressive moving average (ARMA)* filter structure that directly realizes a fourth-order lattice ARMA filter. The filter reflection coefficients are labeled $k(i)$, (i). = 1, ..., 4; the ladder coefficients are labeled $v(i)$, (i). = 1, 2, 3; and the states (used for initial and final state values in filtering) are labeled $z(i)$.



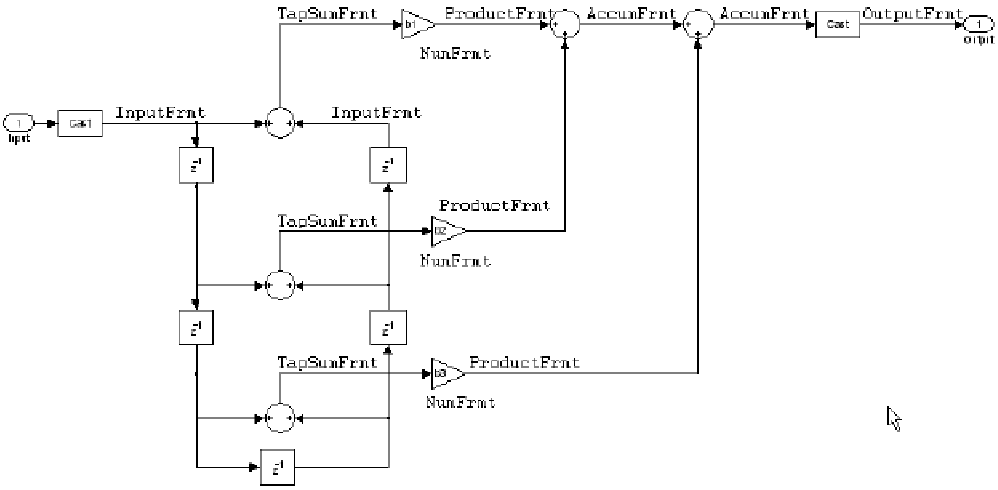
Example — Specifying an Lattice ARMA Filter

The following code specifies a fourth-order lattice ARMA filter structure for a quantized filter `hq`, starting from `hd`, a floating-point version of the filter.

```
k = [.66 .7 .44 .66];
v = [1 0 0];
hd=dfilt.latticearma(k,v);
hq.arithmetic = 'fixed';
```

Direct Form Symmetric FIR Filter Structure (Any Order)

Shown in the next figure, you see signal flow that depicts a *direct form symmetric FIR* filter structure that directly realizes a fifth-order direct form symmetric FIR filter. Filter coefficients are labeled $b(i)$, $i = 1, ..., n$, and states (used for initial and final state values in filtering) are labeled $z(i)$. Showing the filter structure used when you select `fixed` for the `Arithmetic` property value, the first figure details the properties in the filter object.



Example — Specifying an Odd-Order Direct Form Symmetric FIR Filter

By using the following code in MATLAB, you can specify a fifth-order direct form symmetric FIR filter for a fixed-point filter `hq`:

```
b = [-0.008 0.06 0.44 0.44 0.06 -0.008];
hd=dfilt.dfsymfir(b);
set(hq,'arithmetic','fixed');
```

Assigning Filter Coefficients

The syntax you use to assign filter coefficients for your floating-point or fixed-point filter depends on the structure you select for your filter.

Converting Filters Between Representations

Filter conversion functions in this toolbox and in Signal Processing Toolbox software let you convert filter transfer functions to other filter forms, and from other filter forms to transfer function form. Relevant conversion functions include the following functions.

Conversion Function	Description
`ca2tf`	Converts from a coupled allpass filter to a transfer function.
`cl2tf`	Converts from a lattice coupled allpass filter to a transfer function.

Conversion Function	Description
convert	Convert a discrete-time filter from one filter structure to another.
sos	Converts quantized filters to create second-order sections. We recommend this method for converting quantized filters to second-order sections.
tf2ca	Converts from a transfer function to a coupled allpass filter.
tf2cl	Converts from a transfer function to a lattice coupled allpass filter.
tf2latc	Converts from a transfer function to a lattice filter.
tf2sos	Converts from a transfer function to a second-order section form.
tf2ss	Converts from a transfer function to state-space form.
tf2zp	Converts from a rational transfer function to its factored (single section) form (zero-pole-gain form).
zp2sos	Converts a zero-pole-gain form to a second-order section form.
zp2ss	Conversion of zero-pole-gain form to a state-space form.
zp2tf	Conversion of zero-pole-gain form to transfer functions of multiple order sections.

Note that these conversion routines do not apply to `dfilt` objects.

The function `convert` is a special case — when you use convert to change the filter structure of a fixed-point filter, you lose all of the filter states and settings. Your new filter has default values for all properties, and it in not fixed-point.

To demonstrate the changes that occur, convert a fixed-point direct form I transposed filter to direct form II structure.

```
hd=dfilt.df1t

hd =

         FilterStructure: 'Direct-Form I Transposed'
              Arithmetic: 'double'
               Numerator: 1
             Denominator: 1
```

```
        PersistentMemory: false
                  States: Numerator:  [0x0 double]
                          Denominator:[0x0 double]


hd.arithmetic='fixed'
hd =

         FilterStructure: 'Direct-Form I Transposed'
              Arithmetic: 'fixed'
               Numerator: 1
             Denominator: 1
        PersistentMemory: false
                  States: Numerator:  [0x0 fi]
                          Denominator:[0x0 fi]

convert(hd,'df2')

Warning: Using reference filter for structure conversion.
Fixed-point attributes will not be converted.

ans =

         FilterStructure: 'Direct-Form II'
              Arithmetic: 'double'
               Numerator: 1
             Denominator: 1
        PersistentMemory: false
                  States: [0x1 double]
```

You can specify a filter with L sections of arbitrary order by

1. Factoring your entire transfer function with `tf2zp`. This converts your transfer function to zero-pole-gain form.
2. Using `zp2tf` to compose the transfer function for each section from the selected first-order factors obtained in step 1.

Note You are not required to normalize the leading coefficients of each section's denominator polynomial when you specify second-order sections, though `tf2sos` does.

Gain

dfilt.scalar filters have a gain value stored in the gain property. By default the gain value is one — the filter acts as a wire.

InputFracLength

InputFracLength defines the fraction length assigned to the input data for your filter. Used in tandem with InputWordLength, the pair defines the data format for input data you provide for filtering.

As with all fraction length properties in dfilt objects, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, in this case InputWordLength, as well.

InputWordLength

Specifies the number of bits your filter uses to represent your input data. Your word length option is limited by the arithmetic you choose — up to 32 bits for double, float, and fixed. Setting Arithmetic to single (single-precision floating-point) limits word length to 16 bits. The default value is 16 bits.

Ladder

Included as a property in dfilt.latticearma filter objects, Ladder contains the denominator coefficients that form an IIR lattice filter object. For instance, the following code creates a high pass filter object that uses the lattice ARMA structure.

```
[b,a]=cheby1(5,.5,.5,'high')

b =

    0.0282   -0.1409    0.2817   -0.2817    0.1409   -0.0282


a =

    1.0000    0.9437    1.4400    0.9629    0.5301    0.1620

hd=dfilt.latticearma(b,a)

hd =

         FilterStructure: [1x44 char]
```

```
            Arithmetic: 'double'
               Lattice: [1x6 double]
                Ladder: [1 0.9437 1.4400 0.9629 0.5301 0.1620]
      PersistentMemory: false
                States: [6x1 double]

hd.arithmetic='fixed'

hd =

       FilterStructure: [1x44 char]
            Arithmetic: 'fixed'
               Lattice: [1x6 double]
                Ladder: [1 0.9437 1.4400 0.9629 0.5301 0.1620]
      PersistentMemory: false
                States: [1x1 embedded.fi]

       CoeffWordLength: 16
        CoeffAutoScale: true
                Signed: true

       InputWordLength: 16
       InputFracLength: 15

      OutputWordLength: 16
            OutputMode: 'AvoidOverflow'

       StateWordLength: 16
       StateFracLength: 15

           ProductMode: 'FullPrecision'

       AccumWordLength: 40
         CastBeforeSum: true

             RoundMode: 'convergent'
          OverflowMode: 'wrap'
```

LadderAccumFracLength

Autoregressive, moving average lattice filter objects (`lattticearma`) use ladder coefficients to define the filter. In combination with `LadderFracLength` and `CoeffWordLength`, these three properties specify or reflect how the accumulator outputs data stored there. As with all fraction length properties, `LadderAccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers. The default value is 29 bits.

LadderFracLength

To let you control the way your `latticearma` filter interprets the denominator coefficients, `LadderFracLength` sets the fraction length applied to the ladder coefficients for your filter. The default value is 14 bits.

As with all fraction length properties, `LadderFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers.

Lattice

When you create a lattice-based IIR filter, your numerator coefficients (from your IIR prototype filter or the default `dfilt` lattice filter function) get stored in the `Lattice` property of the `dfilt` object. The properties `CoeffWordLength` and `LatticeFracLength` define the data format the object uses to represent the lattice coefficients. By default, lattice coefficients are in double-precision format.

LatticeAccumFracLength

Lattice filter objects (`latticeallpass`, `latticearma`, `latticemamax`, and `latticemamin`) use lattice coefficients to define the filter. In combination with `LatticeFracLength` and `CoeffWordLength`, these three properties specify how the accumulator outputs lattice coefficient-related data stored there. As with all fraction length properties, `LatticeAccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers. By default, the property is set to 31 bits.

LatticeFracLength

To let you control the way your filter interprets the denominator coefficients, `LatticeFracLength` sets the fraction length applied to the lattice coefficients for your lattice filter. When you create the default lattice filter, `LatticeFracLength` is 16 bits.

As with all fraction length properties, `LatticeFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers.

MultiplicandFracLength

Each input data element for a multiply operation has both word length and fraction length to define its representation. `MultiplicandFracLength` sets the fraction length to use when the filter object performs any multiply operation during filtering. For default filters, this is set to 15 bits.

As with all word and fraction length properties, MultiplicandFracLength can be any integer, including integers larger than CoeffWordLength, and positive or negative integers.

MultiplicandWordLength

Each input data element for a multiply operation has both word length and fraction length to define its representation. MultiplicandWordLength sets the word length to use when the filter performs any multiply operation during filtering. For default filters, this is set to 16 bits. Only the df1t and df1tsos filter objects include the MultiplicandFracLength property.

Only the df1t and df1tsos filter objects include the MultiplicandWordLength property.

NumAccumFracLength

Filter structures df1, df1t, df2, and df2t that use fixed arithmetic have this property that defines the fraction length applied to numerator coefficients in output from the accumulator. In combination with AccumWordLength, the NumAccumFracLength property fully specifies how the accumulator outputs numerator-related data stored there.

As with all fraction length properties, NumAccumFracLength can be any integer, including integers larger than AccumWordLength, and positive or negative integers. 30 bits is the default value when you create the filter object. To be able to change the value for this property, set FilterInternals for the filter to SpecifyPrecision.

Numerator

The numerator coefficients for your filter, taken from the prototype you start with or from the default filter, are stored in this property. Generally this is a 1-by-N array of data in double format, where N is the length of the filter.

All of the filter objects include Numerator, except the lattice-based and second-order section filters, such as dfilt.latticema and dfilt.df1tsos.

NumFracLength

Property NumFracLength contains the value that specifies the fraction length for the numerator coefficients for your filter. NumFracLength specifies the fraction length used to interpret the numerator coefficients. Used in combination with `CoeffWordLength`, these two properties define the interpretation of the coefficients stored in the vector that contains the numerator coefficients.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 15 bits, with the `CoeffWordLength` of 16 bits.

NumProdFracLength

A property of all of the direct form IIR `dfilt` objects, except the ones that implement second-order sections, `NumProdFracLength` specifies the fraction length applied to data output from product operations the filter performs on numerator coefficients.

Looking at the signal flow diagram for the `dfilt.df1t` filter, for example, you see that denominators and numerators are handled separately. When you set `ProductMode` to `SpecifyPrecision`, you can change the `NumProdFracLength` setting manually. Otherwise, for multiplication operations that use the numerator coefficients, the filter sets the word length as defined by the `ProductMode` setting.

NumStateFracLength

All the variants of the direct form I structure include the property `NumStateFracLength` to store the fraction length applied to the numerator states for your filter object. By default, this property has the value 15 bits, with the `CoeffWordLength` of 16 bits, which you can change after you create the filter object.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well.

NumStateWordLength

When you look at the flow diagram for the `df1sos` filter object, the states associated with the numerator coefficient operations take the data format from this property and the `NumStateFracLength` property. In combination, these properties fully specify how the filter interprets the state it uses.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 16 bits, with the `NumStateFracLength` of 11 bits.

OutputFracLength

To define the output from your filter object, you need both the word and fraction lengths. `OutputFracLength` determines the fraction length applied to interpret the output data. Combining this with `OutputWordLength` fully specifies the format of the output.

Your fraction length can be any negative or positive integer, or zero. In addition, the fraction length you specify can be larger than the associated word length. Generally, the default value is 11 bits.

OutputMode

Sets the mode the filter uses to scale the filtered (output) data. You have the following choices:

- `AvoidOverflow` — directs the filter to set the property that controls the output data fraction length to avoid causing the data to overflow. In a `df2` filter, this would be the `OutputFracLength` property.
- `BestPrecision` — directs the filter to set the property that controls the output data fraction length to maximize the precision in the output data. For `df1t` filters, this is the `OutputFracLength` property. When you change the word length (`OutputWordLength`), the filter adjusts the fraction length to maintain the best precision for the new word size.
- `SpecifyPrecision` — lets you set the fraction length used by the filtered data. When you select this choice, you can set the output fraction length using the `OutputFracLength` property to define the output precision.

All filters include this property except the direct form I filter which takes the output format from the filter states.

Here is an example that changes the mode setting to `bestprecision`, and then adjusts the word length for the output.

```
hd=dfilt.df2

hd =

         FilterStructure: 'Direct-Form II'
              Arithmetic: 'double'
               Numerator: 1
             Denominator: 1
        PersistentMemory: false
                  States: [0x1 double]
```

```
hd.arithmetic='fixed'

hd =

         FilterStructure: 'Direct-Form II'
              Arithmetic: 'fixed'
               Numerator: 1
             Denominator: 1
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

         StateWordLength: 16
         StateFracLength: 15

             ProductMode: 'FullPrecision'

         AccumWordLength: 40
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'

get(hd)
        PersistentMemory: false
FilterStructure: 'Direct-Form II'
                  States: [1x1 embedded.fi]
               Numerator: 1
             Denominator: 1
              Arithmetic: 'fixed'
         CoeffWordLength: 16
          CoeffAutoScale: 1
                  Signed: 1
               RoundMode: 'convergent'
```

```
             OverflowMode: 'wrap'
          InputWordLength: 16
          InputFracLength: 15
         OutputWordLength: 16
               OutputMode: 'AvoidOverflow'
              ProductMode: 'FullPrecision'
          StateWordLength: 16
          StateFracLength: 15
            NumFracLength: 14
            DenFracLength: 14
         OutputFracLength: 13
        ProductWordLength: 32
        NumProdFracLength: 29
        DenProdFracLength: 29
          AccumWordLength: 40
       NumAccumFracLength: 29
       DenAccumFracLength: 29
            CastBeforeSum: 1

hd.outputMode='bestprecision'

hd =

          FilterStructure: 'Direct-Form II'
               Arithmetic: 'fixed'
                Numerator: 1
              Denominator: 1
         PersistentMemory: false
                   States: [1x1 embedded.fi]

          CoeffWordLength: 16
           CoeffAutoScale: true
                   Signed: true

          InputWordLength: 16
          InputFracLength: 15

         OutputWordLength: 16
               OutputMode: 'BestPrecision'

          StateWordLength: 16
          StateFracLength: 15

              ProductMode: 'FullPrecision'
```

```
          AccumWordLength: 40
            CastBeforeSum: true

                RoundMode: 'convergent'
             OverflowMode: 'wrap'

hd.outputWordLength=8;

get(hd)
        PersistentMemory: false
         FilterStructure: 'Direct-Form II'
                  States: [1x1 embedded.fi]
               Numerator: 1
             Denominator: 1
              Arithmetic: 'fixed'
         CoeffWordLength: 16
          CoeffAutoScale: 1
                  Signed: 1
               RoundMode: 'convergent'
            OverflowMode: 'wrap'
         InputWordLength: 16
         InputFracLength: 15
        OutputWordLength: 8
              OutputMode: 'BestPrecision'
             ProductMode: 'FullPrecision'
         StateWordLength: 16
         StateFracLength: 15
           NumFracLength: 14
           DenFracLength: 14
        OutputFracLength: 5
       ProductWordLength: 32
       NumProdFracLength: 29
       DenProdFracLength: 29
         AccumWordLength: 40
      NumAccumFracLength: 29
      DenAccumFracLength: 29
           CastBeforeSum: 1
```

Changing the `OutputWordLength` to `8` bits caused the filter to change the `OutputFracLength` to `5` bits to keep the best precision for the output data.

OutputWordLength

Use the property `OutputWordLength` to set the word length used by the output from your filter. Set this property to a value that matches your intended hardware. For example, some digital signal processors use 32-bit output so you would set `OutputWordLength` to `32`.

```
[b,a] = butter(6,.5);
hd=dfilt.df1t(b,a);

set(hd,'arithmetic','fixed')

hd

hd =

         FilterStructure: 'Direct-Form I Transposed'
              Arithmetic: 'fixed'
               Numerator: [1x7 double]
             Denominator: [1 0 0.7777 0 0.1142 0 0.0018]
     PersistentMemory: false
                  States: Numerator:  [6x1 fi]
                          Denominator:[6x1 fi]


         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

  MultiplicandWordLength: 16
  MultiplicandFracLength: 15

         StateWordLength: 16
          StateAutoScale: true

             ProductMode: 'FullPrecision'

         AccumWordLength: 40
```

```
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'

hd.outputwordLength=32

hd =

         FilterStructure: 'Direct-Form I Transposed'
              Arithmetic: 'fixed'
               Numerator: [1x7 double]
             Denominator: [1 0 0.7777 0 0.1142 0 0.0018]
        PersistentMemory: false
                  States: Numerator:  [6x1 fi]
                          Denominator:[6x1 fi]


         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 32
              OutputMode: 'AvoidOverflow'

  MultiplicandWordLength: 16
  MultiplicandFracLength: 15

         StateWordLength: 16
          StateAutoScale: true

             ProductMode: 'FullPrecision'

         AccumWordLength: 40
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'
```

When you create a filter object, this property starts with the value 16.

OverflowMode

The `OverflowMode` property is specified as one of the following two strings indicating how to respond to overflows in fixed-point arithmetic:

- `'saturate'` — saturate overflows.

  When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the applicable word length and fraction length properties), these values are quantized to the value of either the largest or smallest representable value, depending on which is closest. `saturate` is the default value for `OverflowMode`.
- `'wrap'` — wrap all overflows to the range of representable values.

  When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the data format properties), these values are wrapped back into that range using modular arithmetic relative to the smallest representable number. You can learn more about modular arithmetic in Fixed-Point Designer documentation.

These rules apply to the `OverflowMode` property.

- Applies to the accumulator and output data only.
- Does not apply to coefficients or input data. These always saturate the results.
- Does not apply to products. Products maintain full precision at all times. Your filters do not lose precision in the products.

Note Numbers in floating-point filters that extend beyond the dynamic range overflow to ±`inf`.

ProductFracLength

After you set `ProductMode` for a fixed-point filter to `SpecifyPrecision`, this property becomes available for you to change. `ProductFracLength` sets the fraction length the filter uses for the results of multiplication operations. Only the FIR filters such as asymmetric FIRs or lattice autoregressive filters include this dynamic property.

Your fraction length can be any negative or positive integer, or zero. In addition, the fraction length you specify can be larger than the associated word length. Generally, the default value is 11 bits.

ProductMode

This property, available when your filter is in fixed-point arithmetic mode, specifies how the filter outputs the results of multiplication operations. All `dfilt` objects include this property when they use fixed-point arithmetic.

When available, you select from one of the following values for `ProductMode`:

- `FullPrecision` — means the filter automatically chooses the word length and fraction length it uses to represent the results of multiplication operations. The setting allow the product to retain the precision provided by the inputs (multiplicands) to the operation.
- `KeepMSB` — means you specify the word length for representing product operation results. The filter sets the fraction length to discard the LSBs, keep the higher order bits in the data, and maintain the precision.
- `KeepLSB` — means you specify the word length for representing the product operation results. The filter sets the fraction length to discard the MSBs, keep the lower order bits, and maintain the precision. Compare to the `KeepMSB` option.
- `SpecifyPrecision` — means you specify the word length and the fraction length to apply to data output from product operations.

When you switch to fixed-point filtering from floating-point, you are most likely going to throw away some data bits after product operations in your filter, perhaps because you have limited resources. When you have to discard some bits, you might choose to discard the least significant bits (LSB) from a result since the resulting quantization error would be small as the LSBs carry less weight. Or you might choose to keep the LSBs because the results have MSBs that are mostly zero, such as when your values are small relative to the range of the format in which they are represented. So the options for `ProductMode` let you choose how to maintain the information you need from the accumulator.

For more information about data formats, word length, and fraction length in fixed-point arithmetic, refer to “Notes About Fraction Length, Word Length, and Precision” on page 6-26.

ProductWordLength

You use `ProductWordLength` to define the data word length used by the output from multiplication operations. Set this property to a value that matches your intended application. For example, the default value is 32 bits, but you can set any word length.

```
set(hq,'arithmetic','fixed');
set(hq,'ProductWordLength',64);
```

Note that `ProductWordLength` applies only to filters whose `Arithmetic` property value is `fixed`.

PersistentMemory

Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter object. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false` — the filter does not retain memory about filtering operations from one to the next. Maintaining memory (setting `PersistentMemory` to `true`) lets you filter large data sets as collections of smaller subsets and get the same result.

In this example, filter `hd` first filters data `xtot` in one pass. Then you can use `hd` to filter `x` as two separate data sets. The results `ytot` and `ysec` are the same in both cases.

```
xtot=[x,x];
ytot=filter(hd,xtot)
ytot =

         0   -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
reset(hm1);  % Clear history of the filter
hm1.PersistentMemory='true';
ysec=[filter(hd,x) filter(hd,x)]

ysec =

         0   -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
```

This test verifies that `ysec` (the signal filtered by sections) is equal to `ytot` (the entire signal filtered at once).

RoundMode

The `RoundMode` property value specifies the rounding method used for quantizing numerical values. Specify the `RoundMode` property values as one of the following five strings.

RoundMode String	Description of Rounding Algorithm
`Ceiling`	Round toward positive infinity.
`Floor`	Round toward negative infinity.

RoundMode String	Description of Rounding Algorithm
Nearest	Round toward nearest. Ties round toward positive infinity.
Nearest(Convergent)	Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.
Round	Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.
Zero	Round toward zero.

The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

ScaleValueFracLength

Filter structures `df1sos`, `df1tsos`, `df2sos`, and `df2tsos` that use `fixed` arithmetic have this property that defines the fraction length applied to the scale values the filter uses between sections. In combination with `CoeffWordLength`, these two properties fully specify how the filter interprets and uses the scale values stored in the property `ScaleValues`. As with fraction length properties, `ScaleValueFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter.

ScaleValues

The `ScaleValues` property values are specified as a scalar (or vector) that introduces scaling for inputs (and the outputs from cascaded sections in the vector case) during filtering:

- When you only have a single section in your filter:
  - Specify the `ScaleValues` property value as a scalar if you only want to scale the input to your filter.
  - Specify the `ScaleValues` property as a vector of length 2 if you want to specify scaling to the input (scaled with the first entry in the vector) and the output (scaled with the last entry in the vector).

- When you have L cascaded sections in your filter:
  - Specify the ScaleValues property value as a scalar if you only want to scale the input to your filter.
  - Specify the value for the ScaleValues property as a vector of length L+1 if you want to scale the inputs to every section in your filter, along with the output:

    The first entry of your vector specifies the input scaling

    Each successive entry specifies the scaling at the output of the next section

    The final entry specifies the scaling for the filter output.

The default value for ScaleValues is 0.

The interpretation of this property is described as follows with diagrams in "Interpreting the ScaleValues Property" on page 6-75.

Note: The value of the ScaleValues property is not quantized. Data affected by the presence of a scaling factor in the filter is quantized according to the appropriate data format.

When you apply normalize to a fixed-point filter, the value for the ScaleValues property is changed accordingly.

It is good practice to choose values for this property that are either positive or negative powers of two.

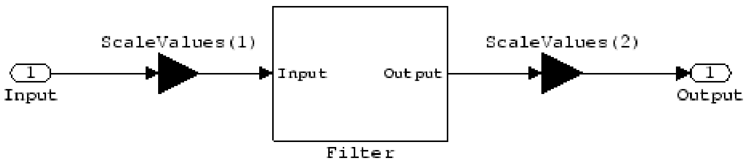
Interpreting the ScaleValues Property

When you specify the values of the ScaleValues property of a quantized filter, the values are entered as a vector, the length of which is determined by the number of cascaded sections in your filter:

- When you have only one section, the value of the Scalevalues property can be a scalar or a two-element vector.
- When you have L cascaded sections in your filter, the value of the ScaleValues property can be a scalar or an L+1-element vector.

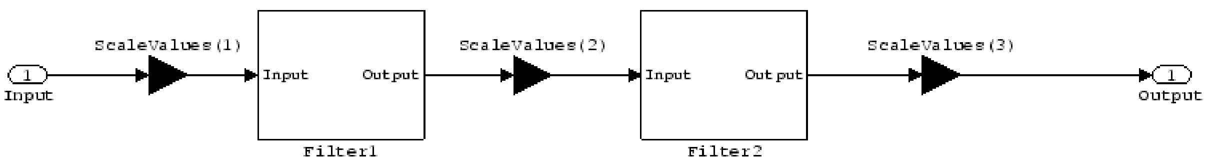
The following diagram shows how the ScaleValues property values are applied to a quantized filter with only one section.

Application of ScaleValues to a Single Section



The following diagram shows how the `ScaleValues` property values are applied to a quantized filter with two sections.

Application of ScaleValues to Multiple Sections



Signed

When you create a `dfilt` object for fixed-point filtering (you set the property `Arithmetic` to `fixed`, the property `Signed` specifies whether the filter interprets coefficients as signed or unsigned. This setting applies only to the coefficients. While the default setting is `true`, meaning that all coefficients are assumed to be signed, you can change the setting to `false` after you create the fixed-point filter.

For example, create a fixed-point direct-form II transposed filter with both negative and positive coefficients, and then change the property value for `Signed` from `true` to `false` to see what happens to the negative coefficient values.

```
hd=dfilt.df2t(-5:5)

hd =

         FilterStructure: 'Direct-Form II Transposed'
              Arithmetic: 'double'
               Numerator: [-5 -4 -3 -2 -1 0 1 2 3 4 5]
```

```
            Denominator: 1
       PersistentMemory: false
                 States: [10x1 double]

set(hd,'arithmetic','fixed')
hd.numerator

ans =

    -5    -4    -3    -2    -1     0
     1     2     3     4     5

set(hd,'signed',false)
hd.numerator

ans =

     0     0     0     0     0     0
     1     2     3     4     5
```

Using unsigned coefficients limits you to using only positive coefficients in your filter. `Signed` is a dynamic property — you cannot set or change it until you switch the setting for the `Arithmetic` property to `fixed`.

SosMatrix

When you convert a `dfilt` object to second-order section form, or create a second-order section filter, `sosMatrix` holds the filter coefficients as property values. Using the `double` data type by default, the matrix is in [sections coefficients per section] form, displayed as `[15-x-6]` for filters with 6 coefficients per section and 15 sections, [15 6].

To demonstrate, the following code creates an order 30 filter using second-order sections in the direct-form II transposed configuration. Notice the `sosMatrix` property contains the coefficients for all the sections.

```
d = fdesign.lowpass('n,fc',30,0.5);
hd = butter(d);

hd =

  FilterStructure: 'Direct-Form II, Second-Order Sections'
       Arithmetic: 'double'
        sosMatrix: [15x6 double]
      ScaleValues: [16x1 double]
```

```
PersistentMemory: false
          States: [2x15 double]

hd.arithmetic='fixed'

hd =

 FilterStructure: 'Direct-Form II, Second-Order Sections'
      Arithmetic: 'fixed'
       sosMatrix: [15x6 double]
     ScaleValues: [16x1 double]
PersistentMemory: false
                  States: [1x1 embedded.fi]

 CoeffWordLength: 16
  CoeffAutoScale: true
          Signed: true

 InputWordLength: 16
 InputFracLength: 15

  SectionInputWordLength: 16
   SectionInputAutoScale: true

 SectionOutputWordLength: 16
  SectionOutputAutoScale: true

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

         StateWordLength: 16
         StateFracLength: 15

             ProductMode: 'FullPrecision'

         AccumWordLength: 40
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'

hd.sosMatrix

ans =
```

```
    1.0000    2.0000    1.0000    1.0000         0    0.9005
    1.0000    2.0000    1.0000    1.0000         0    0.7294
    1.0000    2.0000    1.0000    1.0000         0    0.5888
    1.0000    2.0000    1.0000    1.0000         0    0.4724
    1.0000    2.0000    1.0000    1.0000         0    0.3755
    1.0000    2.0000    1.0000    1.0000         0    0.2948
    1.0000    2.0000    1.0000    1.0000         0    0.2275
    1.0000    2.0000    1.0000    1.0000         0    0.1716
    1.0000    2.0000    1.0000    1.0000         0    0.1254
    1.0000    2.0000    1.0000    1.0000         0    0.0878
    1.0000    2.0000    1.0000    1.0000         0    0.0576
    1.0000    2.0000    1.0000    1.0000         0    0.0344
    1.0000    2.0000    1.0000    1.0000         0    0.0173
    1.0000    2.0000    1.0000    1.0000         0    0.0062
    1.0000    2.0000    1.0000    1.0000         0    0.0007
```

The SOS matrix is an M-by-6 matrix, where M is the number of sections in the second-order section filter. Filter `hd` has M equal to 15 as shown above (15 rows). Each row of the SOS matrix contains the numerator and denominator coefficients (b's and a's) and the scale factors of the corresponding section in the filter.

SectionInputAutoScale

Second-order section filters include this property that determines who the filter handles data in the transitions from one section to the next in the filter.

How the filter represents the data passing from one section to the next depends on the property value of `SectionInputAutoScale`. The representation the filter uses between the filter sections depends on whether the value of `SectionInputAutoScale` is `true` or `false`.

- `SectionInputAutoScale` = `true` means the filter chooses the fraction length to maintain the value of the data between sections as close to the output values from the previous section as possible. `true` is the default setting.
- `SectionInputAutoScale` = `false` removes the automatic scaling of the fraction length for the intersection data and exposes the property that controls the coefficient fraction length (`SectionInputFracLength`) so you can change it. For example, if the filter is a second-order, direct form FIR filter, setting `SectionInputAutoScale` to `false` exposes the `SectionInputFracLength` property that specifies the fraction length applied to data between the sections.

SectionInputFracLength

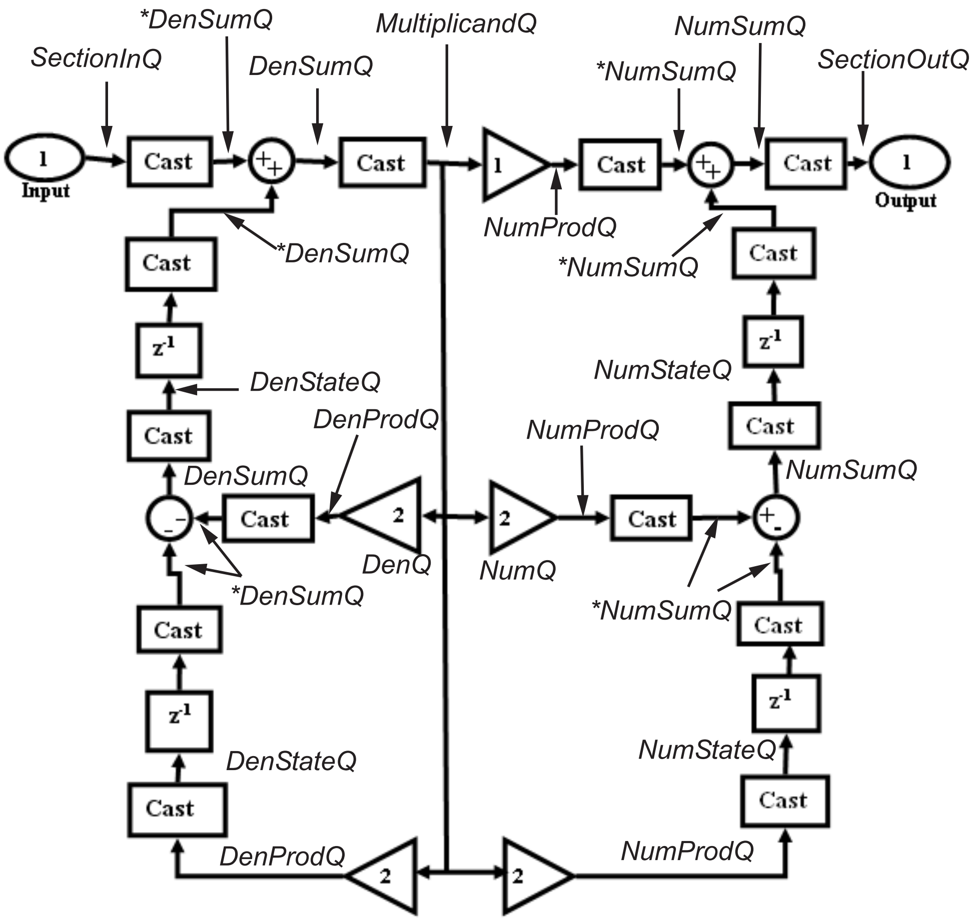
Second-order section filters use quantizers at the input to each section of the filter. The quantizers apply to the input data entering each filter section. Note that the quantizers for each section are the same. To set the fraction length for interpreting the input values, use the property value in `SectionInputFracLength`.

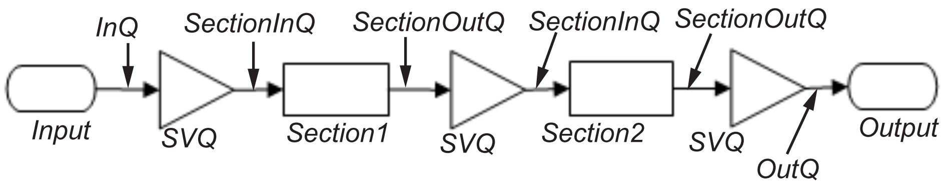
In combination with `CoeffWordLength`, `SectionInputFracLength` fully determines how the filter interprets and uses the state values stored in the property `States`. As with all word and fraction length properties, `SectionInputFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter object.

SectionInputWordLength

SOS filters are composed of sections, each one a second-order filter. Filtering data input to the filter involves passing the data through each filter section. `SectionInputWordLength` specifies the word length applied to data as it enters one filter section from the previous section. Only second-order implementations of direct-form I transposed and direct-form II transposed filters include this property.

The following diagram shows an SOS filter composed of sections (the bottom part of the diagram) and a possible internal structure of each Section (the top portion of the diagram), in this case — a direct form I transposed second order sections filter structure. Note that the output of each section is fed through a multiplier. If the gain of the multiplier `=1`, then the last Cast block of the Section is ignored, and the format of the output is NumSumQ.


*DenSumQ
MultiplicandQ
NumSumQ
SectionInQ
DenSumQ
*NumSumQ
SectionOutQ
1
Input
Cast
Cast
1
Cast
Cast
1
Output
NumProdQ
*DenSumQ
*NumSumQ
Cast
Cast
z-1
z-1
DenStateQ
NumStateQ
Cast
DenProdQ
NumProdQ
Cast
DenSumQ
NumSumQ
Cast
2
2
Cast
DenQ
NumQ
*DenSumQ
*NumSumQ
Cast
Cast
z-1
z-1
DenStateQ
NumStateQ
Cast
Cast
DenProdQ
2
2
NumProdQ
* : only if CastBeforeSum is True


InQ
SectionInQ
SectionOutQ
SectionInQ
SectionOutQ
Input
SVQ
Section1
SVQ
Section2
SVQ
OutQ
Output

SectionInputWordLength defaults to 16 bits.

SectionOutputAutoScale

Second-order section filters include this property that determines who the filter handles data in the transitions from one section to the next in the filter.

How the filter represents the data passing from one section to the next depends on the property value of SectionOutputAutoScale. The representation the filter uses between the filter sections depends on whether the value of SectionOutputAutoScale is true or false.

- SectionOutputAutoScale = true means the filter chooses the fraction length to maintain the value of the data between sections as close to the output values from the previous section as possible. true is the default setting.
- SectionOutputAutoScale = false removes the automatic scaling of the fraction length for the intersection data and exposes the property that controls the coefficient fraction length (SectionOutputFracLength) so you can change it. For example, if the filter is a second-order, direct form FIR filter, setting SectionOutputAutoScale = false exposes the SectionOutputFracLength property that specifies the fraction length applied to data between the sections.

SectionOutputFracLength

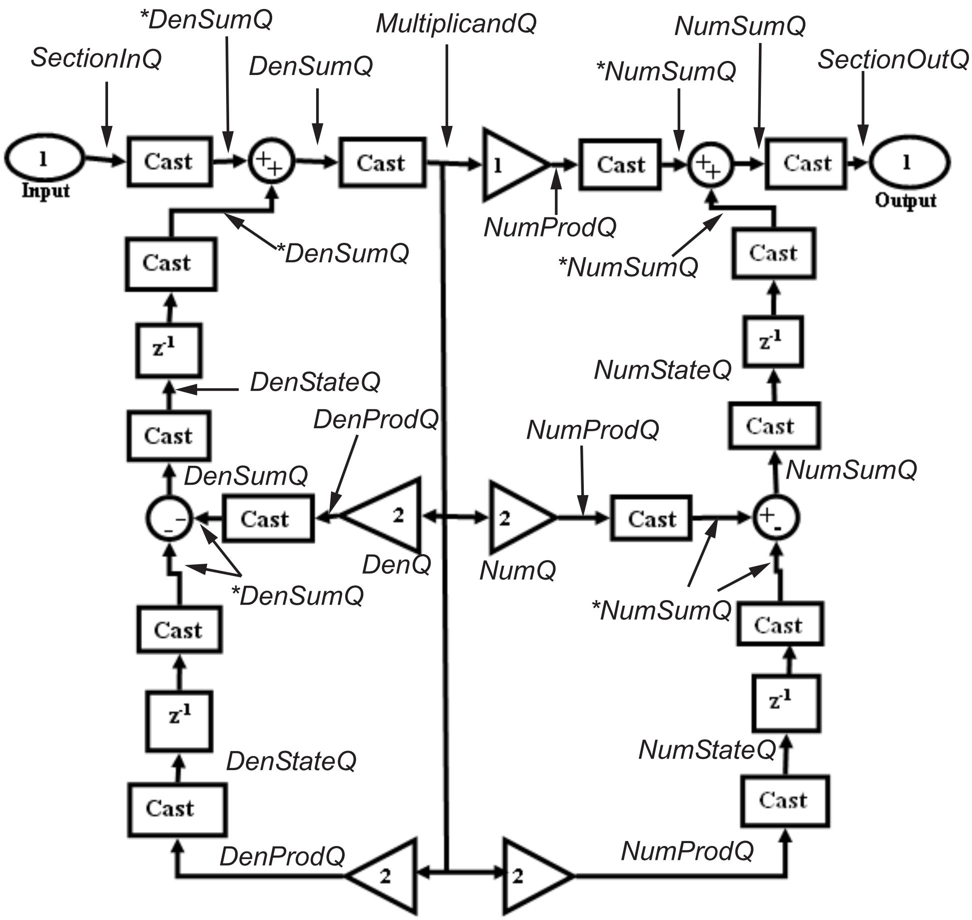
Second-order section filters use quantizers at the output from each section of the filter. The quantizers apply to the output data leaving each filter section. Note that the quantizers for each section are the same. To set the fraction length for interpreting the output values, use the property value in SectionOutputFracLength.

In combination with CoeffWordLength, SectionOutputFracLength determines how the filter interprets and uses the state values stored in the property States. As with all fraction length properties, SectionOutputFracLength can be any integer, including integers larger than CoeffWordLength, and positive or negative integers. 15 bits is the default value when you create the filter object.

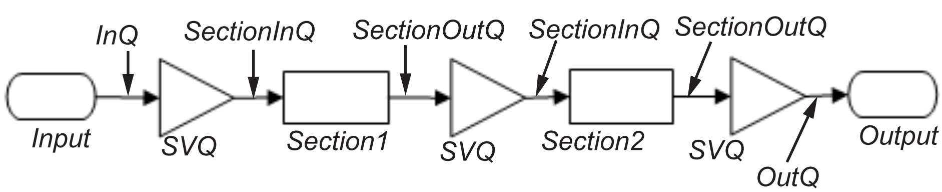
SectionOutputWordLength

SOS filters are composed of sections, each one a second-order filter. Filtering data input to the filter involves passing the data through each filter section. SectionOutputWordLength specifies the word length applied to data as it leaves one filter section to go to the next. Only second-order implementations direct-form I transposed and direct-form II transposed filters include this property.

The following diagram shows an SOS filter composed of sections (the bottom part of the diagram) and a possible internal structure of each Section (the top portion of the diagram), in this case — a direct form I transposed second order sections filter structure. Note that the output of each section is fed through a multiplier. If the gain of the multiplier =1, then the last Cast block of the Section is ignored, and the format of the output is NumSumQ.



* : only if CastBeforeSum is True



SectionOutputWordLength defaults to 16 bits.

StateAutoScale

Although all filters use states, some do not allow you to choose whether the filter automatically scales the state values to prevent overruns or bad arithmetic errors. You select either of the following settings:

- StateAutoScale = true means the filter chooses the fraction length to maintain the value of the states as close to the double-precision values as possible. When you change the word length applied to the states (where allowed by the filter structure), the filter object changes the fraction length to try to accommodate the change. true is the default setting.
- StateAutoScale = false removes the automatic scaling of the fraction length for the states and exposes the property that controls the coefficient fraction length so you can change it. For example, in a direct form I transposed SOS FIR filter, setting StateAutoScale = false exposes the NumStateFracLength and DenStateFracLength properties that specify the fraction length applied to states.

Each of the following filter structures provides the StateAutoScale property:

- df1t
- df1tsos
- df2t
- df2tsos
- dffirt

Other filter structures do not include this property.

StateFracLength

Filter states stored in the property States have both word length and fraction length. To set the fraction length for interpreting the stored filter object state values, use the property value in StateFracLength.

In combination with CoeffWordLength, StateFracLength fully determines how the filter interprets and uses the state values stored in the property States.

As with all fraction length properties, StateFracLength can be any integer, including integers larger than CoeffWordLength, and positive or negative integers. 15 bits is the default value when you create the filter object.

States

Digital filters are dynamic systems. The behavior of dynamic systems (their response) depends on the input (stimulus) to the system and the current or previous *state* of the system. You can say the system has memory or inertia. All fixed- or floating-point digital filters (as well as analog filters) have states.

Filters use the states to compute the filter output for each input sample, as well using them while filtering in loops to maintain the filter state between loop iterations. This toolbox assumes zero-valued initial conditions (the dynamic system is at rest) by default when you filter the first input sample. Assuming the states are zero initially does not mean the states are not used; they are, but arithmetically they do not have any effect.

Filter objects store the state values in the property `States`. The number of stored states depends on the filter implementation, since the states represent the delays in the filter implementation.

When you review the display for a filter object with fixed arithmetic, notice that the states return an embedded `fi` object, as you see here.

```
b = ellip(6,3,50,300/500);
hd=dfilt.dffir(b)

hd =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'double'
               Numerator: [0.0773 0.2938 0.5858 0.7239 0.5858 0.2938 0.0773]
        PersistentMemory: false
                  States: [6x1 double]


hd.arithmetic='fixed'

hd =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'fixed'
               Numerator: [0.0773 0.2938 0.5858 0.7239 0.5858 0.2938 0.0773]
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: 'on'
                  Signed: 'on'

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

             ProductMode: 'FullPrecision'
```

```
AccumWordLength: 40
  CastBeforeSum: 'on'

      RoundMode: 'convergent'
   OverflowMode: 'wrap'

InheritSettings: 'off'
```

fi objects provide fixed-point support for the filters. To learn more about the details about fi objects, refer to your Fixed-Point Designer documentation.

The property States lets you use a fi object to define how the filter interprets the filter states. For example, you can create a fi object in MATLAB, then assign the object to States, as follows:

```
statefi=fi([],16,12)

statefi =

[]
   DataTypeMode = Fixed-point: binary point scaling
         Signed = true
     Wordlength = 16
 Fractionlength = 12
```

This fi object does not have a value associated (notice the [] input argument to fi for the value), and it has word length of 16 bits and fraction length of 12 bit. Now you can apply statefi to the States property of the filter hd.

```
set(hd,'States',statefi);
Warning: The 'States' property will be reset to the value
specified at construction before filtering.
Set the 'PersistentMemory' flag to 'True'
to avoid changing this property value.
hd

hd =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'fixed'
               Numerator: [0.0773 0.2938 0.5858 0.7239 0.5858
                           0.2938 0.0773]
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: 'on'
```

```
             Signed: 'on'

    InputWordLength: 16
    InputFracLength: 15

   OutputWordLength: 16
         OutputMode: 'AvoidOverflow'

        ProductMode: 'FullPrecision'
    AccumWordLength: 40
      CastBeforeSum: 'on'

          RoundMode: 'convergent'
       OverflowMode: 'wrap'
```

StateWordLength

While all filters use states, some do not allow you to directly change the state representation — the word length and fraction lengths — independently. For the others, `StateWordLength` specifies the word length, in bits, the filter uses to represent the states. Filters that do not provide direct state word length control include:

- `df1`
- `dfasymfir`
- `dffir`
- `dfsymfir`

For these structures, the filter derives the state format from the input format you choose for the filter — except for the `df1` IIR filter. In this case, the numerator state format comes from the input format and the denominator state format comes from the output format. All other filter structures provide control of the state format directly.

TapSumFracLength

Direct-form FIR filter objects, both symmetric and antisymmetric, use this property. To set the fraction length for output from the sum operations that involve the filter tap weights, use the property value in `TapSumFracLength`. To enable this property, set the `TapSumMode` to `SpecifyPrecision` in your filter.

As you can see in this code example that creates a fixed-point asymmetric FIR filter, the `TapSumFracLength` property becomes available after you change the `TapSumMode` property value.

```
hd=dfilt.dfasymfir

hd =

         FilterStructure: 'Direct-Form Antisymmetric FIR'
              Arithmetic: 'double'
               Numerator: 1
        PersistentMemory: false
                  States: [0x1 double]

set(hd,'arithmetic','fixed');
hd

hd =

         FilterStructure: 'Direct-Form Antisymmetric FIR'
              Arithmetic: 'fixed'
               Numerator: 1
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

              TapSumMode: 'KeepMSB'
        TapSumWordLength: 17

             ProductMode: 'FullPrecision'

         AccumWordLength: 40

           CastBeforeSum: true
               RoundMode: 'convergent'
            OverflowMode: 'wrap'
```

With the filter now in fixed-point mode, you can change the `TapSumMode` property value to `SpecifyPrecision`, which gives you access to the `TapSumFracLength` property.

```
set(hd,'TapSumMode','SpecifyPrecision');
hd

hd =

         FilterStructure: 'Direct-Form Antisymmetric FIR'
              Arithmetic: 'fixed'
               Numerator: 1
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

              TapSumMode: 'SpecifyPrecision'
        TapSumWordLength: 17
        TapSumFracLength: 15

             ProductMode: 'FullPrecision'

         AccumWordLength: 40

           CastBeforeSum: true
               RoundMode: 'convergent'
            OverflowMode: 'wrap'
```

In combination with `TapSumWordLength`, `TapSumFracLength` fully determines how the filter interprets and uses the state values stored in the property `States`.

As with all fraction length properties, `TapSumFracLength` can be any integer, including integers larger than `TapSumWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter object.

TapSumMode

This property, available only after your filter is in fixed-point mode, specifies how the filter outputs the results of summation operations that involve the filter tap weights.

Only symmetric (`dfilt.dfsymfir`) and antisymmetric (`dfilt.dfasymfir`) FIR filters use this property.

When available, you select from one of the following values:

- `FullPrecision` — means the filter automatically chooses the word length and fraction length to represent the results of the sum operation so they retain all of the precision provided by the inputs (addends).
- `KeepMSB` — means you specify the word length for representing tap sum summation results to keep the higher order bits in the data. The filter sets the fraction length to discard the LSBs from the sum operation. This is the default property value.
- `KeepLSB` — means you specify the word length for representing tap sum summation results to keep the lower order bits in the data. The filter sets the fraction length to discard the MSBs from the sum operation. Compare to the `KeepMSB` option.
- `SpecifyPrecision` — means you specify the word and fraction lengths to apply to data output from the tap sum operations.

TapSumWordLength

Specifies the word length the filter uses to represent the output from tap sum operations. The default value is 17 bits. Only `dfasymfir` and `dfsymfir` filters include this property.

Adaptive Filter Properties

In this section...
"Compatibility" on page 6-92
"Property Summaries" on page 6-92
"Property Details for Adaptive Filter Properties" on page 6-96
"References for Adaptive Filters" on page 6-103

Compatibility

`adaptfilt` objects will be removed in a future release. Refer to the reference page for a specific `adaptfilt` object to see its recommended replacement.

Property Summaries

The following table summarizes the adaptive filter properties and provides a brief description of each. Full descriptions of each property, in alphabetical order, are given in the subsequent section.

Property	Description
`Algorithm`	Reports the algorithm the object uses for adaptation. When you construct your adaptive filter object, this property is set automatically by the constructor, such as `adaptfilt.nlms` creating an adaptive filter that uses the normalized LMS algorithm. You cannot change the value — it is read only.
`AvgFactor`	Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. `AvgFactor` should lie between zero and one. For default filter objects, `AvgFactor` equals (1 - `step`). `lambda` is the input argument that represents `AvgFactor`
`BkwdPredErrorPower`	Returns the minimum mean-squared prediction error. Refer to [3] in the bibliography for details about linear prediction.

Property	Description
BkwdPrediction	Returns the predicted samples generated during adaptation. Refer to [3] in the bibliography for details about linear prediction.
Blocklength	Block length for the coefficient updates. This must be a positive integer such that (l/blocklength) is also an integer. For faster execution, blocklength should be a power of two. blocklength defaults to two.
Coefficients	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
ConversionFactor	Conversion factor defaults to the matrix [1 -1] that specifies soft-constrained initialization. This is the gamma input argument for some of the fast transversal algorithms.
Delay	Update delay given in time samples. This scalar should be a positive integer—negative delays do not work. delay defaults to 1 for most algorithms.
DesiredSignalStates	Desired signal states of the adaptive filter. dstates defaults to a zero vector with length equal to (blocklen - 1) or (swblocklen - 1) depending on the algorithm.
EpsilonStates	Vector of the epsilon values of the adaptive filter. EpsilonStates defaults to a vector of zeros with (projectord - 1) elements.
ErrorStates	Vector of the adaptive filter error states. ErrorStates defaults to a zero vector with length equal to (projectord - 1).
FFTCoefficients	Stores the discrete Fourier transform of the filter coefficients in coeffs.
FFTStates	Stores the states of the FFT of the filter coefficients during adaptation.
FilteredInputStates	Vector of filtered input states with length equal to l - 1.

Property	Description
`FilterLength`	Contains the length of the filter. Note that this is not the filter order. Filter length is 1 greater than filter order. Thus a filter with length equal to 10 has filter order equal to 9.
`ForgettingFactor`	Determines how the RLS adaptive filter uses past data in each iteration. You use the forgetting factor to specify whether old data carries the same weight in the algorithm as more recent data.
`FwdPredErrorPower`	Returns the minimum mean-squared prediction error in the forward direction. Refer to [3] in the bibliography for details about linear prediction.
`FwdPrediction`	Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.
`InitFactor`	Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. Called `delta` as an input argument, this defaults to one.
`InvCov`	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix. Dimensions are `l`-by-`l`, where `l` is the filter length.
`KalmanGain`	Empty when you construct the object, this gets populated after you run the filter.
`KalmanGainStates`	Contains the states of the Kalman gain updates during adaptation.
`Leakage`	Contains the setting for leakage in the adaptive filter algorithm. Using a leakage factor that is not 1 forces the weights to adapt even when they have found the minimum error solution. Forcing the adaptation can improve the numerical performance of the LMS algorithm.
`OffsetCov`	Contains the offset covariance matrix.

Property	Description
Offset	Specifies an optional offset for the denominator of the step size normalization term. You must specify offset to be a scalar greater than or equal to zero. Nonzero offsets can help avoid a divide-by-near-zero condition that causes errors.
Power	A vector of 2*l elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.
ProjectionOrder	Projection order of the affine projection algorithm. projectord defines the size of the input signal covariance matrix and defaults to two.
ReflectionCoeffs	Coefficients determined for the reflection portion of the filter during adaptation.
ReflectionCoeffsStep	Size of the steps used to determine the reflection coefficients.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states and coefficients from previous filtering runs.
SecondaryPathCoeffs	A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor.
SecondaryPathEstimate	An estimate of the secondary path filter model.
SecondaryPathStates	The states of the secondary path filter, the unknown system.
SqrtCov	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.
SqrtInvCov	Square root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.

Property	Description
States	Vector of the adaptive filter states. states defaults to a vector of zeros whose length depends on the chosen algorithm. Usually the length is a function of the filter length l and another input argument to the filter object, such as projectord.
StepSize	Reports the size of the step taken between iterations of the adaptive filter process. Each adaptfilt object has a default value that best meets the needs of the algorithm.
SwBlockLength	Block length of the sliding window. This integer must be at least as large as the filter length. swblocklen defaults to 16.

Like dfilt objects, adaptfilt objects have properties that govern their behavior and store some of the results of filtering operations. The following pages list, in alphabetical order, the name of every property associated with adaptfilt objects. Note that not all adaptfilt objects have all of these properties. To view the properties of a particular adaptive filter, such as an adaptfilt.bap filter, use get with the object handle, like this:

```
ha = adaptfilt.bap(32,0.5,4,1.0);
get(ha)
        PersistentMemory: false
               Algorithm: 'Block Affine Projection FIR Adaptive Filter'
            FilterLength: 32
            Coefficients: [1x32 double]
                  States: [35x1 double]
                StepSize: 0.5000
         ProjectionOrder: 4
               OffsetCov: [4x4 double]
```

get shows you the properties for ha and the values for the properties. Entering the object handle returns the same values and properties without the formatting of the list and the more familiar property names.

Property Details for Adaptive Filter Properties

Algorithm

Reports the algorithm the object uses for adaptation. When you construct you adaptive filter object, this property is set automatically. You cannot change the value—it is read only.

AvgFactor

Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. `AvgFactor` should lie between zero and one. For default filter objects, `AvgFactor` equals (1 - `step`). `lambda` is the input argument that represent `AvgFactor`

BkwdPredErrorPower

Returns the minimum mean-squared prediction error in the backward direction. Refer to [3] in the bibliography for details about linear prediction.

BkwdPrediction

When you use an adaptive filter that does backward prediction, such as `adaptfilt.ftf`, one property of the filter contains the backward prediction coefficients for the adapted filter. With these coefficient, the forward coefficients, and the system under test, you have the full set of knowledge of how the adaptation occurred. Two values stored in properties compose the `BkwdPrediction` property:

- Coefficients, which contains the coefficients of the system under test, as determined using backward predictions process.
- Error, which is the difference between the filter coefficients determined by backward prediction and the actual coefficients of the sample filter. In this example, `adaptfilt.ftf` identifies the coefficients of an unknown FIR system.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
N  = 31;               % Adaptive filter order
lam = 0.99;            % RLS forgetting factor
del = 0.1;             % Soft-constrained initialization factor
ha = adaptfilt.ftf(32,lam,del);
[y,e] = filter(ha,x,d);

ha

ha =

              Algorithm: 'Fast Transversal Least-Squares Adaptive Filter'
           FilterLength: 32
           Coefficients: [1x32 double]
                 States: [31x1 double]
       ForgettingFactor: 0.9900
             InitFactor: 0.1000
          FwdPrediction: [1x1 struct]
         BkwdPrediction: [1x1 struct]
             KalmanGain: [32x1 double]
```

```
        ConversionFactor: 0.7338
        KalmanGainStates: [32x1 double]
        PersistentMemory: false

ha.coefficients

ans =

  Columns 1 through 8

   -0.0055    0.0048    0.0045    0.0146   -0.0009    0.0002   -0.0019    0.0008

  Columns 9 through 16

   -0.0142   -0.0226    0.0234    0.0421   -0.0571   -0.0807    0.1434    0.4620

  Columns 17 through 24

    0.4564    0.1532   -0.0879   -0.0501    0.0331    0.0361   -0.0266   -0.0220

  Columns 25 through 32

    0.0231    0.0026   -0.0063   -0.0079    0.0032    0.0082    0.0033    0.0065

ha.bkwdprediction

ans =

    Coeffs: [1x32 double]
     Error: 82.3394

>> ha.bkwdprediction.coeffs

ans =

  Columns 1 through 8

    0.0067    0.0186    0.1114   -0.0150   -0.0239   -0.0610   -0.1120   -0.1026

  Columns 9 through 16

    0.0093   -0.0399   -0.0045    0.0622    0.0997    0.0778    0.0646   -0.0564

  Columns 17 through 24

    0.0775    0.0814    0.0057    0.0078    0.1271   -0.0576    0.0037   -0.0200

  Columns 25 through 32

   -0.0246    0.0180   -0.0033    0.1222    0.0302   -0.0197   -0.1162    0.0285
```

Blocklength

Block length for the coefficient updates. This must be a positive integer such that (`l/blocklen`) is also an integer. For faster execution, `blocklen` should be a power of two. `blocklen` defaults to two.

Coefficients

Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.

ConversionFactor

Conversion factor defaults to the matrix [1 -1] that specifies soft-constrained initialization. This is the `gamma` input argument for some of the fast transversal algorithms.

Delay

Update delay given in time samples. This scalar should be a positive integer — negative delays do not work. `delay` defaults to 1 for most algorithms.

DesiredSignalStates

Desired signal states of the adaptive filter. `dstates` defaults to a zero vector with length equal to (`blocklen` - 1) or (`swblocklen` - 1) depending on the algorithm.

EpsilonStates

Vector of the epsilon values of the adaptive filter. `EpsilonStates` defaults to a vector of zeros with (`projectord` - 1) elements.

ErrorStates

Vector of the adaptive filter error states. `ErrorStates` defaults to a zero vector with length equal to (`projectord` - 1).

FFTCoefficients

Stores the discrete Fourier transform of the filter coefficients in `coeffs`.

FFTStates

Stores the states of the FFT of the filter coefficients during adaptation.

FilteredInputStates

Vector of filtered input states with length equal to `l` - 1.

FilterLength

Contains the length of the filter. Note that this is not the filter order. Filter length is 1 greater than filter order. Thus a filter with length equal to 10 has filter order equal to 9.

ForgettingFactor

Determines how the RLS adaptive filter uses past data in each iteration. You use the forgetting factor to specify whether old data carries the same weight in the algorithm as more recent data.

This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting `forgetting factor = 1` denotes infinite memory while adapting to find the new filter. Note that this is the `lambda` input argument.

FwdPredErrorPower

Returns the minimum mean-squared prediction error in the forward direction. Refer to [3] in the bibliography for details about linear prediction.

FwdPrediction

Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.

InitFactor

Returns the soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. `delta` defaults to one.

InvCov

Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix. Dimensions are `l`-by-`l`, where `l` is the filter length.

KalmanGain

Empty when you construct the object, this gets populated after you run the filter.

KalmanGainStates

Contains the states of the Kalman gain updates during adaptation.

Leakage

Contains the setting for leakage in the adaptive filter algorithm. Using a leakage factor that is not 1 forces the weights to adapt even when they have found the minimum error solution. Forcing the adaptation can improve the numerical performance of the LMS algorithm.

OffsetCov

Contains the offset covariance matrix.

Offset

Specifies an optional offset for the denominator of the step size normalization term. You must specify offset to be a scalar greater than or equal to zero. Nonzero offsets can help avoid a divide-by-near-zero condition that causes errors.

Use this to avoid dividing by zero or by very small numbers when input signal amplitude becomes very small, or dividing by very small numbers when any of the FFT input signal powers become very small. `offset` defaults to one.

Power

A vector of 2*`l` elements, each initialized with the value `delta` from the input arguments. As you filter data, `Power` gets updated by the filter process.

ProjectionOrder

Projection order of the affine projection algorithm. `projectord` defines the size of the input signal covariance matrix and defaults to two.

ReflectionCoeffs

For adaptive filters that use reflection coefficients, this property stores them.

ReflectionCoeffsStep

As the adaptive filter changes coefficient values during adaptation, the step size used between runs is stored here.

PersistentMemory

Determines whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter.

PersistentMemory returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to false.

SecondaryPathCoeffs

A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor.

SecondaryPathEstimate

An estimate of the secondary path filter model.

SecondaryPathStates

The states of the secondary path filter, the unknown system.

SqrtCov

Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.

SqrtInvCov

Square root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.

States

Vector of the adaptive filter states. states defaults to a vector of zeros whose length depends on the chosen algorithm. Usually the length is a function of the filter length l and another input argument to the filter object, such as projectord.

StepSize

Reports the size of the step taken between iterations of the adaptive filter process. Each adaptfilt object has a default value that best meets the needs of the algorithm.

SwBlockLength

Block length of the sliding window. This integer must be at least as large as the filter length. swblocklength defaults to 16.

References for Adaptive Filters

[1] Griffiths, L.J., *A Continuously Adaptive Filter Implemented as a Lattice Structure*, Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, Hartford, CT, pp. 683-686, 1977.

[2] Hayes, M.H., *Statistical Digital Signal Processing and Modeling*, John Wiley and Sons, 1996.

[3] Haykin, S., *Adaptive Filter Theory*, Third Edition, Prentice-Hall, Inc., 1996.

Multirate Filter Properties

In this section...
"Compatibility" on page 6-104
"Property Summaries" on page 6-104
"Property Details for Multirate Filter Properties" on page 6-108
"References for Multirate Filters" on page 6-118

Compatibility

`mfilt` objects will be removed in a future release. Refer to the reference page for a specific `mfilt` object to see its recommended replacement.

Property Summaries

The following table summarizes the multirate filter properties and provides a brief description of each. Full descriptions of each property are given in the subsequent section.

Name	Values	Default	Description
`BlockLength`	Positive integers	100	Length of each block of data input to the FFT used in the filtering. `fftfirinterp` multirate filters include this property.
`DecimationFactor`	Any positive integer	2	Amount to reduce the input sampling rate.
`DifferentialDelay`	Any integer	1	Sets the differential delay for the filter. Usually a value of one or two is appropriate.
`FilterInternals`	`FullPrecision`, `MinWordlengths`, `SpecifyWordLengths` `SpecifyPrecision`	`FullPrecision`	Controls whether the filter sets the output word and fraction lengths, and the accumulator word and fraction lengths automatically to maintain

Name	Values	Default	Description
			the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` exposes the output and accumulator related properties so you can set your own word and fraction lengths for them.
`FilterStructure`	`mfilt` structure string	None	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output. You cannot set this property — it is always read only and results from your choice of `mfilt` object.
`InputOffset`	Integers	0	Contains the number of input data samples processed without generating an output sample. `InputOffset = mod(length(nx),m)` where `nx` is the number of input samples that have been processed so far and m is the decimation factor.
`InterpolationFactor`	Positive integers	2	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate.

Name	Values	Default	Description
`NumberOfSections`	Any positive integer	2	Number of sections used in the decimator, or in the comb and integrator portions of CIC filters.
`Numerator`	Array of double values	No default values	Vector containing the coefficients of the FIR lowpass filter used for interpolation.
`OverflowMode`	saturate, [wrap]	wrap	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.

Name	Values	Default	Description
PolyphaseAccum	Values depend on filter type. Either double, single, or fixed-point object	0	Stores the value remaining in the accumulator after the filter processes the last input sample. The stored value for PolyphaseAccum affects the next output when PersistentMemory is true and InputOffset is not equal to 0. Always provides full precision values. Compare the AccumWordLength and AccumFracLength.
PersistentMemory	false or true	false	Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.
RateChangeFactors	[l,m]	[2,3] or [3,2]	Reports the decimation (m) and interpolation (l) factors for the filter object. Combining these factors results in the final rate change for the signal. The default changes depending on whether the filter decimates or interpolates.

Name	Values	Default	Description
States	Any m+1-by-n matrix of double values	2-by-2 matrix, int32	Stored conditions for the filter, including values for the integrator and comb sections. n is the number of filter sections and m is the differential delay. Stored in a filtstates object.
SpecifyWordLengths	Vector of integers	[16 16 16 16] bits	
WordLengthPerSection	Any integer or a vector of length 2*n	16	Defines the word length used in each section while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter WordLengthPerSection as a scalar or vector of length 2*n, where n is the number of sections. When WordLengthPerSection is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the decimator.

The following sections provide details about the properties that govern the way multirate filter work. Creating any multirate filter object puts in place a number of these properties. The following pages list the mfilt object properties in alphabetical order.

Property Details for Multirate Filter Properties

BitsPerSection

Any integer or a vector of length 2*n.

Defines the bits per section used while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using `wrap` arithmetic). Enter `bps` as a scalar or vector of length 2*`n`, where `n` is the number of sections. When `bps` is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the decimator.

BlockLength

Length of each block of input data used in the filtering.

`mfilt.fftfirinterp` objects process data in blocks whose length is determined by the value you set for the `BlockLength` property. By default the property value is 100. When you set the `BlockLength` value, try choosing a value so that [`BlockLength` + `length`(filter order)] is a power of two.

Larger block lengths generally reduce the computation time.

DecimationFactor

Decimation factor for the filter. `m` specifies the amount to reduce the sampling rate of the input signal. It must be an integer. You can enter any integer value. The default value is 2.

DifferentialDelay

Sets the differential delay for the filter. Usually a value of one or two is appropriate. While you can set any value, the default is one and the maximum is usually two.

FilterInternals

Similar to the FilterInternals pane in FDATool, this property controls whether the filter sets the output word and fraction lengths automatically, and the accumulator word and fraction lengths automatically as well, to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. Setting `FilterInternals` to `SpecifyPrecision` exposes the output and accumulator related properties so you can set your own word and fraction lengths for them.

About FilterInternals Mode

There are four usage modes for this that you set using the `FilterInternals` property in multirate filters.

- `FullPrecision` — All word and fraction lengths set to B_{max} + 1, called B_{accum} by Fred Harris in [2]. Full precision is the default setting.
- `MinWordLengths` — Minimum Word Lengths
- `SpecifyWordLengths` — Specify Word Lengths
- `SpecifyPrecision` — Specify Precision

Full Precision

In full precision mode, the word lengths of all sections and the output are set to B_{accum} as defined by

$$B_{accum} = ceil(N_{\mathrm{secs}}(Log_2(D \times M)) + InputWordLength)$$

where N_{secs} is the number of filter sections.

Section fraction lengths and the fraction length of the output are set to the input fraction length.

Here is the display looks for this mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'FullPrecision'
```

Minimum Word Lengths

In minimum word length mode, you control the output word length explicitly. When the output word length is less than B_{accum}, roundoff noise is introduced at the output of the filter. Hogenauer's bit pruning theory (refer to [3] in the following References section) states that one valid design criterion is to make the word lengths of the different sections of the filter smaller than B_{accum} as well, so that the roundoff noise introduced by all sections does not exceed the roundoff noise introduced at the output.

In this mode, the design calculates the word lengths of each section to meet the Hogenauer criterion. The algorithm subtracts the number of bits computed using eq. 21 in Hogenauer's paper from B_{accum} to determine the word length each section.

To compute the fraction lengths of the different sections, the algorithm notes that the bits thrown out for this word length criterion are least significant bits (LSB), therefore each bit thrown out at a particular section decrements the fraction length of that section by one bit compared to the input fraction length. Setting the output word length for the filter automatically sets the output fraction length as well.

Here is the display for this mode:

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'MinWordLengths'

OutputWordLength: 16
```

Specify Word Lengths

In this mode, the design algorithm discards the LSBs, adjusting the fraction length so that unrecoverable overflow does not occur, always producing a reasonable output.

You can specify the word lengths for all sections and the output, but you cannot control the fraction lengths for those quantities.

To specify the word lengths, you enter a vector of length 2*(`NumberOfSections`), where each vector element represents the word length for a section. If you specify a scalar, such as B_{accum}, the full-precision output word length, the algorithm expands that scalar to a vector of the appropriate size, applying the scalar value to each section.

The CIC design does not check that the specified word lengths are monotonically decreasing. There are some cases where the word lengths are not necessarily monotonically decreasing, for example

```
hcic=mfilt.cicdecim;
hcic.FilterInternals='minwordlengths';
hcic.Outputwordlength=14;
```

which are valid CIC filters but the word lengths do not decrease monotonically across the sections.

Here is the display looks like for the `SpecifyWordLengths` mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyWordLengths'

SectionWordLengths: [19 18 18 17]

OutputWordLength: 16
```

Specify Precision

In this mode, you have full control over the word length and fraction lengths of all sections and the filter output.

When you elect the `SpecifyPrecision` mode, you must enter a vector of length 2*(`NumberOfSections`) with elements that represent the word length for each section. When you enter a scalar such as B_{accum}, the CIC algorithm expands that scalar to a vector of the appropriate size and applies the scalar value to each section and the output. The design does not check that this vector is monotonically decreasing.

Also, you must enter a vector of length 2*(`NumberOfSections`) with elements that represent the fraction length for each section as well. When you enter a scalar such as B_{accum}, the design applies scalar expansion as done for the word lengths.

Here is the `SpecifyPrecision` display.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
```

```
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyPrecision'

SectionWordLengths: [19 18 18 17]
SectionFracLengths: [14 13 13 12]

OutputWordLength: 16
OutputFracLength: 11
```

FilterStructure

Reports the type of filter object, such as a decimator or fractional integrator. You cannot set this property — it is always read only and results from your choice of `mfilt` object. Because of the length of the names of multirate filters, `FilterStructure` often returns a vector specification for the string. For example, when you use `mfilt.firfracinterp` to design a filter, `FilterStructure` returns as [1x49 char].

```
hm=mfilt.firfracinterp

hm =

         FilterStructure: [1x49 char]
               Numerator: [1x72 double]
       RateChangeFactors: [3 2]
        PersistentMemory: false
                  States: [24x1 double]
```

InputOffset

When you decimate signals whose length is not a multiple of the decimation factor M, the last samples — (nM +1) to [(n+1)(M) -1], where *n* is an integer — are processed and used to track where the filter stopped processing input data and when to expect the next output sample. If you think of the filtering process as generating an output for a block of input data, `InputOffset` contains a count of the number of samples in the last incomplete block of input data.

Note `InputOffset` applies only when you set `PersistentMemory` to `true`. Otherwise, `InputOffset` is not available for you to use.

Two different cases can arise when you decimate a signal:

1. The input signal is a multiple of the filter decimation factor. In this case, the filter processes the input samples and generates output samples for all inputs as determined by the decimation factor. For example, processing 99 input samples with a filter that decimates by three returns 33 output samples.
2. The input signal is not a multiple of the decimation factor. When this occurs, the filter processes all of the input samples, generates output samples as determined by the decimation factor, and has one or more input samples that were processed but did not generate an output sample.

   For example, when you filter 100 input samples with a filter which has decimation factor of 3, you get 33 output samples, and 1 sample that did not generate an output. In this case, `InputOffset` stores the value 1 after the filter run.

   `InputOffset` equal to 1 indicates that, if you divide your input signal into blocks of data with length equal to your filter decimation factor, the filter processed one sample from a new (incomplete) block of data. Subsequent inputs to the filter are concatenated with this single sample to form the next block of length m.

One way to define the value stored in `InputOffset` is

```
InputOffset = mod(length(nx),m)
```

where `nx` is the number of input samples in the data set and `m` is the decimation factor.

Storing `InputOffset` in the filter allows you to stop filtering a signal at any point and start over from there, provided that the `PersistentMemory` property is set to `true`. Being able to resume filtering after stopping a signal lets you break large data sets in to smaller pieces for filtering. With `PersistentMemory` set to `true` and the `InputOffset` property in the filter, breaking a signal into sections of arbitrary length and filtering the sections is equivalent to filtering the entire signal at once.

```
xtot=[x,x];
ytot=filter(hm1,xtot)
ytot =

         0   -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
reset(hm1);  % Clear history of the filter
```

```
hm1.PersistentMemory='true';
ysec=[filter(hm1,x) filter(hm1,x)]

ysec =

         0   -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
```

This test verifies that `ysec` (the signal filtered by sections) is equal to `ytot` (the entire signal filtered at once).

InterpolationFactor

Amount to increase the sampling rate. Interpolation factor for the filter. It specifies the amount to increase the input sampling rate. It must be an integer. Two is the default value. You may use any positive value.

NumberOfSections

Number of sections used in the multirate filter. By default multirate filters use two sections, but any positive integer works.

OverflowMode

The `OverflowMode` property is specified as one of the following two strings indicating how to respond to overflows in fixed-point arithmetic:

- `'saturate'` — saturate overflows.

  When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the applicable word length and fraction length properties), these values are quantized to the value of either the largest or smallest representable value, depending on which is closest.
- `'wrap'` — wrap all overflows to the range of representable values.

  When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the data format properties), these values are wrapped back into that range using modular arithmetic relative to the smallest representable number. You can learn more about modular arithmetic in Fixed-Point Designer documentation.

These rules apply to the `OverflowMode` property.

- Applies to the accumulator and output data only.

- Does not apply to coefficients or input data. These always saturate the results.
- Does not apply to products. Products maintain full precision at all times. Your filters do not lose precision in the products.

Default value

: 'saturate'

Note Numbers in floating-point filters that extend beyond the dynamic range overflow to ±inf.

PolyphaseAccum

The idea behind PolyphaseAccum and AccumWordLength/AccumFracLength is to distinguish between the adders that always work in full precision (PolyphaseAccum) from the others [the adders that are controlled by the user (through AccumWordLength and AccumFracLength) and that may introduce quantization effects when you set property FilterInternals to SpecifyPrecision.

Given a product format determined by the input word and fraction lengths, and the coefficients word and fraction lengths, doing full precision accumulation means allowing enough guard bits to avoid overflows and underflows.

Property PolyphaseAccum stores the value that was in the accumulator the last time your filter ran out of input samples to process. The default value for PolyphaseAccum affects the next output only if PersistentMemory is true and InputOffset is not equal to 0.

PolyphaseAccum stores data in the format for the filter arithmetic. Double-precision filters store doubles in PolyphaseAccum. Single-precision filter store singles in PolyphaseAccum. Fixed-point filters store fi objects in PolyphaseAccum.

PersistentMemory

Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.

Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter object. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `true` — the filter retains memory about filtering operations from one to the next. Maintaining memory lets you filter large data sets as collections of smaller subsets and get the same result.

```
xtot=[x,x];
ytot=filter(hm1,xtot)
ytot =

         0   -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
reset(hm1);  % Clear history of the filter
hm1.PersistentMemory='true';
ysec=[filter(hm1,x) filter(hm1,x)]

ysec =

         0   -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
```

This test verifies that `ysec` (the signal filtered by sections) is equal to `ytot` (the entire signal filtered at once).

RateChangeFactors

Reports the decimation (`m`) and interpolation (`l`) factors for the filter object when you create fractional integrators and decimators, although `m` and `l` are used as arguments to both decimators and integrators, applying the same meaning. Combining these factors as input arguments to the fractional decimator or integrator results in the final rate change for the signal.

For decimating filters, the default is [2,3]. For integrators, [3,2].

States

Stored conditions for the filter, including values for the integrator and comb sections. `m` is the differential delay and `n` is the number of sections in the filter.

About the States of Multirate Filters

In the `states` property you find the states for both the integrator and comb portions of the filter, stored in a `filtstates` object. `states` is a matrix of dimensions `m`+1-by-`n`, with the states in CIC filters apportioned as follows:

- States for the integrator portion of the filter are stored in the first row of the state matrix.

- States for the comb portion fill the remaining rows in the state matrix.

In the state matrix, state values are specified and stored in `double` format.

States stores conditions for the delays between each interpolator phase, the filter states, and the states at the output of each phase in the filter, including values for the interpolator and comb states.

The number of states is (`lh`-1)*`m`+(l-1)*(`lo`+`mo`) where `lh` is the length of each subfilter, and `l` and `m` are the interpolation and decimation factors. `lo` and `mo`, the input and output delays between each interpolation phase, are integers from Euclid's theorem such that `lo`*l-`mo`*`m` = -1 (refer to the reference for more details). Use `euclidfactors` to get `lo` and `mo` for an `mfilt.firfracdecim` object.

`States` defaults to a vector of zeros that has length equal to `nstates`(`hm`)

References for Multirate Filters

[1] Fliege, N.J., *Multirate Digital Signal Processing*, John Wiley and Sons, 1994.

[2] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.

[3] Hogenauer, E. B., “An Economical Class of Digital Filters for Decimation and Interpolation,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-29, No. 2, April 1981, pp. 155-162.

[4] Lyons, Richard G., *Understanding Digital Signal Processing*, Prentice Hall PTR, 2004

[5] Mitra, S.K., *Digital Signal Processing*, McGraw-Hill, 1998.

[6] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Inc., 1996.

Glossary

This glossary defines terms related to fixed-point data types and numbers. These terms may appear in some or all of the documents that describe MathWorks® products that have fixed-point support.

arithmetic shift
Shift of the bits of a binary word for which the sign bit is recycled for each bit shift to the right. A zero is incorporated into the least significant bit of the word for each bit shift to the left. In the absence of overflows, each arithmetic shift to the right is equivalent to a division by 2, and each arithmetic shift to the left is equivalent to a multiplication by 2.

See also binary point, binary word, bit, logical shift, most significant bit

bias
Part of the numerical representation used to interpret a fixed-point number. Along with the slope, the bias forms the scaling of the number. Fixed-point numbers can be represented as

$$\textit{real-world value} = (\textit{slope} \times \textit{stored integer}) + \textit{bias}$$

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}$$

See also fixed-point representation, fractional slope, integer, scaling, slope, [Slope Bias]

binary number
Value represented in a system of numbers that has two as its base and that uses 1's and 0's (bits) for its notation.

See also bit

binary point
Symbol in the shape of a period that separates the integer and fractional parts of a binary number. Bits to the left of the binary point are integer bits and/or sign bits, and bits to the right of the binary point are fractional bits.

See also binary number, bit, fraction, integer, radix point

binary point-only scaling

Scaling of a binary number that results from shifting the binary point of the number right or left, and which therefore can only occur by powers of two.

See also binary number, binary point, scaling

binary word

Fixed-length sequence of bits (1's and 0's). In digital hardware, numbers are stored in binary words. The way in which hardware components or software functions interpret this sequence of 1's and 0's is described by a data type.

See also bit, data type, word

bit

Smallest unit of information in computer software or hardware. A bit can have the value 0 or 1.

ceiling (round toward)

Rounding mode that rounds to the closest representable number in the direction of positive infinity. This is equivalent to the `ceil` mode in Fixed-Point Designer software.

See also convergent rounding, floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)

contiguous binary point

Binary point that occurs within the word length of a data type. For example, if a data type has four bits, its contiguous binary point must be understood to occur at one of the following five positions:

.0000
0.000
00.00
000.0
0000.

See also data type, noncontiguous binary point, word length

convergent rounding

Rounding mode that rounds to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 0.

See also ceiling (round toward), floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)

data type

Set of characteristics that define a group of values. A fixed-point data type is defined by its word length, its fraction length, and whether it is signed or unsigned. A floating-point data type is defined by its word length and whether it is signed or unsigned.

See also fixed-point representation, floating-point representation, fraction length, signedness, word length

data type override

Parameter in the Fixed-Point Tool that allows you to set the output data type and scaling of fixed-point blocks on a system or subsystem level.

See also data type, scaling

exponent

Part of the numerical representation used to express a floating-point or fixed-point number.

1. Floating-point numbers are typically represented as

$$real\text{-}world\ value = mantissa \times 2^{exponent}$$

2. Fixed-point numbers can be represented as

$$real\text{-}world\ value = (slope \times stored\ integer) + bias$$

where the slope can be expressed as

$$slope = fractional\ slope \times 2^{exponent}$$

The exponent of a fixed-point number is equal to the negative of the fraction length:

$$exponent = -1 \times fraction\ length$$

See also bias, fixed-point representation, floating-point representation, fraction length, fractional slope, integer, mantissa, slope

fixed-point representation

Method for representing numerical values and data types that have a set range and precision.

1. Fixed-point numbers can be represented as

$$real\text{-}world\ value = (slope \times stored\ integer) + bias$$

where the slope can be expressed as

$$slope = fractional\ slope \times 2^{exponent}$$

The slope and the bias together represent the scaling of the fixed-point number.

2. Fixed-point data types can be defined by their word length, their fraction length, and whether they are signed or unsigned.

See also bias, data type, exponent, fraction length, fractional slope, integer, precision, range, scaling, slope, word length

floating-point representation

Method for representing numerical values and data types that can have changing range and precision.

1. Floating-point numbers can be represented as

$$real\text{-}world\ value = mantissa \times 2^{exponent}$$

2. Floating-point data types are defined by their word length.

See also data type, exponent, mantissa, precision, range, word length

floor (round toward)

Rounding mode that rounds to the closest representable number in the direction of negative infinity.

See also ceiling (round toward), convergent rounding, nearest (round toward), rounding, truncation, zero (round toward)

fraction

Part of a fixed-point number represented by the bits to the right of the binary point. The fraction represents numbers that are less than one.

See also binary point, bit, fixed-point representation

fraction length

Number of bits to the right of the binary point in a fixed-point representation of a number.

See also binary point, bit, fixed-point representation, fraction

fractional slope

Part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as

$$real\text{-}world\ value = (slope \times stored\ integer) + bias$$

where the slope can be expressed as

$$slope = fractional\ slope \times 2^{exponent}$$

The term *slope adjustment* is sometimes used as a synonym for fractional slope.

See also bias, exponent, fixed-point representation, integer, slope

full range

The broadest range available for a data type. From $-\infty$ to ∞ for floating-point types. For integer types, the representable range is the range from the smallest to largest integer value (finite) the type can represent. For example, from `-128` to `127` for a signed 8–bit integer. Also known as representable range.

guard bits

Extra bits in either a hardware register or software simulation that are added to the high end of a binary word to ensure that no information is lost in case of overflow.

See also binary word, bit, overflow

incorrect range

A range that is too restrictive and does not include values that can actually occur in the model element. A range that is too broad is not considered incorrect because it will not lead to overflow.

See also range analysis

integer

1. Part of a fixed-point number represented by the bits to the left of the binary point. The integer represents numbers that are greater than or equal to one.

2. Also called the "stored integer." The raw binary number, in which the binary point is assumed to be at the far right of the word. The integer is part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as

$$real\text{-}world\ value = 2^{-fraction\ length} \times stored\ integer$$

or

$$real\text{-}world\ value = (slope \times stored\ integer) + bias$$

where the slope can be expressed as

$$slope = fractional\ slope \times 2^{exponent}$$

See also bias, fixed-point representation, fractional slope, integer, real-world value, slope

integer length

Number of bits to the left of the binary point in a fixed-point representation of a number.

See also binary point, bit, fixed-point representation, fraction length, integer

least significant bit (LSB)

Bit in a binary word that can represent the smallest value. The LSB is the rightmost bit in a big-endian-ordered binary word. The weight of the LSB is related to the fraction length according to

$$weight\ of\ LSB = 2^{-fraction\ length}$$

See also big-endian, binary word, bit, most significant bit

logical shift

Shift of the bits of a binary word, for which a zero is incorporated into the most significant bit for each bit shift to the right and into the least significant bit for each bit shift to the left.

See also arithmetic shift, binary point, binary word, bit, most significant bit

mantissa

Part of the numerical representation used to express a floating-point number. Floating-point numbers are typically represented as

$$real\text{-}world\ value = mantissa \times 2^{exponent}$$

See also exponent, floating-point representation

model element

Entities in a model that range analysis software tracks, for example, blocks, signals, parameters, block internal data (such as accumulators, products).

See also range analysis

most significant bit (MSB)

Bit in a binary word that can represent the largest value. The MSB is the leftmost bit in a big-endian-ordered binary word.

See also binary word, bit, least significant bit

nearest (round toward)

Rounding mode that rounds to the closest representable number, with the exact midpoint rounded to the closest representable number in the direction of positive infinity. This is equivalent to the `nearest` mode in Fixed-Point Designer software.

See also ceiling (round toward), convergent rounding, floor (round toward), rounding, truncation, zero (round toward)

noncontiguous binary point

Binary point that is understood to fall outside the word length of a data type. For example, the binary point for the following 4-bit word is understood to occur two bits to the right of the word length,

0000__.

thereby giving the bits of the word the following potential values:

$2^5 2^4 2^3 2^2$ __.

See also binary point, data type, word length

one's complement representation

Representation of signed fixed-point numbers. Negating a binary number in one's complement requires a bitwise complement. That is, all 0's are flipped to 1's and all 1's are flipped to 0's. In one's complement notation there are two ways to represent zero. A binary word of all 0's represents "positive" zero, while a binary word of all 1's represents "negative" zero.

See also binary number, binary word, sign/magnitude representation, signed fixed-point, two's complement representation

overflow

Situation that occurs when the magnitude of a calculation result is too large for the range of the data type being used. In many cases you can choose to either saturate or wrap overflows.

See also saturation, wrapping

padding

Extending the least significant bit of a binary word with one or more zeros.

See also least significant bit

precision

1. Measure of the smallest numerical interval that a fixed-point data type and scaling can represent, determined by the value of the number's least significant bit. The precision is given by the slope, or the number of fractional bits. The term *resolution* is sometimes used as a synonym for this definition.

2. Measure of the difference between a real-world numerical value and the value of its quantized representation. This is sometimes called quantization error or quantization noise.

See also data type, fraction, least significant bit, quantization, quantization error, range, slope

Q format

Representation used by Texas Instruments™ to encode signed two's complement fixed-point data types. This fixed-point notation takes the form

$Qm.n$

where

- Q indicates that the number is in Q format.
- m is the number of bits used to designate the two's complement integer part of the number.
- n is the number of bits used to designate the two's complement fractional part of the number, or the number of bits to the right of the binary point.

In Q format notation, the most significant bit is assumed to be the sign bit.

See also binary point, bit, data type, fixed-point representation, fraction, integer, two's complement

quantization

Representation of a value by a data type that has too few bits to represent it exactly.

See also bit, data type, quantization error

quantization error

Error introduced when a value is represented by a data type that has too few bits to represent it exactly, or when a value is converted from one data type to a shorter data type. Quantization error is also called quantization noise.

See also bit, data type, quantization

radix point

Symbol in the shape of a period that separates the integer and fractional parts of a number in any base system. Bits to the left of the radix point are integer and/or sign bits, and bits to the right of the radix point are fraction bits.

See also binary point, bit, fraction, integer, sign bit

range

Span of numbers that a certain data type can represent.

See also data type, full range, precision, representable range

range analysis

Static analysis of model to derive minimum and maximum range values for elements in the model. The software statically analyzes the ranges of the individual computations in the model based on specified design ranges, inputs, and the semantics of the calculation.

real-world value

Stored integer value with fixed-point scaling applied. Fixed-point numbers can be represented as

$$real\text{-}world\ value = 2^{-fraction\ length} \times stored\ integer$$

or

$$real\text{-}world\ value = (slope \times stored\ integer) + bias$$

where the slope can be expressed as

$$slope = fractional\ slope \times 2^{exponent}$$

See also integer

representable range

The broadest range available for a data type. From –∞ to ∞ for floating-point types. For integer types, the representable range is the range from the smallest to largest integer value (finite) the type can represent. For example, from -128 to 127 for a signed 8–bit integer. Also known as full range.

resolution

See **precision**

rounding

Limiting the number of bits required to express a number. One or more least significant bits are dropped, resulting in a loss of precision. Rounding is necessary when a value cannot be expressed exactly by the number of bits designated to represent it.

See also bit, ceiling (round toward), convergent rounding, floor (round toward), least significant bit, nearest (round toward), precision, truncation, zero (round toward)

saturation

Method of handling numeric overflow that represents positive overflows as the largest positive number in the range of the data type being used, and negative overflows as the largest negative number in the range.

See also overflow, wrapping

scaled double

A double data type that retains fixed-point scaling information. For example, in Simulink and Fixed-Point Designer software you can use data type override to convert your fixed-point data types to scaled doubles. You can then simulate to determine the ideal floating-point behavior of your system. After you gather that information you can turn data type override off to return to fixed-point data types, and your quantities still have their original scaling information because it was held in the scaled double data types.

scaling

1. Format used for a fixed-point number of a given word length and signedness. The slope and bias together form the scaling of a fixed-point number.

2. Changing the slope and/or bias of a fixed-point number without changing the stored integer.

See also bias, fixed-point representation, integer, slope

shift

Movement of the bits of a binary word either toward the most significant bit ("to the left") or toward the least significant bit ("to the right"). Shifts to the right can be either logical, where the spaces emptied at the front of the word with each shift are filled in with zeros, or arithmetic, where the word is sign extended as it is shifted to the right.

See also arithmetic shift, logical shift, sign extension

sign bit

Bit (or bits) in a signed binary number that indicates whether the number is positive or negative.

See also binary number, bit

sign extension

Addition of bits that have the value of the most significant bit to the high end of a two's complement number. Sign extension does not change the value of the binary number.

See also binary number, guard bits, most significant bit, two's complement representation, word

sign/magnitude representation

Representation of signed fixed-point or floating-point numbers. In sign/magnitude representation, one bit of a binary word is always the dedicated sign bit, while the remaining bits of the word encode the magnitude of the number. Negation using sign/magnitude representation consists of flipping the sign bit from 0 (positive) to 1 (negative), or from 1 to 0.

See also binary word, bit, fixed-point representation, floating-point representation, one's complement representation, sign bit, signed fixed-point, signedness, two's complement representation

signed fixed-point

Fixed-point number or data type that can represent both positive and negative numbers.

See also data type, fixed-point representation, signedness, unsigned fixed-point

signedness

The signedness of a number or data type can be signed or unsigned. Signed numbers and data types can represent both positive and negative values, whereas unsigned numbers and data types can only represent values that are greater than or equal to zero.

See also data type, sign bit, sign/magnitude representation, signed fixed-point, unsigned fixed-point

slope

Part of the numerical representation used to express a fixed-point number. Along with the bias, the slope forms the scaling of a fixed-point number. Fixed-point numbers can be represented as

$$real\text{-}world\ value = (slope \times stored\ integer) + bias$$

where the slope can be expressed as

$$slope = fractional\ slope \times 2^{exponent}$$

See also bias, fixed-point representation, fractional slope, integer, scaling, [Slope Bias]

slope adjustment

See **fractional slope**

[Slope Bias]

Representation used to define the scaling of a fixed-point number.

See also bias, scaling, slope

stored integer

See **integer**

trivial scaling

Scaling that results in the real-world value of a number being simply equal to its stored integer value:

$$real\text{-}world\ value = stored\ integer$$

In [Slope Bias] representation, fixed-point numbers can be represented as

$$real\text{-}world\ value = (slope \times stored\ integer) + bias$$

In the trivial case, slope = 1 and bias = 0.

In terms of binary point-only scaling, the binary point is to the right of the least significant bit for trivial scaling, meaning that the fraction length is zero:

$$real\text{-}world\ value = stored\ integer \times 2^{-fraction\ length} = stored\ integer \times 2^{0}$$

Scaling is always trivial for pure integers, such as `int8`, and also for the true floating-point types `single` and `double`.

See also bias, binary point, binary point-only scaling, fixed-point representation, fraction length, integer, least significant bit, scaling, slope, [Slope Bias]

truncation

Rounding mode that drops one or more least significant bits from a number.

See also ceiling (round toward), convergent rounding, floor (round toward), nearest (round toward), rounding, zero (round toward)

two's complement representation
Common representation of signed fixed-point numbers. Negation using signed two's complement representation consists of a translation into one's complement followed by the binary addition of a one.

See also binary word, one's complement representation, sign/magnitude representation, signed fixed-point

unsigned fixed-point
Fixed-point number or data type that can only represent numbers greater than or equal to zero.

See also data type, fixed-point representation, signed fixed-point, signedness

word
Fixed-length sequence of binary digits (1's and 0's). In digital hardware, numbers are stored in words. The way hardware components or software functions interpret this sequence of 1's and 0's is described by a data type.

See also binary word, data type

word length
Number of bits in a binary word or data type.

See also binary word, bit, data type

wrapping
Method of handling overflow. Wrapping uses modulo arithmetic to cast a number that falls outside of the representable range the data type being used back into the representable range.

See also data type, overflow, range, saturation

zero (round toward)
Rounding mode that rounds to the closest representable number in the direction of zero. This is equivalent to the `fix` mode in Fixed-Point Designer software.

See also ceiling (round toward), convergent rounding, floor (round toward), nearest (round toward), rounding, truncation